Francesca Saglietti
Norbert Oster (Eds.)

Computer Safety, Reliability, and Security

LNCS 4680

26th International Conference, SAFECOMP 2007
Nuremberg, Germany, September 2007
Proceedings

Springer

Lecture Notes in Computer Science 4680

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Francesca Saglietti Norbert Oster (Eds.)

Computer Safety, Reliability, and Security

26th International Conference, SAFECOMP 2007
Nuremberg, Germany, September 18-21, 2007
Proceedings

Springer

Volume Editors

Francesca Saglietti
Department of Software Engineering
University of Erlangen-Nuremberg
Germany
E-mail: saglietti@informatik.uni-erlangen.de

Norbert Oster
Department of Software Engineering
University of Erlangen-Nuremberg
Germany
E-mail: oster@informatik.uni-erlangen.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.1-4, E.4, C.3, F.3, K.6.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-75100-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75100-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12161898 06/3180 5 4 3 2 1 0

Preface

Since 1979, when it was first established by the Technical Committee on Reliability, Safety and Security of the European Workshop on Industrial Computer Systems (EWICS TC7), the SAFECOMP Conference series has regularly and continuously contributed to improving the state of the art of highly dependable computer-based systems, since then increasingly applied to safety-relevant industrial domains.

In this expanding technical field SAFECOMP offers a platform for knowledge and technology transfer between academia, industry, research and licensing institutions, providing ample opportunities for exchanging insights, experiences and trends in the areas of safety, reliability and security regarding critical computer applications. In accordance with the growing spread of critical infrastructures involving both safety and security threats, this year's SAFECOMP program included a considerable number of contributions addressing technical problems and engineering solutions across the border between safety-related and security-related concerns.

The reaction to our call for papers was particularly gratifying and impressive, including 136 full papers submitted by authors representing 29 countries from Europe, Asia, North and South America as well as Australia. The selection of 33 full papers and 16 short papers for presentation and publication was a challenging task requiring a huge amount of reviewing and organizational effort. In view of the particularly high number of articles submitted, obvious practical constraints led – to our regret – to the rejection of a considerable amount of high-quality work. To all authors, invited speakers, members of the International Program Committee and external reviewers go our heartfelt thanks!

The local organization of SAFECOMP 2007, hosted in Nuremberg, is also gratefully acknowledged. The intensive preparatory activities demanded year-long dedication from the members of the Department of Software Engineering at the University of Erlangen-Nuremberg, which co-organized the event in co-operation with the German Computer Society (Gesellschaft für Informatik). Particular thanks are due to all colleagues and friends from the Organizing Committee, whose support we regard as crucial for the success of this conference.

We are confident that – when reading the present volume of the *Lecture Notes in Computer Science* – you will find its contents interesting enough to consider joining the SAFECOMP community. In the name of EWICS TC7 and of the future organizers we welcome you and invite you to attend future SAFECOMP conferences – among them SAFECOMP 2008 in Newcastle upon Tyne (UK) – and to contribute actively to their technical program.

July 2007

Francesca Saglietti
Norbert Oster

Organization

Program Chair

Francesca Saglietti (Germany)

EWICS Chair

Udo Voges (Germany)

International Program Committee

Stuart Anderson (UK)
Robin Bloomfield (UK)
Sandro Bologna (Italy)
Jens Braband (Germany)
Inga Bratteby-Ribbing (SE)
Bettina Buth (Germany)
Peter Daniel (UK)
Christian Diedrich (Germany)
Jana Dittmann (Germany)
Wolfgang Ehrenberger (Germany)
Massimo Felici (UK)
Robert Genser (Austria)
Bjorn Axel Gran (Norway)
Karl-Erwin Großpietsch (Germany)
Wolfgang Halang (Germany)
Monika Heiner (Germany)
Maritta Heisel (Germany)
Constance Heitmeyer (USA)
Janusz Gorski (Poland)
Karl-Heinz John (Germany)
Karama Kanoun (France)
Floor Koornneef (The Netherlands)
Peter B. Ladkin (Germany)
Søren Lindskov Hansen (Denmark)
Bev Littlewood (UK)
Vic Maggioli (USA)
Odd Nordland (Norway)
Gerd Rabe (Germany)
Felix Redmill (UK)
Martin Rothfelder (Germany)
Krzysztof Sacha (Poland)
Erwin Schoitsch (Austria)
Werner Stephan (Germany)
Mark Sujan (UK)
Pascal Traverse (France)
Jos Trienekens (The Netherlands)
Meine Van der Meulen (The Netherlands)
Udo Voges (Germany)
Albrecht Weinert (Germany)
Rune Winther (Norway)
Stefan Wittmann (Belgium)
Zdzislaw Zurakowski (Poland)

Organizing Committee

Francesca Saglietti (Co-chair)
Wolfgang Ehrenberger (Co-chair)
Norbert Oster
Jutta Radke
Gerd Schober
Sven Söhnlein

External Reviewers

Myla Archer
Lassaad Cheikhrouhou
DeJiu Chen
Yves Crouzet
Håkan Edler
Jonas Elmqvist
Denis Hatebur
Tobias Hoppe
Ralph D. Jeffords
Björn Johansson
Johan Karlsson
Stefan Kiltz
Andreas Lang
Bruno Langenstein
Tiejun Ma
Oliver Meyer
M. Oliver Möller
Simin Nadjm-Tehrani
Vincent Nicomette
Andreas Nonnengart
Andrea Oermann
Ulf Olsson
Norbert Oster
Christian Raspotnig
Ronny Richter
Georg Rock
Jan Sanders
Thomas Santen
Tobias Scheidat
Holger Schmidt
Bernd Schomburg
Martin Schwarick
Dirk Seifert
Sven Söhnlein
Marc Spisländer
Mirco Tribastone
Arno Wacker
Dirk Wischermann

Scientific Sponsor

EWICS – European Workshop on Industrial Computer Systems
TC7 – Technical Committee on Reliability, Safety and Security

in collaboration with the following **Scientific Co-sponsors**:

IFAC
International Federation of Automatic Control

IFIP
International Federation for Information Processing

ENCRESS
European Network of Clubs for Reliability
and Safety of Software-Intensive Systems

SCSC
The Safety-Critical Systems Club

SRMC
The Software Reliability & Metrics Club

OCG
Austrian Computer Society

DECOS
Dependable Embedded Components and Systems

SAFECOMP 2007 Organizers

SWE
Department of Software Engineering
University of Erlangen-Nuremberg

GI
Gesellschaft für Informatik e.V.

Table of Contents

Fault Tree Analysis

Safety Analysis

Security Aspects

Poster Session 2

Verification and Validation

Platform Reliability

Reliability Evaluation

Poster Session 3

Formal Methods

Static Code Analysis

Safety-Related Architectures

Establishing Evidence for Safety Cases in Automotive Systems – A Case Study

Willem Ridderhof[1], Hans-Gerhard Gross[2], and Heiko Doerr[3]

[1] ISPS Medical Software, Rotterdamseweg 145, 2628 AL Delft
willem.ridderhof@isps-medical-software.nl
[2] Embedded Software Laboratory, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
h.g.gross@tudelft.nl
[3] CARMEQ GmbH, Carnotstr. 4, 10587 Berlin, Germany
heiko.doerr@carmeq.com

Abstract. The upcoming safety standard ISO/WD 26262 that has been derived from the more general IEC 61508 and adapted for the automotive industry, introduces the concept of a safety case, a scheme that has already been successfully applied in other sectors of industry such as nuclear, defense, aerospace, and railway. A safety case communicates a clear, comprehensive and defensible argument that a system is acceptably safe in its operating context. Although, the standard prescribes that there should be a safety argument, it does not establish detailed guidelines on how such an argument should be organized and implemented, or which artifacts should be provided.

In this paper, we introduce a methodology and a tool chain for establishing a safety argument, plus the evidence to prove the argument, as a concrete reference realization of the ISO/WD 26262 for automotive systems. We use the Goal-Structuring-Notation to decompose and refine safety claims of an emergency braking system (EBS) for trucks into sub-claims until they can be proven by evidence. The evidence comes from tracing the safety requirements of the system into their respective development artifacts in which they are realized.

1 Introduction

Safety critical systems have to fulfill safety requirements in addition to functional requirements. Safety requirements describe the characteristics that a system must have in order to be safe [12]. This involves the identification of all hazards that can take place, and that may harm people or the environment. Safety-related issues are often captured in standards describing products and processes to be considered throughout the life-cycle of a safety critical system. The upcoming safety standard ISO/WD 26262 [2] is an implementation of the more general IEC 61508 standard that addresses safety issues in the automotive industry. The objective of the automotive standard is to take the specific constraints of automotive embedded systems and their development processes

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 1–13, 2007.

into account. Domain-dependent challenges are, for instance, the task distribution of OEMs and suppliers, the degree of iterative development, and the high importance of the application of the embedded system to the target vehicle. The short development cycles and large number of produced units are further specifics which disallow the straight-forward application of safety standards from other domains of transportation. The current working draft is divided into a number of volumes considering e.g. determination of safety integrity levels, and requirements to systems development, software development, and supporting processes. Besides many other requirements the standard prescribes that a safety case should be created for every system that has safety-related features. It states that part of the system documentation should provide evidence for the fulfillment of safety requirements, thus guaranteeing functional safety. However, the standard does not provide any details about which artifacts should be produced in order to prove functional safety, nor does it say how such a proof may be devised.

In this paper, we develop a generic safety case that may act as a reference realization for the automotive industry. In section 2, we describe how a safety case may be constructed based on the Goal-Structuring-Notation (GSN). Section 3 shows that constructing a safety argument is, to a large extent, a traceability effort and dealing with the construction of trace tables. Section 4 presents the case study, a safety critical system, for which we have devised part of a safety argument. Here, we concentrate on the traceability part. Section 5 summarizes and concludes the paper.

2 The Safety Case

Part of the certification process in the automotive domain is the assessment of a system through an inspection agency. To convince inspectors that a system is safe, a safety case should be created. The safety case communicates a clear, comprehensive and defensible argument that a system is acceptably safe in its operating context [7]. The argument should make clear that it is reasonable to assume the system can be operated safely. It is typically based on engineering judgment rather than strict formal logic [12], and it provides evidence that the risks (hazards) associated with the operation of the system have been considered carefully, and that steps have been taken to deal with the hazards appropriately.

The safety argument (SA) must identify all matters significant to the safety of the system and demonstrate how these issues have been addressed. A convenient way to define a safety argument is through the Goal-Structuring-Notation devised by Kelly [7] which is based on earlier work by Toulmin on the construction of arguments [13]. An argument consists of claims whose truth should be proven. The facts used to prove the claims are referred to as data, and the justification for why data prove a claim is described by warrants. If it is possible to dispute a warrant, backing can be used to show why the warrant is valid. The structure of this argument is depicted in Fig. 1 as far as it is relevant for

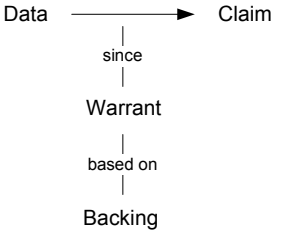


Fig. 1. Structure of Toulmin's Argument [13]

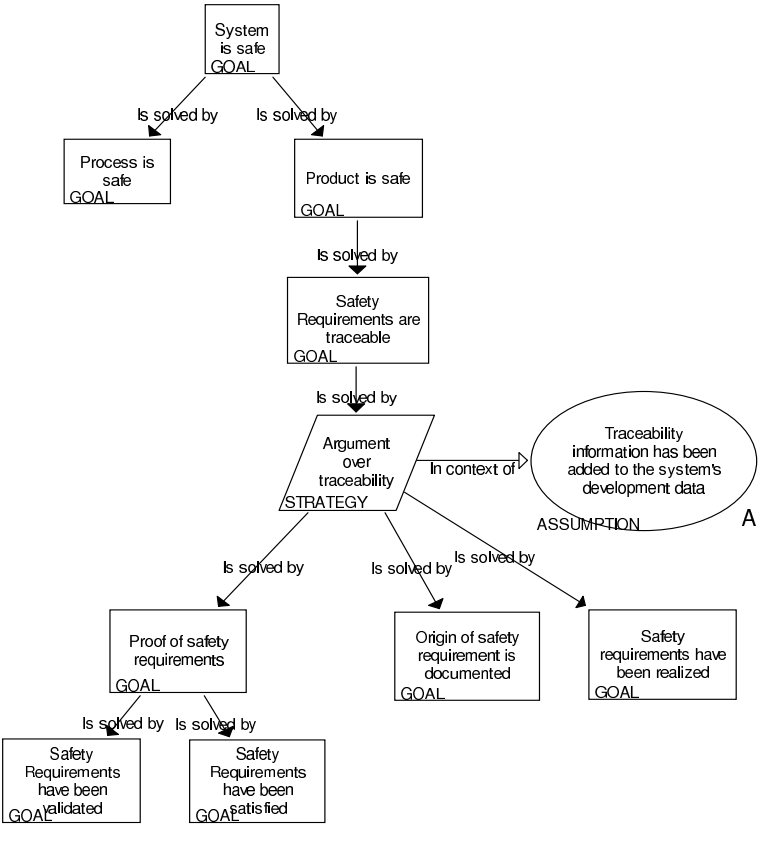


Fig. 2. Example GSN tree, decomposition of the goal "the product is safe"

the safety case. Further concepts being introduced by Toulmin like qualifiers or rebuttals do not contribute to the construction of a safety case. Qualifiers are ordinary language terms like "few", "rarely", or "some" which are used to express the likelihood of a warrant or backing. For safety analysis, formal methods are available to capture this aspect of a safety analysis. Rebuttals deal with the

potential counter-arguments arising in an ongoing discussion. However, a safety case is the result of an intense discussion during which all counter-arguments must have been evaluated and resolved such that a safety case can be accepted. So, rebuttals are not relevant for a safety case.

The main elements of the GSN are goals and solutions. Goals correspond to Toulmin's claims whereas solutions relate to Toulmin's data, also termed evidence. For constructing a safety case, we have to determine which evidence is required for a particular safety argument, and why the evidence supports the claim. According to the GSN, the safety case starts with a top-level claim, or a goal, such as "the system is safe" or "safety requirements have been realized." The top-level claim is then decomposed into sub-ordinate claims down to a level that a sub-claim can be proven by evidence. The concepts of the GSN are displayed in the example in Fig. 2. Claims and sub-claims, or goals, are represented as rectangular boxes and evidence, or solutions, as circles. A strategy-node, represented as rhomboid, contains the explanation why a goal has been decomposed. The argument can be constructed by going through the following steps [3,7]:

1. Identification of the goals to be supported.
2. Definition of the basis on which the goals are stated.
3. Identification of the strategy to support the goals.
4. Definition of the basis on which the strategies are stated.
5. Elaboration of the strategy including the identification of new goals and starting from step 1, or moving to step 6.
6. Identification of a basic solution that can be proven.

It is difficult to propose a standard safety case structure that may be valid for most systems. However, some of the argumentation will be the same for many systems, such as "all safety requirements have been realized," or the like. Such argumentation structures, or so-called safety case patterns [6,15], may be reused in several safety cases for different systems. By using such patterns, safety cases can be devised much faster. Similarly, safety case anti-pattern can be used to express weak and flawed safety arguments [7,14].

A particular GSN decomposition proposed by the EU project EASIS [3] organizes the argumentation into a product branch and a process branch, claiming that "a system is safe" if "the process is safe" and "the product is safe." The safety of the process can be assured through application of certified development standards, such as the IEC 61508 [4] or the V-model [11]. Here, questions should be asked about how the product is developed, such as "did we perform hazard analysis?", "do we have a hazard checklist?", "did we perform a preliminary hazard identification?", "did we implement the results of the preliminary hazard identification?", and so on [3].

On the product side, we can decompose the claim "the product is safe" into the sub-goal "the safety requirements are traceable" which turns the satisfaction of our safety case into a traceability problem. We can argue, that if all safety related aspects of our system can be traced to their origin and to their realization, the system is safe, given the process is safe (proof of the process branch). Traceability

is a prerequisite for assessment and validation of the safety goals, which we refer to as "proof of safety requirements." We can then decompose the traceability goal further into "proof of safety requirements", "origin of safety requirements documented", and "safety requirements realized." This extended organization of the safety case is depicted in Fig. 2 and, through its general nature, it can be used as a pattern for all systems.

3 Traceability of Safety Requirements

In the previous section, we argued that part of the proof of a safety case can be achieved through tracing all safety requirements to the respective development documents. This is fully in line with the ISO/WD 26262 [2] since it demands that "the origin, realization and proof for a requirement are clearly described in the documentation" of a system. A requirement is a condition or an ability that the system should fulfill. The origin of a requirement is a rationale why this requirement has been elicited for the system. The realization demonstrates how/where the requirement is implemented in the final system. A proof for a requirement means that it should be demonstrated that the requirement has an origin and that it is implemented, in other words, that the requirement is traceable across all development documents in both directions, forwards and backwards. The documents comprise the hazards possible, the safety goals, the safety requirements, design elements, and implementation elements, plus associated review documents.

As shown in Fig. 2, the "product is safe"-branch is decomposed into a traceability sub-goal that is split into various traceability claims, i.e., "origin of safety requirements documented", "safety requirements realized", and "proof of safety requirements." This last goal is decomposed into two sub-goals, "safety requirements validated" and "safety requirements satisfied" which can be traced to the respective documents that deal with those issues.

The origin of a safety requirement can be demonstrated by backward traceability. Safety requirements are derived from hazards and safety goals. Every safety requirement should be linked to at least one safety goal, expressed through $\forall srSR \longrightarrow \exists sgSG$, and every safety goal should be linked to a hazard, expressed through $\forall sgSG \longrightarrow \exists hH$. But also forward traceability is important, so that for every hazard, there is a safety goal ($\forall hH \longrightarrow \exists sgSG$), and for every safety goal, there should be an associated safety requirement ($\forall sgSG \longrightarrow \exists srSR$). Consequently, we can extend our safety case as depicted in Fig. 3.

For the lowest-level goals, we can then come up with solutions in the form of trace tables. These are now product-specific. Once all trace relations have been established in a development project we can claim that the system is acceptably safe with respect to those safety goals, e.g. with respect to "origin of safety requirements is documented." We have to do this for all safety goals defined, and we demonstrate how this may be done for a specific case using specific tools in the next section.

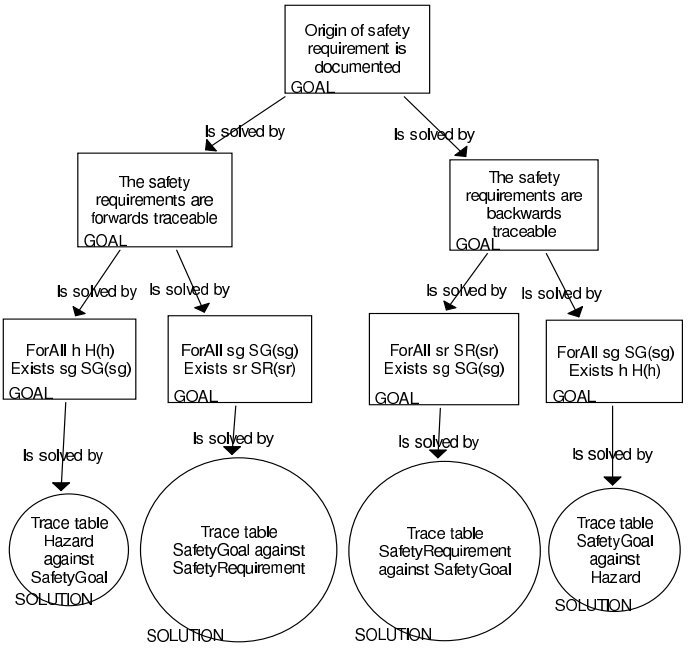


Fig. 3. Further decomposition of the goal "origin of safety requirement is documented"

4 Case Study

We have devised a partial safety case for an emergency brake system (EBS). This innovative assistance functionality becomes part of modern vehicles and lorries. DaimlerChrysler, for instance, markets that type of application as Active Brake Assist. An emergency brake system warns the driver of a likely crash, and, if the driver does not react upon the warning, initiates and emergency braking. It is a distributed system incorporating the braking system, a distance sensor, the hifi-system, a control system, as well as the driver's interface. The individual subsystems are usually interconnected through a vehicle's CAN bus.

In order to devise the traceability part of the safety case, first, we have to take a look at the development processes and tools deployed. The safety requirements (SR) for this system are coming from a preliminary hazard analysis (PHA), or from a hazards and operability analysis (HAZOP) [10]. Once all the potential hazards have been identified, they are associated with safety goals (SG). Safety goals are comparable to top level functional requirements. The requirements are decomposed into sub-system requirements, and eventually, into component requirements. For the execution of the safety analysis the system boundaries have to be determined. This design decision is typically a compromise between controllability and complexity. An emergency brake system is a safety application

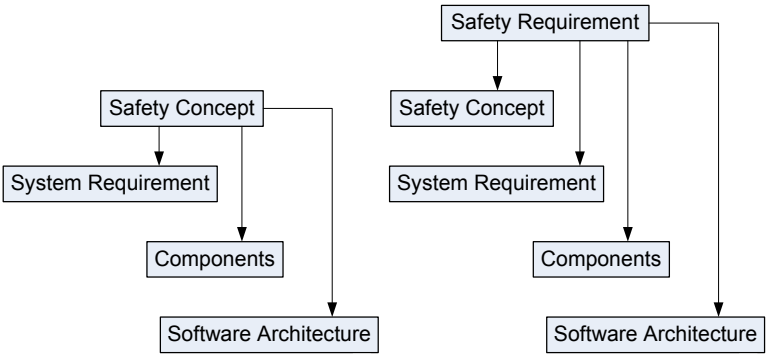


Fig. 4. Classes of development artifacts and their (forward) traceability relations that we implemented for the EBS case study

and an add-on to an already available braking system. The system boundaries of that embedded system therefore are mostly established by electronic signals. Potential hazards due to brake wear for instances will have to be signaled by the brake and these signals must be taken into account by the safety analysis.

All requirements are managed within Telelogic's DOORS, a widely used requirements management tool (http://www.telelogic.com). They are associated with different levels such as safety requirement, safety concept, system requirement, component specification, etc, and the tool maintains also traces between those levels. These traces are shown in Fig. 4.

Apart from DOORS for the requirements management, various other tools should be used throughout the other development phases in order to comply to standard's requirements. A typical tool chain could consist of:

- Matlab's Simulink and Stateflow (http://www.matlab.com) are used for system and component design,
- DSpace's TargetLink (http://www.dspace.com) is used for the implementation and automatic code generation out of the Matlab models.
- Tessy (by Hitex) is used for automated unit testing,
- the Classification Tree Editor (CTE by Hitex) is used to support input domain-based testing,
- Time Partition Testing is employed for generating test cases with continuous input data streams [8,9],
- DSpace's MTest generates test cases automatically based on the Simulink and TargetLink models (http://www.dspace.com),
- QA-C/Misra can be used to analyze the resulting C-code (http://www.qasystems.de),
- PolySpace is a tool that can detect run-time errors during compile time (http://www.polyspace.de), and
- Mercury's Quality Center is a global suite of tools for ensuring software quality (http://www.mercury.com).

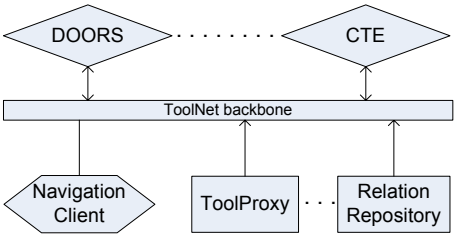


Fig. 5. ToolNet Architecture

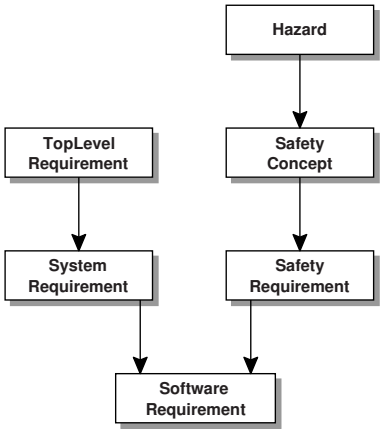


Fig. 6. Information model for the requirements engineering phase

4.1 ToolNet

The tools in the previous list are dealing with different types of work products, and the trace tables that we have to devise for the safety case have to refer to the artifacts stored in these various tools. In other words, in order to realize a full tracing between the different work products, we have to gain access to the tools' various data representations. This is done through another tool, called ToolNet [1] which enables us to create traceability links between various development artifacts independent from the type of tool through which they have been created. ToolNet is based on a bus-architecture, the so-called information backbone that connects each tool via an adapter [1] (Fig. 5). Every single development object recognized by ToolNet is assigned a unique ID (object reference) which is based on its data source (a tool or part of a tool). The development objects must be defined unambiguously, according to a product-specific information model [5]. It describes the available and traceable development objects such as hazard, safety concept, safety requirement, etc. An example information model for our project is displayed in Fig. 6. The development object models and the ToolNet structure permit the tools to interact with each other through services implemented in

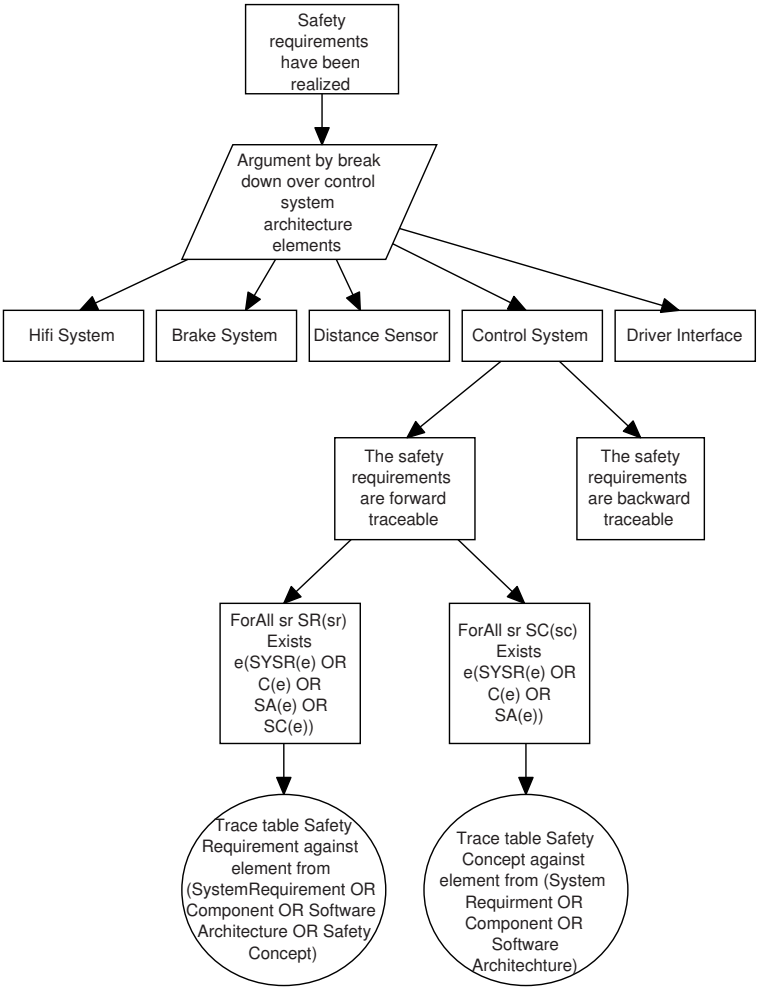


Fig. 7. Product-specific safety case for the emergency braking control system

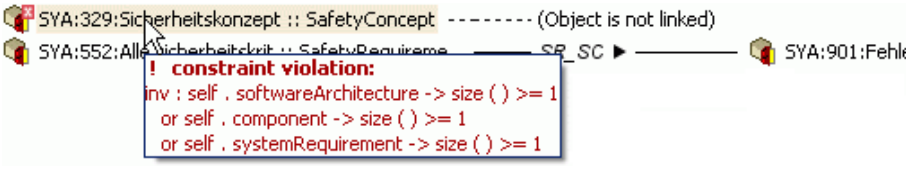


Fig. 8. Evaluation of an OCL constraint in ToolNet

their respective adapters. A user can select a specific development object from one tool, e.g., a requirement within DOORS, and associate it with another

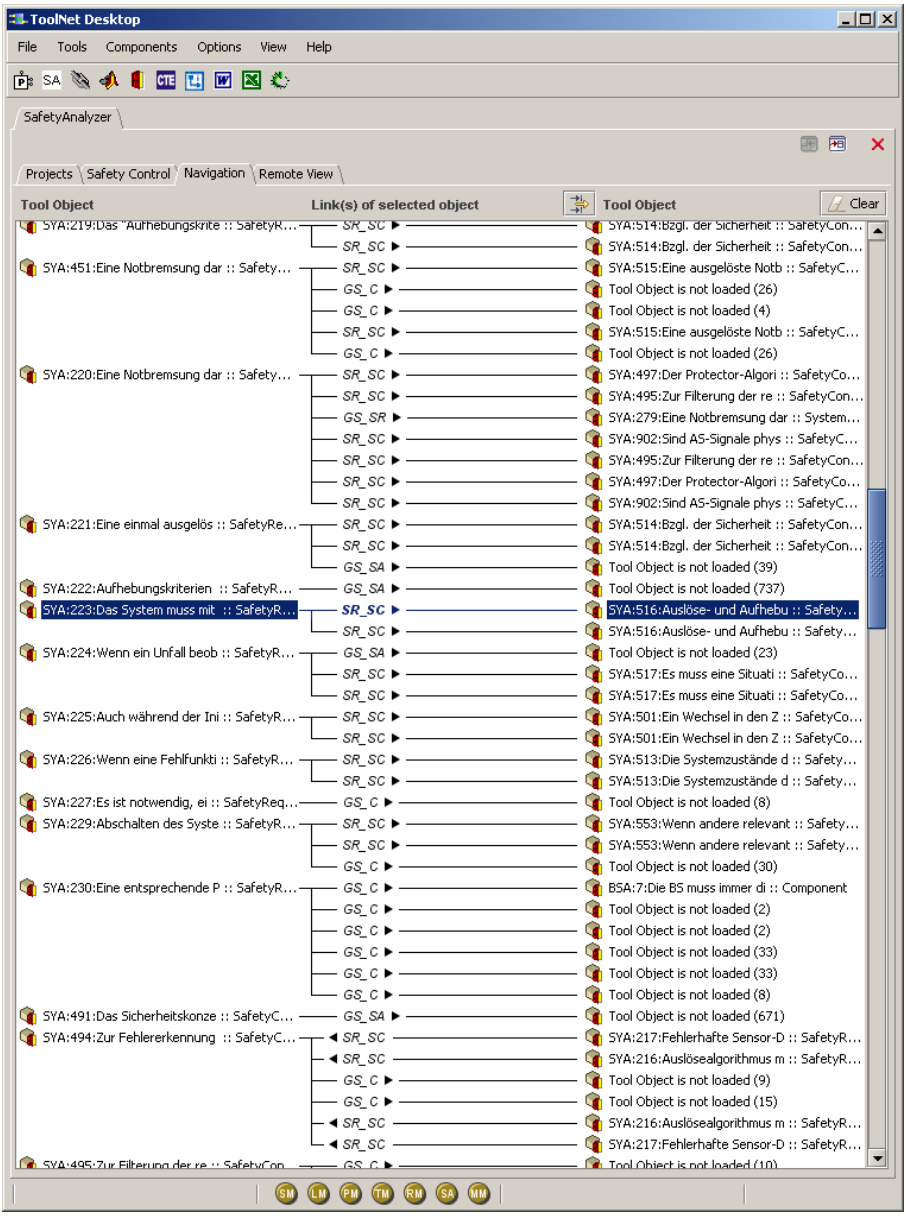


Fig. 9. ToolNet trace table highlighting a trace between a safety requirement and safety concept

development object, e.g., from Matlab/Simulink, and eventually link that to an implementation and a number of test cases. In that way, engineers can build up a network of associations between the many different development objects, and

thus, construct a trace table for an entire project. The tracing data is stored within ToolNet.

4.2 Traceability of the EBS-System

When we created the sample safety case for the emergency braking system, we could, to a large extent, reuse the structure of our generic safety case described in Sect. 2. The only difference is that we had to apply our claim "safety requirements are traceable" to all subsystems of the EBS, the hifi-system, the breaking system, the distance sensor and the control system. We could, therefore, extend the safety case displayed in Fig. 2 with a corresponding structure for each of the four sub-systems, leading to four separate sub-goals of the super-ordinate goal "the system requirements are traceable." This extended product-specific safety case is displayed in Fig. 7. We show the extended safety case for the control system only. Figure 4 shows the forward traceability links between the various types of artifacts that we implemented for the EBS system. Each artifact within the various tools had to be assigned to one of these classes. We implemented that through adding a type attribute to the artifacts. For future projects, we propose to create an individual DOORS module for each of the artifact types in order to facilitate their management. After we had established all traceability links, the final step was the evaluation of our coverage criterion, that is, can all development artifacts be linked to artifacts in the respective other classes of artifacts, and can they be linked to an implementation? In other words, we had to verify the relations stated in our (forward) traceability claims (Figure 7).

4.3 Verification of the Traceability Links

The aim of a safety argument, following the goal structuring notation, is to associate goals with solutions. A goal is a claim corresponding to a required safety property of a system, and a solution is evidence supporting this claim. By associating every claim with a solution, we demonstrate that a system fulfills its safety requirements. Initially, we will have no associations stored in the trace tables, but, eventually, throughout the system development, the trace tables will be filled. ToolNet can be used to depict the information contained in the trace tables. An example is shown in Fig. 9.

Assessing to which extent the claims are proven by evidence in a large system such as the EBS control system, can be daunting. Identifying missing links manually is very tedious and time consuming, and the task must be repeated every time the system requirements are amended. This is why we are currently devising an automatic solution generator that can compile the various trace tables based on the development data found in ToolNet, and then make a traceability analysis. It works based on evaluating OCL constraints expressing the traceability relations. The outcome of such an analysis is twofold. First, we can highlight missing links in the desktop view of ToolNet for the developers (displayed in Fig. 8), and, second, we can generate reports about the safety status of an entire system, i.e., for management or, eventually, the certification authority.

5 Summary, Conclusions and Future Work

The upcoming safety standard ISO/WD 26262 for the automotive industry, introduces the concept of a safety case, which communicates a clear, comprehensive and defensible argument that a system is acceptably safe. In the future, it will be required by safety inspectors in order to assess how and to which extent all safety-related issues have been addressed and treated by the vendor of a safety-critical system in the automotive domain.

In this paper, we demonstrated how a safety case can be constructed based on the goal structuring notation that proposes to create a defensible argument out of goals that are decomposed recursively into subgoals until a subgoal can be proven by evidence. The evidence is provided by documents addressing the safety issues. The standard prescribes that part of the safety case should demonstrate that the origin, realization and proof for every requirement is clearly documented. In other words, all requirements should be traceable to their respective implementations (forward and backward).

We developed a generic safety case that could be applied to emergency braking systems like the Active Brake Assist for lorries of the brand Mercedes-Benz. This safety case may act as a template, or a reference realization for other systems in the automotive domain. The greatest challenge was to incorporate the traceability features of our safety case into the existing development process that employs a number of different tools. We solved that through adding trace information to ToolNet, a framework that integrates many different software engineering tools. Within ToolNet, we can now navigate along the traceability paths and assess which safety requirements have been treated sufficiently.

Current work is now focused on the development of an automatic solution generator that will compile the trace tables required for the safety argument automatically. In the future, this generator can be used to provide the current safety status of a project on the punch of a button, i.e., for project management, display a colored safety argument, with green and red indicating the safety status of system parts, and, eventually, compile the safety reports for the inspection agency. Further experience will also have to show whether the purely positivistic approach presented here must be extended to deal with context information and the explication of arguments. In particular the contribution of traceability information to context and arguments must be studied.

References

1. Altheide, F., Dörfel, S., Dörr, H., Kanzleiter, J.: An Architecture for a Sustainable Tool Integration. In: Workshop on Tool Integration in System Development, Helsinki, Finland, September 2003, pp. 29–32 (2003)
2. Automotive Standards Committee of the German Institute for Standardization: ISO/WD 26262: Road Vehicles – Functional Safety. Preparatory Working Draft, Technical Report (October 2005)
3. Bridal, O., et al.: Deliverable D3.1 Part 1 Appendix E: Safety Case, Version 1.1. Technical Report, EASIS Consortium (February 2006), http://www.easis-online.org

4. Intl. Electrotechnical Commission. IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Technical Report (April 1999)
5. John, G., Hoffmann, M., Weber, M.: EADS-Methodenrichtlinie zur Traceability zwischen Anforderungen und Entwurfsobjekten. Technical Report RM-008, DaimlerChrysler AG (November 2000)
6. Kelly, T.P., McDermid, J.: Safety Case Construction and Reuse using Patterns. In: Proceedings of 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP'97), September 1997, Springer, Heidelberg (1997)
7. Kelly, T.P.: Arguing Safety: A Systematic Approach to Managing Safety Cases. PhD Thesis, University of York, UK (September 1998)
8. Lehmann, E.: Time Partition Testing: A Method for Testing Dynamic Functional Behaviour. In: Proceedings of TEST2000, May 2000, London, Great Britain (2000)
9. Lehmann, E.: Time Partition Testing. PhD Thesis, Technical University of Berlin (February 2004)
10. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Boston, MA (1995)
11. Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung. V-Modell©XT (2004), http://www.kbst.bund.de
12. Storey, N.: Safety Critical Computer Systems. Addison-Wesley, Reading (1996)
13. Toulmin, S.E.: The Uses of Argument. Cambridge University Press, Cambridge (1958)
14. Weaver, R.A.: The Safety of Software – Constructing and Assuring Arguments. DPhil Thesis, Department of Computer Science, University of York, UK (2003)
15. Weaver, R., Despotou, G., Kelly, T., McDermid, J.: Combining Software Evidence: Arguments and Assurance. In: Proceedings of the 2005 workshop on Realising evidence-based software engineering, St. Louis, Missouri, pp. 1–7 (2005)

Goal-Based Safety Cases for Medical Devices: Opportunities and Challenges

Mark-Alexander Sujan[1], Floor Koornneef[2], and Udo Voges[3]

[1] Health Sciences Research Institute, University of Warwick, Coventry CV4 7AL, UK
m-a.sujan@warwick.ac.uk
[2] Delft University of Technology, TPM - Safety Science Group, P.O. Box 5015, 2600 GA Delft, The Netherlands
f.koornneef@tudelft.nl
[3] Forschungszentrum Karlsruhe GmbH, Institut für Angewandte Informatik, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany
udo.voges@iai.fzk.de

Abstract. In virtually all safety-critical industries the operators of systems have to demonstrate a systematic and thorough consideration of safety. This is increasingly being done by demonstrating that certain goals have been achieved, rather than by simply following prescriptive standards. Such goal-based safety cases could be a valuable tool for reasoning about safety in healthcare organisations, such as hospitals. System-wide safety cases are very complex, and a reasonable approach is to break down the safety argument into sub-system safety cases. In this paper we outline the development of a goal-based top-level argument for demonstrating the safety of a particular class of medical devices (medical beds). We review relevant standards both from healthcare and from other industries, and illustrate how these can inform the development of an appropriate safety argument. Finally, we discuss opportunities and challenges for the development and use of goal-based safety cases in healthcare.

1 Introduction

In most safety-related industries the operators of systems have to demonstrate a systematic and thorough consideration of safety. In healthcare there is currently a differentiation between manufacturers of medical devices on the one hand and healthcare providers as users or consumers of such devices on the other hand. The current regulatory practice implies that the device manufacturers are responsible for determining acceptable levels of risk and for ensuring that the device is adequately safe for use in a specific context. However, the manufacturer usually has limited control over how devices are used in the operational context, and whether critical assumptions about aspects, such as training and maintenance are fulfilled. In addition, the healthcare service provider often has to integrate a number of different devices within their environment. The safety of the resulting system can only be assured if sufficient information from the manufacturer is provided.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 14–27, 2007.

In this paper we show how the approach of a goal-based safety case can be used to analyse the safety of a medical device throughout its lifecycle, and to document the respective evidence used. Such an approach aims to overcome the limitations of current practice that results from the two disjoint regulatory contexts. It is also a first step towards investigating the feasibility of more complex system-wide safety cases.

Section 2 provides an argument for goal-based safety cases. In section 3, the regulatory context in the medical device area is described (using the UK as an example). The related standards and their dependence are presented in section 4. To explain the problem more closely, medical beds are used as an example for a medical device, and the outline of a safety case for this example is developed in section 5. Finally, section 6 concludes with a discussion of opportunities and challenges for the development and use of goal-based safety cases in healthcare.

2 Goal-Based Safety Cases

Argumentation is an important part of the development of safety critical systems. It provides information about why a system can be assumed to be sufficiently safe, and it may convey a measure of confidence. In many safety-critical industries such information is documented in a safety case. The purpose of a safety case can be defined as communicating a "clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context" [1]. This definition reflects a goal-based approach, where the justification is constructed via a set of claims about the system, its behaviour, the process through which the system was produced, and about the safety case itself (i.e. the quality and the trustworthiness of the argument and the evidence produced) [2]. To support these claims specific evidence is produced. An essential component of goal-based safety cases is the argument that explains how evidence supports a particular claim. The argument makes explicit in the forms of rules and inferences the relationship between claims and evidence (see [3] for an extensive discussion of argument structure).

The use of goal-based arguments is now increasingly being reflected in standards, such as the UK Defence Standard 00-56 in its latest revision [4, 5]. Goal-based standards tell operators of systems what they need to achieve, rather than what kind of specific safety requirements they have to comply with. As technologies are evolving increasingly rapid, such an approach offers greater flexibility with respect to the use of novel beneficial technologies for which no corresponding assessment method has been defined in the standard, or practices that supersede outdated and costly development and assessment techniques [5].

Many standards also mandate incremental or evolutionary approaches to safety case development [6], such as the above mentioned UK Def Stan 00-56. Such an incremental approach can include multiple issues of, for example, Preliminary Safety Case, Interim Safety Case, and Operational Safety Case. At the Preliminary Safety Case stage the safety argument defines and describes the principal safety objectives, the general approach to arguing safety, and the types of evidence anticipated [1]. As

the design progresses and more detailed information becomes available, the arguments are subsequently extended and refined.

Narrative accounts of safety justifications often make it difficult for the reader or assessor to follow the logical argument that relates evidence to the claim it is intended to support. In addition, multiple cross-references make such documents generally hard to read and difficult to communicate to stakeholders of different backgrounds. Graphical argument notations, such as Goal Structuring Notation (GSN) [7] or ASCAD [8] explicitly represent the key elements of any safety argument, and their relationships. Tools have been developed that facilitate the construction of such graphical representations (SAM, ASCE) [9, 10]. With these tools the construction and the communication of safety cases is greatly facilitated [11].

3 The Regulatory Context

In many European healthcare systems there is a differentiation between manufacturers of medical devices on the one hand, and healthcare providers as users or consumers of such devices on the other hand. In general, manufacturers have to provide evidence that their devices are tolerably safe for a particular use in a specific environment. Healthcare providers, on the other hand, are audited to ensure that the care they provide meets national standards. A part of this is the requirement to utilise only previously certified medical devices. In this section we illustrate the certification process of medical devices and the audit of healthcare providers using the UK environment as an example [12].

The UK Medical Devices Regulations 2002 (MDR 2002) [13] implement a number of European directives relevant to the certification of medical devices. The definition of what constitutes a medical device is broad and comprises devices as diverse as radiation therapy machines, syringes and wheelchairs. The Medicines and Healthcare Products Regulatory Agency (MHRA) acts as the *Competent Authority* overseeing the certification of medical devices. *Notified Bodies* of experts provide evaluation of high and medium risk medical devices undergoing certification to the Competent Authority. The Medical Devices Directive [14] specifies essential requirements that have to be met by any device to be marketed in the EU. It provides classification rules based on the risk that medical devices pose, as well as conformity routes that specify different ways of manufacturer compliance with the essential requirements based on the class of the medical device under consideration. For most medical devices compliance with the requirements is demonstrated not through an explicit argument, but rather through either a self-certification process (lowest risk class) or through the compilation of specified evidence, including general product description, results of the risk analysis, and testing and inspection reports.

Apart from issuing instructions for use, the manufacturer of common medical devices has little influence on the way the devices are actually used in practice. More importantly, the manufacturer does not have detailed information about the specific environment and the processes within which the device will be operated within a

particular healthcare provider's setting. In complex systems this is a serious cause for concern, as in this way the possible interactions between system components and interactions with the environment as well as the system's particular history will not have been accounted for. It is reasonable, therefore, to expect healthcare providers to demonstrate that the services they are providing are acceptably safe. Such a demonstration should make use of data supplied by the manufacturers.

At present, healthcare providers are audited through a process that involves a number of diverse actors and agencies. The annual review of, for example, NHS Trusts is undertaken by the Healthcare Commission. The aim of this review is to establish whether Trusts comply with standards set out by the Department of Health [15]. These standards include aspects of safety, but are generally broader focussing also on issues such as financial health. During the annual review it is assessed whether Trusts have basic mechanisms in place, such as risk management and incident reporting, and whether there is evidence of continuous progress, e.g. learning from incidents. The data is collected throughout the year and includes national data about Trust activities, information from local organisations, documentation provided by the Trust, meetings with patient groups, as well as data from brief site visits conducted by assessors. The focus is on collecting indicators of (in the case of safety) safe practices, and accordingly the recommendations are targeted at specific issues, such as the fact that patients may not receive appropriate levels of nutrition, or that lessons learned from incidents are not shared among directorates.

In conclusion, therefore, within the UK regulatory context, both manufacturers of medical devices and healthcare service providers are regulated and are required to provide some kind of evidence that their devices and the services they provide are acceptably safe. However, in most cases there is no formal argument, and the two regulatory contexts (certification and audit) show little integration. This implies that assumptions and dependencies may not be documented properly, that interactions and unintended consequences of changes may go unnoticed, and that there are no formal notions of issues such as confidence in the evidence or diverse evidence to mitigate possible uncertainty (see e.g. [16] for an attempt of a corresponding formalism).

4 Relevant Standards

Safety of medical devices is regulated in about a thousand standards. In Europe, over two hundred of these are harmonised and provide a technical interpretation of the essential requirements of the MDD [14]. The main standard for electrical medical systems is the IEC 60601 series, which is now well underway in its 3rd edition revision process that started in 1995. The IEC 60601-1 series consists of Part 1: general requirements for basic safety and essential performance [17], and a number of collateral standards IEC 60601-1-XY on EMC, radiation, usability, alarm systems, environment, and physiologic closed-loop controllers. In addition, a particular standards series IEC 60601-2-YZ addresses specific requirements for particular systems, e.g. anaesthetic systems (60601-2-13), ECG monitoring equipment

(60601-2-27), ultrasonic equipment (60601-2-37) and screening thermographs (60601-2-56). Since 2005, the 3rd edition includes an update of all "tried and true" requirements of the 2nd edition and introduction of solutions now possible due to the availability of new technology. It also formalises the introduction of "Risk Management" by integration of standard ISO 14971 [18], see below, in order to make the standard less dependent on rapid growth in technology, and because there is more than "tried and true" requirements listed in the standard. Thus, it can be stated that medical electrical equipment shall remain single fault safe or the risk shall remain acceptable.

ISO 14971, entitled Application of risk management to medical devices, is now in its 2nd edition. This industry standard requires that the manufacturer shall establish, document and maintain throughout the life-cycle a process for identifying hazards associated with a medical device, estimating and evaluating the associated risks, controlling these risks, and monitoring the effectiveness of the controls. The manufacturer needs to demonstrate that the residual risk of using the medical system is acceptable. This is assumed when compliance to the requirements specified in IEC 60601-1 is demonstrated, but the bottom line is that the acceptability of a risk is determined by the manufacturer's policy for risk acceptability. Compliance to the performance of the risk management process is done by inspection of the risk management file (RMF). The RMF is the set of records produced by the risk management process, and remains the manufacturer's property that is not available to users.

Programmable Electrical Medical Systems (PEMS) are addressed in IEC 60601-1 clause 14 and regarding software elaborated in IEC 62304: Medical device software - Software life cycle processes [19]. Note that "it is recognized that the manufacturer might not be able to follow all the processes identified in clause 14 for each constituent component of the PEMS, such as off-the-shelf (OTS) software, subsystems of non-medical origin, and legacy devices. In this case, the manufacturer should take special account of the need for additional risk control measures."

Safety-critical human factors requirements are addressed in the standard IEC 62366 - Medical devices - Application of usability engineering to medical devices [20]. It states that "the manufacturer shall establish, document and maintain a usability engineering process to provide safety for the patient, operator and others related to usability of the human interface. The process shall address operator interactions with the medical device according to the accompanying documents, including, but not limited to transport, storage, installation, operation, maintenance, and disposal."

Other mainly technical standards exist for laboratory equipment, image processing systems and information networks, but these are not further elaborated in this paper. The standards of safety of medical systems have gone through a major revision process since 1995. This 3rd edition process will end with the implementation of the last collateral standard by about 2009.

The whole set of standards on safety of medical systems puts the manufacturer in the position of the decision maker of risk acceptance. The underlying assumption is

that a) the medical system will be used by "laymen", and b) the manufacturer defines normal use. All risk data about hazards, associated risks and acceptance criteria regarding a particular medical system has been elicited with adequate resources using inside information, and is recorded in the RMF. However, the professional user in a health care institution is left empty handed when they combine two medical systems in one configuration. [21, 22]. It is here that safety cases might help relevant parties to improve the understanding of risk assessment and control of operational risks related to the use of medical systems.

5 Development of the Top Level Argument for Medical Beds

5.1 Medical Beds

A medical bed is possibly the most stubborn medical device with respect to risk identification, control and management. Based on risk minimisation, an optimal bed may in many ways be bad for the patient as well as for the person providing care, because the bed will be higher than preferable for the patient, and lower than necessary for appropriate use of lifting and handling techniques. The patient is at risk due to gravitational forces (height) and opportunities for entanglement. Falling out of bed or getting strangled between bedrails are real harm scenarios. Occupational health and safety is relevant to nursing staff in particular: staff is at risk because of incorrect handling of the patient, wrong height of the bed, absence or wrong use of lifting aids, etc., potentially leading to serious back injury and disability to work. Hazards associated with medical beds include e.g. electricity, mechanical, electromagnetic interference and software failures in the bed motion control system. Mechanical hazards include e.g. entrapment, moving beds bumping into walls or other objects, bed instability, a collapsing component, and falls.

The medical bed (see Fig. 1) is defined in the particular standard IEC 60601-2-52 as a "device for which the intended use is sleeping or resting that contains a mattress

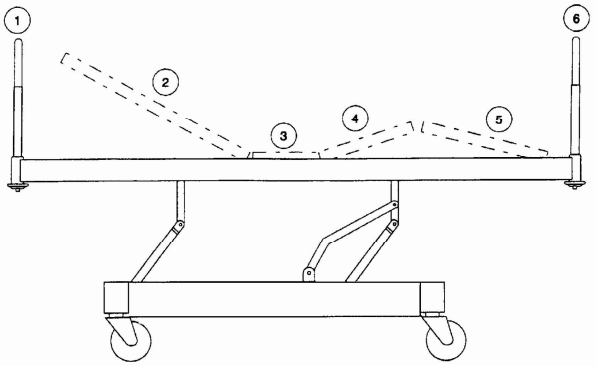


Fig. 1. Schema of a medical bed (from IEC 60601-2-52)

support platform. The device can assist in diagnosis, monitoring, prevention, treatment, alleviation of disease or compensation for an injury or handicap. A bed lift or a detachable mattress support platform in combination with a compatible non-medical bed as specified by the manufacturer is also considered a medical bed. Excluded are devices for which the intended use is examination or transportation under medical supervision (e.g. stretcher, examination table)." [23] Medical beds are meant for patients being defined as "person undergoing a medical, surgical or dental procedure, or disabled person". Maintenance of medical beds includes cleaning before reuse for another patient.

The standard identifies five distinct application environments, see IEC 60601-2-52:

1. Intensive/critical care provided in a hospital where 24 hours/day medical supervision and constant monitoring is required and provision of life support system/equipment used in medical procedures is essential to maintain or improve the vital functions of the patient.
2. Acute care provided in a hospital or other medical facility and medical electrical equipment used in medical procedures is often provided to help maintain or improve the condition of the patient.
3. Long term care in a medical area where medical supervision is required and monitoring is provided if necessary and medical electrical equipment used in medical procedures may be provided to help maintain or improve the condition of the patient.
4. Care provided in a domestic area and medical electrical equipment is used to alleviate or compensate for an injury, or disability or disease.
5. Outpatient (ambulatory) care which is provided in a hospital or other medical facility, under medical supervision and medical electrical equipment is provided for the need of persons with illness, injury or handicap for treatment, diagnosis or monitoring.

The use context of medical beds is important also because opportunities for managing operational risks differ.

5.2 General Top-Level Structure

A safety case essentially attempts to demonstrate that:

- The system under consideration is acceptably safe to enter service (in the UK this usually implies that safety risks are broadly acceptable or have been reduced as low as reasonably practicable).
- Arrangements are in place to ensure that the system will remain acceptably safe throughout its lifecycle.
- The structure and content of the safety case, and the process by which the safety case is produced and maintained are adequately trustworthy to produce a sound and convincing argument.

A common approach to demonstrate that the system under consideration is acceptably safe and continues to be so, is to argue that adequate safety requirements

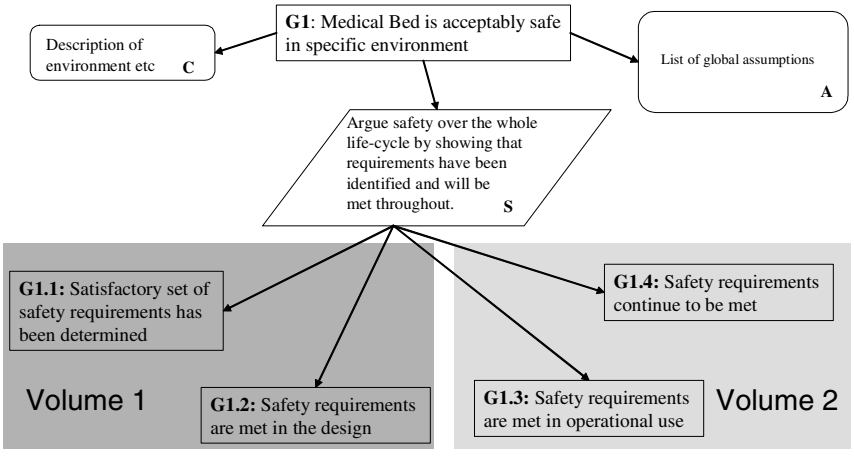


Fig. 2. Structure of the top-level argument and distribution of goals between manufacturer (Volume 1) and service provider (Volume 2)

have been established, that the safety requirements are met in the design, and that they continue to be met throughout all stages of the lifecycle of the system (see, for example, the objectives specified in the UK Defence Standard 00-56 [4]).

A well documented and frequently discussed example of a safety case formulated in GSN is the Eurocontrol RVSM Pre-Implementation Safety Case [24]. Here, the argument relies on four principle claims:

- Safety requirements have been determined and are complete and correct.
- The safety requirements are realised in the concept.
- The safety requirements are realised in the respective national implementations.
- The switch-over period (i.e. the transition from conventional vertical separation to the reduced vertical separation) is adequately safe.

One of the high-level safety requirements relates to continued safety throughout operational life:

- The performance and reliability of the system shall not deteriorate in service.

The arguments are then organized in such a way that for each it is demonstrated that sufficient direct evidence is available, and that this evidence is sufficiently trustworthy.

In principle, a similar general approach can be taken to demonstrate that medical devices, or more specifically medical beds, are adequately safe. We can argue that (see figure 2):

1. G1.1 A satisfactory set of safety requirements has been determined.
2. G1.2 Safety requirements are met in the actual design of the medical device.

3. G1.3 Safety requirements are met in operational use.
4. G1.4 Safety requirements continue to be met throughout the lifecycle of the medical device.

We can adopt the strategy taken in the RVSM Pre-Implementation Safety Case of arguing for each goal that there is sufficient direct evidence available, and that this evidence is sufficiently trustworthy.

Compared to current practice, where everything is addressed to and within the responsibility of the manufacturer, it is clear that healthcare service providers will have to provide some input. Objectives G1.3 and G1.4 exceed the control of the manufacturer. This crucially includes maintenance, as many serious accidents with technological systems relate to failures in the transition from maintenance mode to operational use and vice versa. Similarly, it cannot be assumed that the manufacturer can adequately manage operational risks through safety requirements that are met in the medical device. Rather, the service provider needs to demonstrate that arrangements are in place that satisfy assumptions made and ensure ongoing safe use and maintenance.

5.3 Outline of the Safety Case Structure

The four top-level goals G1.1 – G1.4 described above are then broken down until sufficient evidence has been provided that they are fulfilled, and an argument has been made that the evidence itself is sufficiently trustworthy.

G1.1 (A satisfactory set of safety requirements has been determined) is satisfied by arguing that a set of safety requirements has been identified, and that the safety requirements are complete, consistent and correct. The key strategy followed is the argument that relevant standards have been identified and addressed, and that the identified risks are sufficiently mitigated by the derived safety requirements. This is done by demonstrating that a risk management process according to ISO 14971 has been followed, and by providing the respective evidence.

G1.2 (Safety requirements are met in the design) is satisfied by arguing that the physical and functional properties of the medical device comply with the safety requirements, that procedure and training design comply with the safety requirements, and that any residual risks are tolerable.

G1.3 (Safety requirements are met in operational use) is satisfied by demonstrating that the guidance provided and assumptions made by the manufacturer of the medical device are taken into account by the service provider during operational use, that a hazard identification and risk assessment has been conducted by the service provider, and that risks have been sufficiently controlled through the specification of any required additional safety requirement.

G1.4 (Safety requirements continue to be met throughout the lifecycle) is satisfied by reference to the quality and safety management system of the service provider, and by demonstrating that adequate communication channels between service provider, device manufacturer and corresponding regulatory authorities have been established.

The Appendix provides a sketch of a previous preliminary argument development that was created during a session of the EWICS Medical Devices subgroup.

5.4 Arguing Safety over the Product Lifecycle

As discussed in section 4, the relevant device standards currently address the responsibilities of the device manufacturer. Standards at the organizational level of service provision, such as the UK Department of Health Standards for Better Health [15], are not concerned with medical devices apart from the requirement to use only those devices that have been certified. In the case of medical beds – a Class I medical device – a process of self-certification on part of the manufacturer is all that is required. This implies that decisions about levels of acceptable risks and detailed documentation of hazards considered remain with the manufacturer.

When healthcare providers assemble different devices to create a system within their environment, the safety of the resulting system needs to be assured. To this end the service provider needs to ensure that medical devices are installed according to the manufacturer's instructions for use, that appropriate maintenance is available, and that training and support to the operational staff is provided.

Apart from issuing instructions for use, the manufacturer has little influence on the way the devices are actually used in practice. The manufacturer does not have detailed information about the specific environment and the processes within which the device will be operated within a particular healthcare provider's setting. In complex systems this is a serious cause for concern, as in this way the possible interactions between system components and interactions with the environment as well as the system's particular history will not have been accounted for [12].

The structure that was chosen for the safety case in this paper attempts to bridge this gap by arguing the safety of the medical device throughout its lifecycle. Goals G1.1 and G1.2 (Safety requirements, and safety requirements met in the design) are clearly addressed to the manufacturer, while goals G1.3 and G1.4 (safety requirements are met in operation, and continue to be met) are addressed to the service provider. While in this respect – and quite reasonably – these two parts can be regarded as two volumes of the safety case, the requirements for each have now considerable impact on the structure and the content of the other.

In practical terms, the question arises how the two volumes could be sensibly separated into independent entities that can be produced by manufacturers and by service providers at different points in time. Here, the differentiation between different types of safety cases proposed in the CENELEC standard EN 50129 for railway applications (now to become IEC 62425) [25, 26, 27] could be a useful starting point. EN 50129 proposes three types of safety cases[1]:

- Generic Product Safety Case: provides an argument that a product can be used in safety-related applications; it specifies conditions that must be fulfilled in any safety-related application, where the product will be part of, and it provides descriptions of relevant characteristics of the product.

[1] Although EN 50129 is a standard for railway signaling systems, its definition of safety cases and their interrelationships are generic and are, in fact also applied outside the railway signaling field.

- Generic Application Safety Case: provides an argument that a configuration of products is adequately safe, without presupposing specific products.
- Specific Application Safety Case: provides an argument that a particular combination of specific components is adequately safe.

Each type of safety case specifies explicitly safety-related application conditions, the compliance with which has to be demonstrated in each safety case utilising that safety argument.

In the case of medical beds, the manufacturer would need to produce a *Generic Product Safety Case* (Volume 1 in fig. 2). As part of this, the device manufacturer needs to explicitly disclose all relevant assumptions made on the application, as well as all decisions regarding the acceptability of risks and the resulting mitigation. In addition, the manufacturer needs to demonstrate explicit forethought about the service provider's responsibility of ensuring safety during operation and throughout change. This entails, for example, documentation about appropriate procedures to operate the device, and the training needs of operators.

On the other hand, the service provider would need to produce a *Specific Application Safety Case* (Volume 2 in fig. 2), discharging the responsibility of demonstrating that the requirements established by the manufacturer as well as all assumptions explicitly made, are and continue to be satisfied during operation. In addition, as it is acknowledged that control of operational risks through safety requirements established by the manufacturer based on the device level is inappropriate, the service provider has to identify additional safety requirements based on their own operational environment. This is a very big change from the current practice of auditing that is carried out in order to collect indicators of safe practice.

Finally, to ensure continuing safe operation of the medical device in the operational environment, the service provider has to demonstrate that incidents are picked up, performance is monitored, the impact of changes is assessed, and crucially that effective communication channels to manufacturers and the relevant regulatory authorities are established. This responsibility is reflected on the part of the manufacturer by similar requirements that ensure that mechanisms for detecting and recording incidents and abnormalities are designed (where appropriate), and that arrangements are in place to receive and to react to data provided from the service providers or from regulatory authorities.

As a matter of speculation, we could envisage something similar to a *Generic Application Safety Case* being produced by professional bodies as guidance for the development of the specific safety cases to be produced by the service providers.

6 Opportunities and Challenges

6.1 Opportunities

The systematic consideration of safety through the development of goal-based safety cases has proven useful in industries such as aviation. The same benefits could be

expected in healthcare. Goal-based safety cases using graphical representation are easy to communicate, and can therefore address the variety of stakeholders of different (non-safety, non-engineering) backgrounds. Goal-based approaches also offer greater flexibility and are more suitable to incorporate novel technologies and methods.

In healthcare the disjoint regulation of device manufacturers and service providers has led to a situation where data from the two areas is usually not integrated, and where the device manufacturer defines both the normal operational context as well as acceptable levels of risk. Assurance of medical devices that have been put together by the service provider to form a particular system is hard to achieve. The development of a goal-based safety argument that demonstrates safety throughout the lifecycle of a device is an attempt at integrating data from manufacturers and service providers.

This approach could be a useful step towards whole system safety cases, e.g. for hospitals. This would be highly desirable as the individual device level usually is insufficient to assure safe operation, as the introduction of any device may have far-reaching unanticipated organizational reverberations.

6.2 Challenges

Healthcare does not have the same long tradition of reasoning about safety in systemic and explicit terms. Risk management in many healthcare organizations is still very preliminary and often includes only reactive approaches following incidents.

There is a split in the regulation between device manufacturers (certification) and service providers (audit). It is not clear who would be responsible for delivering such a safety case. Even if the safety case were split into separate volumes for manufacturers and service providers (e.g. along the lines of EN 50129 as proposed above), we may expect serious regulatory confusion as to which body is responsible for setting specific standards and requirements.

Many medical devices by themselves are not critical and do not require the development of a full safety case. However, in their specific application they may contribute to critical failures. The complexity of whole system safety cases needs to be addressed in future, as well as the process of integration of device manufacturers and service providers in the development of safety arguments.

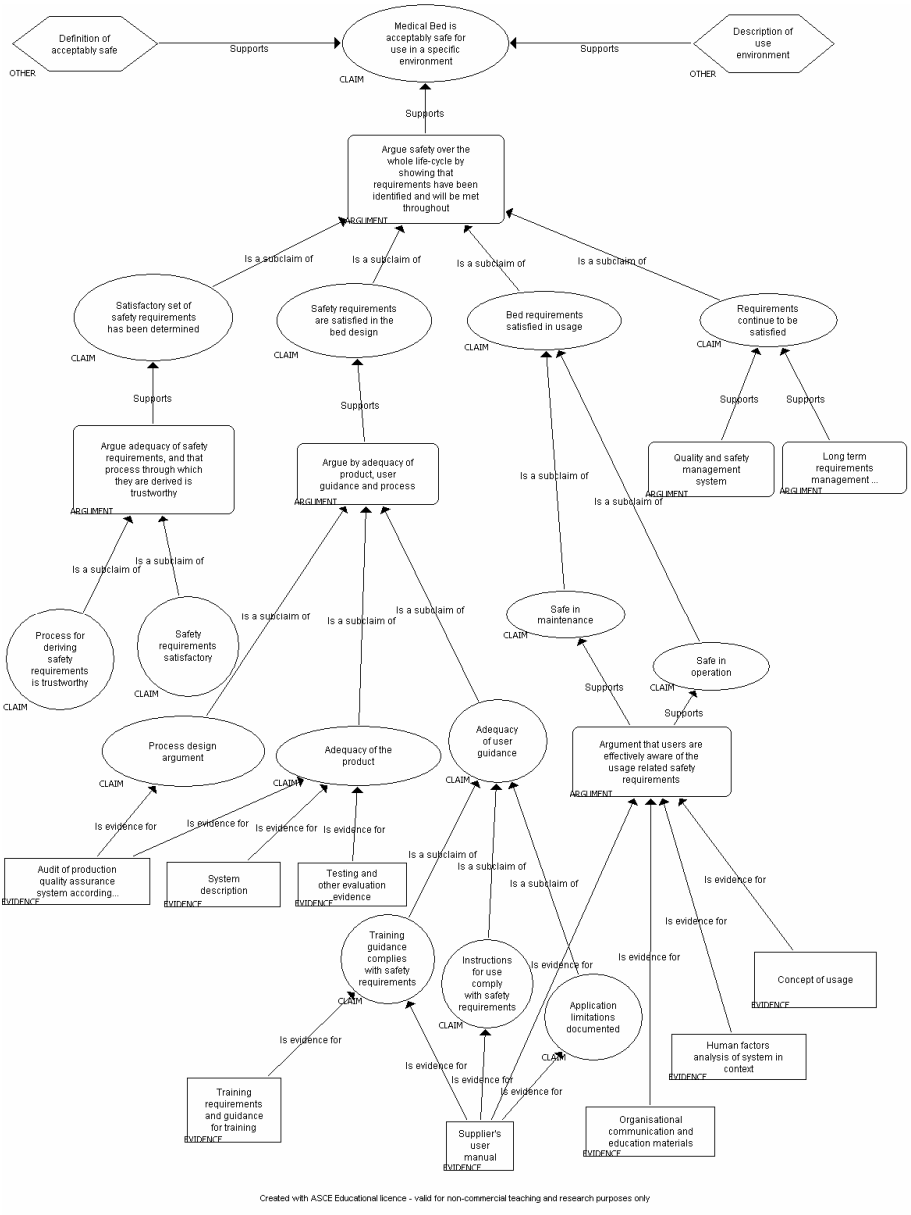
Acknowledgments. Part of the presented work is based on discussions and work conducted within EWICS TC 7, Subgroup on Medical Devices. The safety argument was produced using the free academic license of the Adelard Safety Case Environment. We are grateful to Odd Nordland for input concerning EN 50129.

References

1. Kelly, T.: A Systematic Approach to Safety Case Management. In: Kelly, T. (ed.) Proc. of SAE 2004 World Congress (2004)
2. Bishop, P., Bloomfield, R., Guerra, S.: The Future of Goal-Based Assurance Cases. In: Proc. Workshop on Assurance Cases, pp. 390–395 (2004)
3. Toulmin, S.: The Uses of Argument. Cambridge University Press, Cambridge (1958)

4. DS 00-56 Issue 3: Safety Management Requirements for Defence Systems, Ministry of Defence (2004)
5. Kelly, T., McDermid, J., Weaver, R.: Goal-Based Safety Standards : Opportunities and Challenges. In: Proc. of the 23rd International System Safety Conference (2005)
6. Kelly, T., McDermid, J.: A Systematic Approach to Safety Case Maintenance. Reliability Engineering and System Safety 71, 271–284 (2001)
7. Kelly, T.: Arguing Safety, DPhil Thesis, University of York (1998)
8. Bloomfield, R., Bishop, P., Jones, C., Froome, P.: ASCAD – Adelard Safety Case Development Manual, Adelard (1998)
9. McDermid, J.: Support for safety cases and safety argument using SAM. Reliability Engineering and System Safety 43(2), 111–127 (1994)
10. Emmet, L., Cleland, G.: Graphical Notations, Narratives and Persuasion: a Pliant Approach to Hypertext Tool Design. In: Proc. of ACM Hypertext (2002)
11. Chinneck, P., Pumfrey, D., McDermid, J.: The HEAT/ACT Preliminary Safety Case: A case study in the use of Goal Structuring Notation. In: 9th Australian Workshop on Safety Related Programmable Systems (2004)
12. Sujan, M., Harrison, M., Pearson, P., Steven, A., Vernon, S.: Demonstration of Safety in Healthcare Organisations. In: Proc. Safecomp 2006, Springer, Heidelberg (2006)
13. Medical Devices Regulations 2002. The Stationery Office Limited, London (2002)
14. European Council: Council Directive 93/42/EEC of 14 June 1993 concerning medical devices. Official Journal L 169, 12/07/1993, pp. 0001 – 0043 (1993)
15. Standards for Better Health, UK Department of Health (2004)
16. Bloomfield, R., Littlewood, B.: On the use of diverse arguments to increase confidence in dependability claims. In: Besnard, D., Gacek, C., Jones, C.B. (eds.) Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, pp. 254–268. Springer, Heidelberg (2006)
17. IEC 60601-1 – Ed. 3.0 – Medical electrical equipment – Part 1: General requirements for basic safety and essential performance. IEC Geneva (2005)
18. ISO 14971:2007 – Application of risk management to medical devices. ISO Geneva (2007)
19. IEC 62304 – Ed. 1.0 – Medical device software – Software life cycle processes. IEC Geneva (2006)
20. IEC 62366 – Ed. 1.0 – Medical devices – Application of usability engineering to medical devices. Draft. IEC Geneva (2006)
21. 2nd EWICS MeD Workshop, Edinburgh (unpublished report) (2004)
22. Moore, S.: Integrating the Healthcare Enterprise - IHE NA 2007 Connectathon Fact Sheet (2006) Retrieved from (accessed 2007-03-19) www.ihe.net/Connectathon/upload/NA_2007_Connectathon_Fact_Sheet_1.pdf
23. IEC 60601-2-52 – Ed. 1.0 – Medical electrical equipment – Part 2-52: Particular requirements for basic safety and essential performance of medical beds. Draft. IEC Geneva (2006)
24. RVSM Pre-Implementation Safety Case, Eurocontrol (2001)
25. CENELEC EN 50129 – Railway Applications – Safety related electronic systems for signaling, CENELEC Brussels (2003)
26. Nordland, O.: Safety Case Categories – Which One When? In: Redmill, F., Anderson, T. (eds.) Current issues in security-critical systems, pp. 163–172. Springer, Heidelberg (2003)
27. Kelly, T.: Managing Complex Safety Cases. In: Proc. 11th Safety Critical Systems Symposium, Springer, Heidelberg (2003)

Appendix: High-level argument structure for demonstrating the safety of medical beds



Electronic Distribution of Airplane Software and the Impact of Information Security on Airplane Safety

Richard Robinson[1], Mingyan Li[1], Scott Lintelman[1], Krishna Sampigethaya[2], Radha Poovendran[2], David von Oheimb[3], Jens-Uwe Bußer[3], and Jorge Cuellar[3]

[1] Boeing Phantom Works, Box 3707, Seattle, WA 98124, USA
{richard.v.robinson,mingyan.li,scott.a.lintelman}@boeing.com
[2] Network Security Lab, University of Washington, Seattle, WA 98195, USA
{rkrishna,rp3}@u.washington.edu
[3] Siemens Corporate Technology, Otto-Hahn-Ring 6, 81730 München, Germany
{david.von.oheimb,jens-uwe.busser,jorge.cuellar}@siemens.com

Abstract. The general trend towards ubiquitous networking has reached the realm of airplanes. E-enabled airplanes with wired and wireless network interfaces offer a wide spectrum of network applications, in particular electronic distribution of software (EDS), and onboard collection and off-board retrieval of airplane health reports. On the other hand, airplane safety may be heavily dependent on the security of data transported in these applications. The FAA mandates safety regulations and policies for the design and development of airplane software to ensure continued airworthiness. However, data networks have well known security vulnerabilities that can be exploited by attackers to corrupt and/or inhibit the transmission of airplane assets, i.e. software and airplane generated data. The aviation community has recognized the need to address these security threats. This paper explores the role of information security in emerging information technology (IT) infrastructure for distribution of safety-critical and business-critical airplane software and data. We present our threat analysis with related security objectives and state functional and assurance requirements necessary to achieve the objectives, in the spirit of the well-established Common Criteria (CC) for IT security evaluation. The investigation leverages our involvement with FAA standardization efforts. We present security properties of a generic system for electronic distribution of airplane software, and show how the presence of those security properties enhances airplane safety.

1 Introduction

Safety concerns with airplane software have been extensively studied [10], [15], [17]. The FAA stresses the criticality of some of the software onboard airplanes through well established guidance assuring their proper design and development for continued airworthiness, e.g. RTCA/DO-178B [1] Level A safety-critical software. Yet the guidance does not even address the issue of software distribution and its security. Nowadays, airplane software is still distributed manually using disks and other storage media, and since security is not a primary objective, embedded systems

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 28–39, 2007.

onboard airplanes check (using CRCs) only for accidental modifications of software to be loaded. However, the proposed use of networks to distribute software electronically from ground to onboard systems raises unprecedented challenges to ensuring airworthiness [7], [11]. In particular, while the electronic distribution of software (EDS) reduces overhead and improves efficiency and reliability of airplane manufacturing, operation and maintenance processes, these benefits come only at the cost of exposing the airplane to potential attacks, in particular via data networks. The FAA has recognized that current guidance and regulations for airplane software do not cover the requirements needed to address these vulnerabilities [4], [5].

1.1 Safety vs. Security

Although information security requirements are warranted, assessing their impact on airplane safety is non-trivial. It is clear from the established FAA guidance in [1] and elsewhere that the regulatory community is concerned about assuring the design and implementation of certain software components and that they consider that safety may be affected if such components were to become corrupted. Therefore, vulnerabilities in an EDS may present opportunities for attackers seeking to directly lower airplane safety, e.g. by corrupting safety-critical software distributed onboard, or to impede usability of onboard systems, e.g. by corrupting less critical software such as DO-178B [1] Level D. One must assume that international terrorists, as well as criminals pursuing economic damage, are capable today of employing advanced technologies for attacks. Thus it is now necessary to assess the impact of information security attacks against airplane safety and to develop strategies for mitigating the associated vulnerabilities. There is a body of literature that presents arguments for commonality among the safety and security disciplines [8], [9], [12], [16], but it remains an open question how to integrate the two fields. While indeed security affects safety, it is not clear how to express the relevant security considerations, and how to accommodate security risks and mitigations in the context of a safety analysis. There exist as yet no formal or agreed guidelines for certifying or assessing safety critical systems together with their security needs. In particular two questions remain open:

- How to integrate the mainly discrete methods applied in security analysis into the quantitative, probabilistic approaches typical of reliability analysis?
- How to combine the analysis of security, which refers to non-functional properties, with the functional SW correctness analysis in order to achieve a defined overall system safety level?

We believe that more research in this area is needed. Besides our necessarily limited contributions, we would like to benefit from any scientific advances there.

1.2 Our Contributions

The contributions of this paper are two-fold.

- We present security requirements for a generic EDS system, called Airplane Assets Distribution System (AADS). Our approach is based on the Common Criteria (CC) [3] methodology, identifying threats to AADS from an adversary

attempting to lower airplane safety, deriving objectives to cover the threats, and stating functional requirements to cover the objectives.

- We assess the implications of information security threats on airplane safety. Our approach is based on the Information Assurance Technical Framework [2] to analyze the CC Evaluation Assurance Level (EAL) necessary and sufficient to address threats against the integrity of software of highest criticality.

2 Airplane Assets Distribution System (AADS)

The electronic distribution of airplane *information assets*, i.e. software and data, can be modeled by a generic system that we call the Airplane Assets Distribution System (AADS). Figure 1 illustrates the AADS model with the entities and flow of assets. Not all entities interact directly with all others. Note that functional overlaps are possible, with a single entity assuming roles of multiple entities, e.g. an airplane manufacturer can be a supplier for some software. The nature and content of interactions change depending on the lifecycle state of a specific airplane, which can be: in development, assembly, testing, use, resale, etc. The responsibility of the AADS for an asset begins when it takes over the asset from its producer, e.g. supplier or airplane, and ends when it delivers the asset at its destination, i.e., embedded systems such as a Line Replaceable Unit (LRU) in an airplane, or at the consumer of airplane-generated data. The path between the producer and the destination of the asset is referred to herein as the *end-to-end path*. Each of the links in this path must fulfill the security objectives given in Section 3.

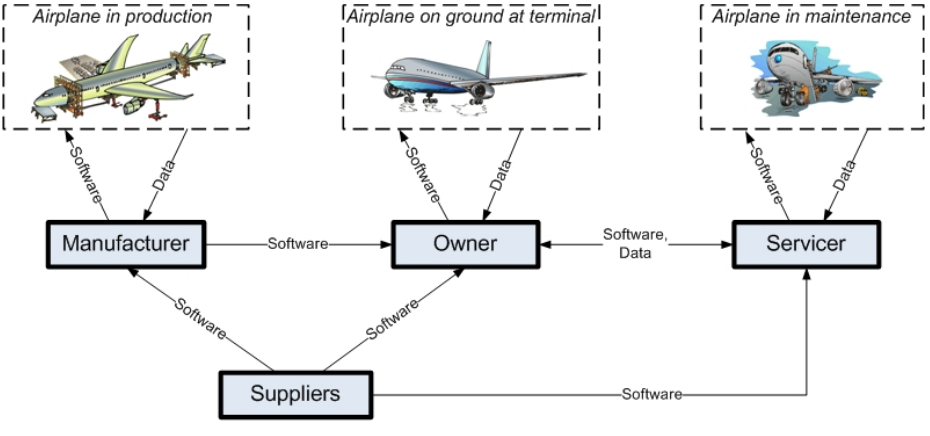


Fig. 1. Airplane Assets Distribution System (AADS)

2.1 Assumptions

Processes in each entity in the AADS are assumed to be operating as designed and expected. In particular, the AADS is assumed to be administered in a proper way. Access privileges must be assigned and managed appropriately at each entity.

Passwords and private keys are kept secret, and certificates are properly managed and protected. Each supplier is accountable to produce safety-assured software as per [1]. The networks used for asset distribution are assumed to be robust against well known denial of service attacks. It is worth noting that software distribution via physical media is generally adequate to meet requirements for timely software delivery to an aircraft. Finally, it is assumed that airplane owners are capable to manage the software configurations of airplanes reliably and correctly, and that airplanes produce status information accurately.

2.2 Adversary Model

An adversary in the AADS model is assumed to be capable of passive network traffic analysis as well as active traffic manipulation, node impersonation, and insider attacks. The objective of the adversary is to actually lower the safety margins of airplanes (as in the case of international terrorists) and/or to induce safety *concerns* and disturb business (as would be expected of sophisticated hackers or international criminal organizations).

For purposes of the present analysis, we consider the scope of adversarial attacks to be limited to security attacks over data networks. The process of loading software on LRUs within an airplane is assumed to be sufficiently protected with specific physical, logical, and organizational inhibitors, checks, and control. Loading is only performed at specified times, for example, when the airplane is in maintenance mode, and by authorized personnel using authorized equipment. Moreover, certain checks are in place to enable detection of corrupted software, e.g. checking the list of parts to be loaded with a configuration list provided by the airline, and if software is compatible with the destination LRU hardware and software environment.

Furthermore, it can be assumed that due to software and hardware redundancies (e.g. several code instances executing in parallel on different system platforms on the airplane), most unintentional or unsophisticated corruptions or misconfigurations in safety-critical software are detectable at least when loaded into an LRU. Therefore, to effectively cripple a safety-critical function in the airplane, the representation of software must be modified at several positions. This significantly increases the effort needed from the adversary.

Based on the motivation and impact of adversarial attacks over networks, we can classify security threats as described next.

2.3 Safety Threats

The adversary can attack the AADS to threaten airplane safety. We identify the following specific threats that could amount to sabotage of the airplane.

Asset Corruption. The contents of distributed software can be altered or replaced (in an undetectable manner) to provoke accidents. This type of corruption to airplane-loadable software is sometimes referred to as *coherent corruption*, emphasizing a distinction from arbitrary bit-substitutions, which generally would render a software component unloadable. Airplane-generated data can be also corrupted to threaten airplane safety, e.g. by altering safety-related reports.

Software Misconfiguration. In order to cause havoc, a mismatch between the airplane's intended and actual configuration can be provoked by preventing delivery of software, deleting software, or injecting inappropriate software during distribution.

Asset Diversion. Software can be diverted to an unsuitable recipient to provoke accidents, e.g. by disturbing the execution of other software at that destination.

Asset Staleness. The revocation and update of software that need to be changed for safety reasons can be blocked and delayed, thus impeding the distribution processes.

2.4 Business Threats

The adversary can attack the AADS to induce unjustified airplane safety concerns or to cause flight delays, and thereby present threats to business of airplane manufacturer and/or owner.

Asset Unavailability. Assets can be made inaccessible or unusable, for example by jamming asset distribution to disrupt airplane service.

Late Detection. Assets can be intentionally corrupted so that the tampering is detected late enough for the airplane to be put out of service. For example, when tampering of software is not detected during distribution from ground systems to airplane, but is detected only upon final load at the destination LRU in the receiving airplane. Software corruption that is detectable by an LRU, or whose installation renders the LRU non-functional, is distinct from that referred to above as *coherent corruption*.

False Alarm. Assets can be tampered to cause economic damage from misleading safety concerns. In particular,, corruption of configuration reports might cause an airplane to appear as if incorrectly configured, creating unwarranted flight delays from the misleading safety concerns.

Repudiation. Any entity in the AADS could deny having performed *security-relevant actions*, e.g. deny having distributed or received some software.

3 Securing AADS

The threats listed in the previous section must be countered and mitigated by appropriate security objectives, which in turn must be implemented using suitable mechanisms. This section presents the security objectives to counter the security threats listed above, followed by an overview of the mechanisms proposed to achieve them, as well as a brief rationale why they should be sufficient for this purpose.

3.1 Safety-Relevant Security Objectives

1. *Integrity.* For every asset that is accepted at its destination, its identity and contents must not have been altered on the way—it must be exactly the same as at the source of the distribution. This includes protection against injection of viruses and other malicious code.

2. *Correct Destination.* An airplane must accept only those assets for which it is the true intended destination.
3. *Correct Version.* An airplane must accept assets only in the appropriate version.
4. *Authenticity.* For every security-relevant action, the identity of entities involved must be correct. This applies in particular to the alleged source of an asset.
5. *Authorization.* Whenever an entity performs a security-relevant action, it must have the authorization or privilege to do so. Otherwise the action must be denied.
6. *Timeliness.* Required software installations and updates must be capable of being performed and confirmed by appropriate status reports within a specified period of time. Note that otherwise the airline's configuration management (which is not strictly part of the AADS) must take a deliberate decision whether the respective airplane is still considered airworthy.

3.2 Business-Relevant Security Objectives

7. *Availability.* All necessary assets must be available in a time window adequate to support regulatory requirements and business needs.
8. *Early Detection.* The fact that attackers have tampered with assets must be detected as early as possible; that is, by the next trusted entity handling it. In particular, a tampered part should be detected well before actually being loaded by its destination LRU.
9. *Correct Status Reporting.* Status information concerning asset disposition, in particular reports listing the contents of the current airplane on-board parts storage, must be kept intact during transport in order to avoid false claims about, for instance, missing parts.
10. *Traceability.* For every security-relevant action, as well as unsuccessful attempts to perform such actions, all relevant information must be kept for a period of time sufficient to support regulatory requirements and business needs, such as general short-term security audits. This information includes the identity of entity involved, the action type with essential parameters, and a timestamp.
11. *Nonrepudiation.* To support forensics, for instance after an airplane crash, entities must not be able to deny their security-relevant actions. Evidence for this must be verifiable by third parties and must be long-lived: at least 50 years.

Table 1 (see overleaf) shows which security objectives mitigate which threats. The mechanisms employed in the AADS to address the above objectives are described next.

3.3 Securing Distributed Assets Using Digital Signatures

Digital signatures constitute the main mechanism to secure distributed assets in the AADS. We note that the choice of using digital signatures, as opposed to other integrity protection solutions such as keyed hashes and virtual private networks

Table 1. Security threats and objectives to cover them

Objectives \ Threats		Safety				Business			
		Cor-ruption	Misconfig-uration	Diver-sion	Stale-ness	Asset Un-availability	Late Detection	False Alarm	Repudi-ation
Safety Relevant	Integrity	√							
	Correct Destination			√					
	Latest Version				√				
	Authenticity	√	√						√
	Authorization	√	√						
	Timeliness				√				
Business Relevant	Availability					√			
	Early Detection						√		
	Correct Status Reporting							√	
	Traceability	√	√						√
	Nonrepudiation								√

(VPN), is made in order to additionally provide nonrepudiation of origin as well as data authenticity and data integrity across multiple AADS entities. The message sent from a source to destination appears as follows:

$$asset, metadata, sign_{source}(asset, metadata) \quad (1)$$

where $sign_X$ denotes a signature with the private key of entity *X*, and *metadata* denotes additional information associated with the asset or its handling. Common examples include the destination (or a class of destinations) constituting the intended delivery target for assets, and timestamps or similar tags that can be used to inhibit later replay. Using the public key of the source and the hash function, the receiver can check all the information received. In this way, the signature ensures the integrity of the asset and the metadata, thus covering also freshness and the correctness of the destination.

A major challenge remains with respect to authenticity (and the related authorization requirement): how does the receiver reliably know the public key of the source? The management of identities and associated keys and certificates is an important task [13], requiring implementation of key management facilities or availability of a PKI. A Public Key Infrastructure (PKI) [6] consists of a Registration Authority (RA) to authorize key/certificate requests from entities, a Certification Authority (CA) to generate and issue asymmetric key pairs and corresponding digital certificates for requesting entities and to determine validity of certificates, and a Certificate Repository to store and distribute certificates. An airline may assign the role of RA and/or CA to a trusted third party, e.g. a government agency or commercial vendor. Alternatively, an airline can implement its own PKI and itself function as RA and CA.

Relying on a PKI, the source can simply append to its message a standard digital certificate, $cert_{source}$, provided by a CA trusted by the receiver:

$$cert_{source} = sign_{CA}(id_{source}, K_{source}, id_{CA}, validityperiod) \quad (2)$$

where id_X is an identifier for entity *X and* K_{source} is the public key of *source*. The receiver can check the certificate, needing to know only the public key of the CA, and thus obtain and verify the authenticity of K_{source}.

Yet a PKI is a complex system, which in turn needs to be certified, which is a major undertaking in itself. Driven by the considerations briefly shared in section 4.2, we are currently investigating light-weight alternatives to PKI.

The verification of asset signatures can be end-to-end or entity-to-entity, as follows.

Entity-to-entity integrity protection. For every signed asset, each intermediate entity verifies and re-signs it, and then forwards it to the next entity along the desired path for that asset. Depending upon business requirements, and state of an asset's life-cycle or workflow, re-signing may constitute replacement of an existing signature or addition of a new one. In an entity-to-entity arrangement, localized key management capabilities suffice to establish trust and authenticity.

End-to-end integrity protection. Each asset, signed by its producer, is verified at each intermediate entity as well as by the final destination. The end-to-end architecture may be argued to have stronger security properties than the entity-to-entity architecture.

Implementing an end-to-end architecture requires distributed entities to have access to information about more identities than merely those of their immediate neighbors. Generally, this means making use of a mature public key infrastructure. The main security advantage of end-to-end architecture is that the final receiver need not trust intermediate entities but just the first sender whom it can authenticate directly. Intermediate entities cannot undetectably tamper with data in transit.

As the life-cycle of AADS-distributed parts evolves, the practical lifetime of signatures must be considered. The cryptanalytic capabilities available to attackers improve over time, and the potential for compromise of secret keys increases. Signature lifetimes may be extended via periodic refreshment or replacement of signatures. New signatures can be based on longer keys and improved cryptographic algorithms, as they become available.

3.4 Other Security Mechanisms

Security-relevant actions like releasing, approving, ordering, receiving, and loading software, as well as issuing and revoking certificates must be authorized. This can be achieved, for instance, via role-based access control or certificate-based capabilities.

In order to support traceability, all security-relevant actions, as well as unsuccessful attempts to perform such actions, are timestamped and logged. Logs must be implemented with tamper-proof storage.

High availability can be achieved with host and network protection mechanisms, for instance efficient filtering, channel switching, and redundant storage and bandwidth.

3.5 Coverage Analysis

The security mechanisms given in sections 3.3 and 3.4 suffice to cover required EDS security objectives given in sections 3.1 and 3.2, as described below. A more in-depth examination of the requirements coverage is contained in [14].

The *integrity* and *authenticity* of assets is guaranteed by the digital signature of the source and the corresponding public key or certificate(s), with the validity period of the signatures extended by refreshing them. Checking signatures as soon as possible during transmission (i.e. at each intermediate entity) contributes to *early detection* of improper contents. In the source signed asset, the timestamp together with version numbers ensures that an outdated asset is not accepted, satisfying *latest version*, in accordance with the principle that airlines must be responsible for managing the configurations of the aircrafts they own. Further, by including the intended destination among signed meta-data with distributed assets, diverted assets are not accepted, meeting *correct destination*. None of the above mechanisms can mitigate insider attacks, though appropriate access controls ensure that critical actions are initiated by *authorized* personnel only.

Signatures for integrity protection of status information and authorization of status-changing actions contribute to *correct status reporting* of information on assets. Signatures and audit logs are sufficient for achieving *nonrepudiation* and *traceability*.

Although *availability* cannot be fully guaranteed in AADS, existing techniques can be used to mitigate jamming attacks. Backup mechanisms, such as traditional physical transfer of storage media using bonded carriers, can be used to reduce impact of non-availability of assets or asset distribution. *Timeliness* relies on availability, timestamping and organizational measures: in case of asset uploads being due, the providers must notify the respective receivers in a timely way and specify the new version numbers as well as a deadline by which the assets must have been loaded. Moreover, they must make sure that the assets are available for being pulled by the receivers during the required period of time.

4 Assurance Levels and Impact on Safety

In this section we present our analysis of the implications of security threats to the safety of airplanes, and determine the minimum assurance levels that must be met by AADS. Moreover, we mention pragmatic considerations on achieving them.

4.1 Determination of Assurance Levels

The Threat Level for the expected threat source on airplane *safety*, according to [2], is that of international terrorists, i.e. T5 - sophisticated adversary with moderate resources who is willing to take significant risk. Some software is of ultimate

criticality for flight safety and is assigned RTCA/DO-178B [1] Level A[1], and thus according to [2] have Information Value V5 - violation of the information protection policy would cause exceptionally grave damage to the security, safety, financial posture, or infrastructure of the organization. Since the failure of parts with software Level A is catastrophic, so too can be the effect of not achieving the integrity and authenticity protection that should be guaranteed by the AADS distributing such software. According to [2], the above assigned Threat Level T5 and Information Value V5 together imply selection of EAL 6.

To address security concerns emerging from *business threats*, an assurance level of EAL 4 is sufficient, as follows. The Threat Level according to [1] for the expected business Threat Source is that of organized crime, sophisticated hackers, and international corporations, i.e. T4 - sophisticated adversary with moderate resources who is willing to take little risk. Attacks against the availability of assets can cause major damage to airlines from the business perspective, by putting individual airplanes out of service. Moreover, one must be able to counter attacks against software that have a highly visible effect to passengers, in particular if they affect more than one airplane. For example, a hacker could corrupt Level D or Level E software e.g. controlling the cabin light or sound system, for which generally no strong defense may exist. In this way the attacker could create anomalies to provoke safety *concerns*. This can cause severe damage to the reputation of both the airline and the airplane manufacturer, in particular as it might appear that little confidence can be put on their ability to protect other, highly critical software in the airplane. This could cause the whole fleet to be grounded, even though mainly for psychological reasons. From scenarios like these, we propose that there are assets that have a business Information Value of V4 - violation of the information protection policy would cause serious damage to the security, safety, financial posture, or infrastructure of the organization. Given a Threat Level T4 and a Information Value V4 for parts, according to [2], EAL 4 is sufficient.

4.2 Pragmatic Issues

An assurance level of EAL 4 permits maximum assurance for the development of the AADS with the use of positive security engineering based on good commercial development practices [3]. Although these practices are rigorous, they do not require significant specialist knowledge, skills, and other economically taxing or time consuming resources.

As mentioned in Section 3.3, the state-of-the-art requires the AADS to make use of digital signatures which rely on some form of key management. Unfortunately, the maximum assurance level of current commercially available Public Key Infrastructure is EAL 4, and the practical value of evaluating the system to a level higher than its PKI environment can support is questionable. This could motivate specification of assurance for the AADS at the highest EAL available for PKI, which currently is EAL 4. Yet EAL 4 would be insufficient for the integrity protection needs of Level A

[1] For software of lower criticality level (B thru E according to [1]), some lower value would be sufficient, but since the AADS should uniformly handle software of all criticality levels, the desired EAL with respect to *safety threats* should be the one for Level A software.

software. Moreover, evaluating a whole system as complex as an AADS at an assurance level of EAL 6 would be extremely costly.

As a viable solution to the discrepancy just described, we suggest a two-level approach where the mechanisms covering the most critical safety-relevant objectives, namely those which counter the threat of corrupted software (i.e., integrity, authenticity, and authorization), reach EAL 6, while the remaining components are kept at EAL 4. Since the mechanisms requiring EAL 6 include key management, it is necessary to raise the certification of an existing PKI to that level, or to implement the necessary functionality within the highly-critical part of the AADS. Designing the AADS architecture such that the key management for the EAL 6 components is minimized should make the high-level certification effort bearable.

5 Conclusions and Future Work

In this paper, we have studied the safety and security aspects of electronic distribution of software (EDS) and data. We have identified information security threats to airplane safety emerging from attacks on safety-critical software. Additionally, we have found that attacks on less critical (and hence less protected) software controlling onboard utility systems can induce unwarranted and misleading safety concerns, impeding business of airplanes. We have proposed a secure EDS system, Airplane Assets Distribution System (AADS), which addresses the threats and serves as a guideline for design and evaluation of EDS systems implemented for use with airplanes. Further, we have evaluated the impact of security threats on safety, and suggested suitable assurance levels for enabling a Common Criteria security evaluation of EDS system implementations. Concerning the assurance assessment and certification effort for AADS, we have proposed a two-assurance-level approach that addresses integrity protection of safety-critical software while keeping evaluation cost manageable.

It is hoped that the security requirements described above and the analysis detailed in the AADS Protection Profile [14], will provide a permanently useful public reference, and that they may be adopted by the regulatory community in much the same way as existing RTCA and other guidance have been. Several difficult and interesting issues remain to be investigated and resolved. Future work should focus on advancing the knowledge on the relations of security and safety analysis of an EDS system, including quantifying vulnerabilities to evaluate and certify a safety critical system under security threats, and correlating security assessment methods with development assurance guidelines in RTCA/DO-178B [1] as well as using this mapping for insights into the interaction between information security and airplane safety.

Acknowledgements

We would like to thank Prof. Peter Hartmann from the Landshut University of Applied Sciences and several anonymous reviewers for their insightful and valuable comments that helped us to improve specific sections of this paper.

References

1. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (RTCA) (1992)
2. Information Assurance Technical Framework, Release 3.1. US National Security Agency, http://www.iatf.net/framework_docs/version-3_1/
3. Common Criteria, http://www.commoncriteriaportal.org/
4. Federal Aviation Administration, 14 CFR Part 25, Special Conditions: Boeing Model 787–8 Airplane; Systems and Data Networks Security—Isolation or Protection from Unauthorized Passenger Domain Systems Access, [Docket No. NM364 Special Conditions No. 25–07–01–SC], Federal Register, vol. 72(71) (2007) http://edocket.access.gpo.gov/2007/pdf/E7-7065.pdf
5. Federal Aviation Administration, 14 CFR Part 25, Special Conditions: Boeing Model 787–8 Airplane; Systems and Data Networks Security—Protection of Airplane Systems and Data Networks From Unauthorized External Access, [Docket No. NM365 Special Conditions No. 25–07–02–SC], Federal Register, vol. 72(72) (2007) http://edocket.access.gpo.gov/2007/pdf/07-1838.pdf
6. Adams, C., Lloyd, S.: Understanding PKI: Concepts, Standards, and Deployment Considerations, 2nd edn. Addison-Wesley, Reading (2003)
7. Bird, G., Christensen, M., Lutz, D., Scandura, P.: Use of integrated vehicle health management in the field of commercial aviation. NASA ISHEM Forum (2005) http://ase.arc.nasa.gov/projects/ishem/Papers/Scandura_Aviation.pdf
8. Brostoff, S., Sasse, M.: Safe and sound: a safety-critical approach to security. In: ACM workshop on new security paradigms, pp. 41–50 (2001)
9. Ibrahim, L., Jarzombek, J., Ashford, M., Bate, R., Croll, P., Horn, M., LaBruyere, L., Wells, C.: Safety and Security Extensions for Integrated Capability Maturity Models, United States Federal Aviation Administration (2004), http://www.faa.gov/about/office_org/headquarters_offices/aio/ documents
10. Leveson, N.: Safeware: System Safety and Computers. Addison Wesley Longman, Reading, Massachusetts (1995)
11. Lintelman, S., Robinson, R., Li, M., von Oheimb, D., Sampigethaya, K., Poovendran, R.: Security Assurance for IT Infrastructure Supporting Airplane Production, Maintenance, and Operation. National Workshop on Aviation Software Systems (2006), http:// chess.eecs.berkeley.edu/hcssas/papers/Lintelman-HCSS-Boeing-Position_092906_2.pdf
12. Pfitzmann, A.: Why Safety and Security should and will merge, Invited Talk. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, Springer, Heidelberg (2004)
13. Robinson, R., Li, M., Lintelman, S., Sampigethaya, K., Poovendran, R., von Oheimb, D., Bußer, J.: Impact of Public Key Enabled Applications on the Operation and Maintenance of Commercial Airplanes. AIAA Aviation Technology, Integration and Operations (ATIO) conference (to appear, 2007)
14. Robinson, R., von Oheimb, D., Li, M., Sampigethaya, K., Poovendran, R.: Security Specification for Distribution and Storage of Airplane-Loadable Software and Airplane-Generated Data, Protection Profile, Available upon request (2007)
15. Rodriguez-Dapena, P.: Software safety certification: a multidomain problem. IEEE Software 16(4), 31–38 (1999)
16. Stavridou, V., Dutertre, B.: From security to safety and back, Conference on Computer Security. Dependability and Assurance, 182–195 (1998)
17. Weaver, R.: The Safety of Software - Constructing and Assuring Arguments, DPhil Thesis, Department of Computer Science, University of York, UK (2003)

Future Perspectives: The Car and Its IP-Address – A Potential Safety and Security Risk Assessment

Andreas Lang, Jana Dittmann, Stefan Kiltz, and Tobias Hoppe

Otto-von-Guericke University of Magdeburg
ITI Research Group on Multimedia and Security
Universitätsplatz 2
39106 Magdeburg, Germany
{andreas.lang,jana.dittmann,stefan.kiltz,
tobias.hoppe}@iti.cs.uni-magdeburg.de

Abstract. The fast growing Internet technology has affected many areas of human life. As it offers a convenient and widely accepted approach for communication and service distribution it is expected to continue its influence to future system design. Motivated from this successful spreading we assume hypothetical scenarios in our paper, whereby automotive components might also be influenced by omnipresent communication in near future. If such a development would take place it becomes important to investigate the influence to security and safety aspects. Based on today's wide variety of Internet based security attacks our goal is therefore to simulate and analyze potential security risks and their impact to safety constraints when cars would become equipped and connected with an IP based protocol via unique IP addresses. Therefore, our work should motivate the inserting of security mechanisms into the design, implementation and configuration of the car IT systems from the beginning of the development, which we substantiate by practical demo attacks on recent automotive technology.

Keywords: Security, Safety, Automotive, Future Vision.

1 Introduction

The growing IT technology offers many applications. Office computers are connected to each other and required business data is available everywhere. Home users use their computers for writing emails, chatting with friends and more. The Internet is for many applications one of the most important requirements for daily business. The connection between the computers and to the Internet conceals IT-security risks, which are used by attackers with different motivations. Since last decade many different forms of attacks are known and well studied. Thereby, an incident, which includes the attack and event, is introduced in a taxonomy [1], which is created in order to standardize the terminology used when dealing with incidents. This taxonomy helps the users of an IT system (for example system administrators or forensic teams) to identify and to classify threats, incidents, attacks and events. The safety aspect is not in focus of this taxonomy and neglected. In [2] this taxonomy is

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 40–53, 2007.

enhanced to identify the violated security aspects (confidentiality, integrity, authenticity, availability, non-repudiability) from the result of the attacks. Thereby it is shown that different attacked targets with different attack results violate different security aspects. The attacker and his running attacks specify which and how security aspects are violated. To focus on the safety other publications are available. In [3] assignments of safety integrity levels (SILs) are defined, which focus on the acceptable failure rate and controllability by the person concerned. Thereby 4 main levels (and one without safety) are defined. The abstractness of safety violation differs and increases between five safety levels which are as follows:

- Level 0: Not addressed. It means, that the probability of events regarding this safety level are reasonable possible with the effect, that they are nuisance only.
- Level 1: In this level are events categorized which are unlikely and would distract the person concerned.
- Level 2: In this level are events categorized, which have a remote change to occur and they debilitate the person concerned.
- Level 3: In this level are events categorized, which are very remote and if they occur, then they are difficult to control.
- Level 4: In this level are events categorized, which are extremely improbable, but when they occur, they are uncontrollable for the persons concerned.

Note, that these safety integrity levels primary focus on the probability of occurrence and the controllability by humans. For our potential attacker scenario, the probability of randomly or naturally failures can be neglected, because the potential attacker has the goal to attack the IT system.

The paper is structured as follows. In section 2 known threats and vulnerabilities on example attack scenarios are introduced and described. Furthermore, these attack scenarios are associated with the violated security aspects. Section 3 introduces our hypothetical risk assessment on exemplary selected attack scenarios for cars. Additionally, these car attack scenarios are discussed with the violated security aspects and violated safety level by focusing on its controllability. In section 4, we introduce a practical simulation on cars today and adapt this on cars using IP protocol based communication. Section 5 closes the paper by summarizing it and discussing the importance of future work.

2 Selection of Known Threats and Vulnerabilities of IP Based Protocols and Communication

The IT security, attacks and the prevention of the attacks are well studied and analyzed. Attackers try to find new strategies and types of attacks to compromise information, computers or networks. The CERT taxonomy [1] classifies the vulnerabilities into the three main classes: design, implementation and configuration. The vulnerability design exists, when the design of the system is not well analyzed and studied in focus of security. An implementation error provides an implementation vulnerability, which is mostly exploited by an attacker to get unauthorized access to the system. If the configuration of the IT system includes mistakes, then a configuration vulnerability can be used by attackers for compromising.

Many network based attacks (A_i) require an Internet Protocol (IP) based communication. Thereby, we classify the main basic attack principles, based on network communication as follows, introduced in the six scenarios shown in Figure 1. The assumption is that the normal data flow (A0) exists. An attacker can read and/or sniff the transmitted information (A1) whereby the sender and receiver do not notice it. If the attacker is between both communication partners (A2), than she/he can modify the data as man in the middle. An interruption (A3) occurs, when the source sends information, but the destination never receives them. The attacker can also create data and send them by spoofing the source address to the destination (A4). The last possibility of our basic attack principles of the attacker is to steal (A5) the transmitted data.

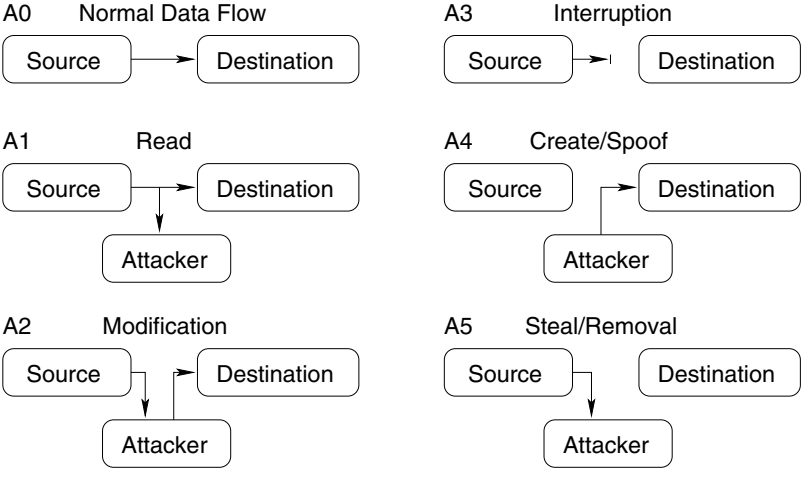


Fig. 1. General Attacking Strategies

Each of these basic attack principles address a different security aspect and if combinations of these attacks are performed by an attacker, then more different attack strategies which violate more security aspects can be described.

Depending on the type and goal of the attacker, different security aspects can be violated. Table 1 shows the association between the presented types of attacks (A1,…,A5) and the violated security aspects.

Table 1. Association between Attack Types and Violated Security Aspects

Security Aspect	A1	A2	A3	A4	A5
Confidentiality	X	X			X
Integrity		X		X	
Authenticity		X		X	
Non-Repudiability		X			
Availability			X	X	X

Note, that combination or special cases of the attacks also address other security aspects. It is shown, that different attack types violate different security aspects. Existing combinations of attacks or the modification of attacks have as result, that more than the introduced security aspects are violated.

In the literature [4] different practically useable attack scenarios (AS_i) are identified and described. Well known and often used attacks from attackers are selected exemplarily and briefly summarized and introduced in the following listing [5].

- **Sniffing attack – AS1:** An attacker who uses this type of attack gets unnoticed information like the attack A1. The source and receiver computers or users do not know that their communication is eavesdropped by an attacker. In a computer network, the attacker sniffs the transmitted IP packets and derived from it, she/he knows, for example, which services are used and which data are transmitted. The attacker can read the complete communication.
- **Man-in-the-Middle attack – AS2** (MITM)**:** This attack requires a direct eavesdrop of the communication between two computers, like A2. Instead of sniffing them, by using this type of attack, the attacker captures the transmitted data and modifies them with the content she/he wants and sends the modified data to the destination. In a computer network, the attacker can work as gateway or router, where all IP traffic has to pass trough. Without any security mechanisms, the sender and receiver do not notice the presence of the attacker.
- **Spoofing attack – AS3:** Is an attack, whereby the transmission appears to have come from another source, which is introduced within attack A4. Often, data are transmitted, which seems to be sent from a valid, authentic source computer or user. Typical examples are IP-spoofing, where the source IP address of the attacker is replaced by any or a valid IP address accepted by the receiver.
- **Replay attack – AS4:** By this type of attack, the attacker uses a combination of A1 and A4. She/he sniffs as first transmitted data (IP packets) from the network (A1). The second step is to resend the same data to the attacked computer (A4). If this attack is combined with the spoofing attack, then replayed IP packet seems to sent from the valid source computer.
- **Denial of Service attack – AS5** (DoS)**:** This attack tries to prevent valid users the access to a service, system or network. Mostly, the attacker floods the target with so many as possible IP packets per second or disrupts the target by alternation of the system configuration (for example packet filter) or destruction of physical components. By flooding, it is like the attack scenario A4, but the created IP packets are sent, for example, as often as possible per second. Note, that distributed denial of service (DDoS) is a subclass of DoS with the difference, that the attack is performed from many computers at the same time to the target, thereby increasing the effectiveness.
- **Malformed packet – AS6:** This attack is based on sending nonstandard IP packets, which produce unexpected results on the target computer, also a type of A4. Depending on the payload of the sent IP packet, the result can be, for example, crashing the attacked service or computer (DoS) or unauthorized access by exploiting a service.

- **Malicious code – AS7:** The usage of malicious code attacks the target system by running the attacker code which causes to harm the software or data on the target computer. The attacker can use the attack scenarios A2 or A4 to bring in the malicious code. Typical examples for this attack scenario are viruses, worms, Trojan horses or ticks [6].
- **Social Engineering – AS8:** This type of attack focuses on (the weak point) humans and is a special case of the introduced attacks. Because social engineering does not address, for example, the network communication or an IT system. The attacker can employ human- or computer based social engineering to persuade another person to compromise information or the system security. If IT systems are connected each other via untrusted networks, then computer based social engineering is also enabled for attackers, which can use all introduced attack principles (A1,…,A5).

The literature [7] divides the attacks into active and passive. A passive attack Ξ_P means that the attacker eavesdrops the communication without modifying anything of the communication. In contrast, an active attack Ξ_A is an attack, where the attacker modifies at least one bit of the communication, replays old messages or deletes a selected IP packet or bit from the wire.

Furthermore, all introduced and all other existing attacks can be used from internal Θ_I and external Θ_E attackers. An internal attacker is already located close to the target computer, has a network connection and has knowledge about the internal network topology. If the attacker is external, then she/he is somewhere in the Internet and attacks the target computer with one or a combination of the attacks from outside the attacked network (company, authority or home).

If the existing and well studied IT systems are used in the growing area of car and its IT, then the security risk and its safety implications should be analyzed and simulated, done exemplary in the following section.

3 Hypothetical Risk Assessments on the Selected Examples

In this section, we introduce a hypothetical risk assessment and we map the existing types of attacks on selected examples for automotive with its communications.

Imagine, in the near future, the electronic and IT part of available cars has maybe similar electronic technology like today (2007). If somebody travels by car, the car would be opened by using a wireless unique key. Thereby, the boot process of all devices and the main car control unit starts. It includes loading the BIOS (Basic Input/Output System), starting the device kernel with all available applications and services including the setup of the IP stack by initializing the car IP address and its Internet name like *car.car-producer.de*. This unique identification might also be used in the car-to-car (C2C) communication. Furthermore, the driver has to be identified by, for example, biometrics such as her/his voice and the car initializes all components to the user by assigning its personal internet identification like *driver@car.car-producer.de*. This identification is also used to adjust personal configuration settings like seat position, mirror and climate control, phone address book, entertainment system, private list of the navigation system and the energy and power setting, which is allowed for the user. In particular, the car has a permanent Internet connection to receive actual relevant information, like the traffic

announcements, device updates and security patches. Furthermore, the car sends its GPS location to different servers for registration and the username of the driver is used to enable VoIP (Voice over IP) communication. Additionally, the car insurance is informed, that the identified driver is using the car to enable the insurance on request and in accordance to driver profile (pay per drive contracts). To enable local communication between the cars (also used for car-to-car communication) mobile ad-hoc networks are created. Note that a potential attacker who used the mobile ad-hoc network to attack the victim is not permanent close to them. The mobility let the attacker drive away and it is difficult to trace and identify she/he. Figure 2 introduces existing communication networks for mobile cars. The mobile ad-hoc networks are used for short range communication to change local road or traffic information. In additional, the network topology of these networks permanently changes. For the long-term Internet connection, a link via UMTS (Universal Mobile Telecommunications System) or HSDPA (High-Speed Downlink Packet Access) is used, for example, to enable VoIP and to receive navigation and traffic information.

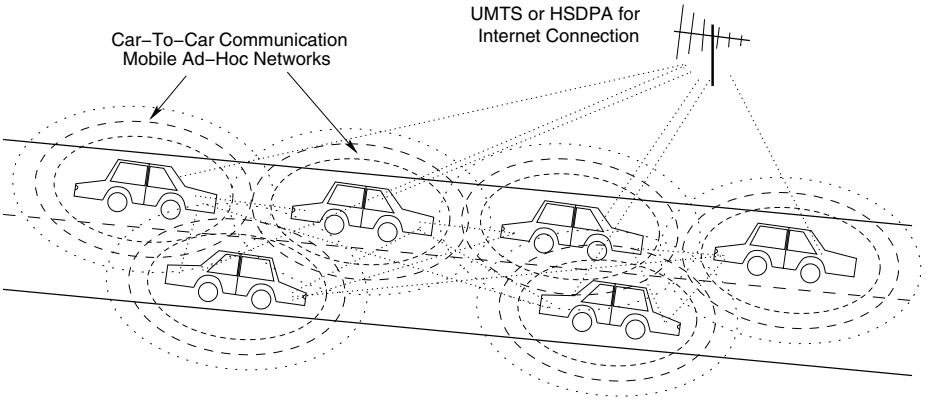


Fig. 2. Car Communication Scenario

The Internet Protocol is the enabling protocol with a lot of years of experience from the IT. Thereby, it is suggested, that this protocol is used in automotive for its local short range and long range communication (maybe except real time applications like engine management system or airbag systems). In particular, the following examples introduce the risk assessment of security and its derived safety aspects by connecting cars to each other and with the internet via IP. Thereby, we assume, that both types of attackers Θ_I and Θ_E exists and attack the car communication.

- The selected attack scenario (S1) is, for example, that we have a car-to-car communication between cars (*carA* and *carB*) and their drivers (Alice and Bob) *alice@carA.car-producer1.de* and *bob@carB.car-producer2.de*. In our example scenario, Alice drives in front of Bob and Bob wants to get ahead of Alice. Bob fakes the message "Slippery Road Ahead – Decrease Speed!" and sends it via car-to-car communication to Alice by spoofing (AS3) the source in that way, that the car of Alice "visualizes" the message is if it was sent by the *carX* in front of Alice. The spoofing attack changes the original source address

in such way, that sender *carB.car-producer2.de* is now *carX.car-producer3.com*. For Alice, it is hard to control the attack by verifying the sender without any security mechanisms.

- In this attack scenario (S2), Alice drives with her car's identification *alice@carA.car-producer1.de* in front of Bob (*bob@carB.car-producer2.de*). The car of Alice sends via car-to-car communication the current speed to Bob and all other cars behind Alice. This is useful to adjust the speed of all cars and to prevent rear-end collision accidents. The oncoming car *carX.car-producer3.com*, which can be a police car, converges to Alice. Thereby, *carX* read via AS1 the car-to-car communication and sniffs the current speed of Alice. If the police identifies, that Alice drives to fast, then she gets a ticket and she has to pay. For a mobile ad-hoc network, where the users are unknown, it is difficult to protect the network for unauthorized reading, which decreases the controllability.
- In attack scenario (S3), Alice wants to drive with the car. She opens the door and all car systems start the booting process. As introduced above, the car downloads, for example, the current security patches and navigation system information on the IT systems. A potential attacker Θ_E located in the Internet, performs a Man in the Middle attack (AS2) and captures the navigation request of Alice and send it to the navigation server. The response of the navigation server is also captured by the attacker. Without the protection of the confidentiality of the transmitted data, the attacker knows the route, which Alice wants to drive. Additional, the attacker can modify the route and Θ_E can send a much longer way and it is known, how long at least, Alice wants to drive the car. With this information, the attacker could, for example, rob the house of Alice, anger Alice by sending her to a traffic jam or increase the required gasoline of the car of Alice or kidnap her, because the exact driving route is known. Note, that the controllability of such an scenario should be given, which requires the assumption and defense of potential attackers.
- Alice drives with its car *alice@carA.car-producer1.de* in front ob Bob *bob@carB.car-producer2.de* in our exemplary attack scenario S4. Bob drives in front of Chris *chris@carC.car-producer3.de*. All cars communicate via car-to-car communication. Sensors of *carA* identify, that there is road hole on the road. To avoid a damage of the cars behind Alice, the car of Alice sends the Information "Attention - Road Hole Ahead!" to *carB* from Bob. The car of Bob receives the message and interrupts the car-to-car communication (A3) by not distributing this message to cars behind him. Thereby, *carC.car-producer3.de* does not receive the message and Chris cannot be getting the warning. Depending on the road hole, the car of Chris drives into the hole and gets damage. Bob, which is the attacker in this scenario, was motivated by malicious joy. For Chris is it difficult to control the attack, because he does not know that the communication is interrupted.
- In our attack scenario S5, a potential attacker can read/sniff the car initialization process of Alice car, whereby the car sends all requested information to, for example, the car insurance via Internet connection. The captured IP packets could be resend with a replay attack (A4), to enable another car with the insurance of Alice. Security mechanisms are strongly recommended to prevent such an attack. Otherwise, the controllability decreases.

- If an attacker wants, for example in our attack scenario S6, to perform a replay attack, then it is maybe useful, if the original source (for example, the IP address of the car of Alice) is not available and disconnected from Internet and/or the mobile ad-hoc networks. If this car is driving or has a permanent connection to the Internet, then an attacker would be disabling and disconnecting the original source with a denial of service attack (AS5). Thereby, the attacker could flood the IP address of the car of Alice (*carA.car-producer1.de*) and/or provided services of it. After disabling the Internet or mobile ad-hoc network connection, the attacker can perform the replay or spoofing attack with a low delectability and controllability.
- Scenario S7: If the cars have an IP addresses and if they are connected to each other via mobile ad-hoc networks and the Internet, then an attacker could send malformed packets (AS6), to crash a provided service or the car connection itself. If a design or implementation error exists, then the addressed service can crash. The result can be that this part of the car software or the addressed application does not work anymore and mostly only a reboot can solve the occurred problem. Note, that many car components cannot be rebooted during a tour on a highway. Additionally to control such an attack is difficult, because one specific IP packet might perform this attack.
- In our attack scenario S8, an attacker can use different existing distribution channels for malicious code known from the IT [6]. Thereby, the attacker can send the malicious code (AS7) via email or other communication channels to the car driver, who runs the code. Or the malicious code is injected into the car IT system by exploiting a specific service provided by a car application. If the malicious code runs on one or more car components, then, in the worst case, the attacker can control many car functions. Thereby, the route of navigation system or the user specific control setting can be changed. Additionally, if the malicious code focuses on the electronic throttle control, then the attacker can remotely increase or decrease the speed of the car, which implies and violates safety aspects. Note: if malicious code runs inside of the car IT, then the car IT is out of control of the owner.
- Scenario S9: One of the well known and not IP based attack focuses on the humans by social engineering them to get unauthorized information like a password. It is distinguish between computer and human based social engineering. By computer based social engineering, the attacker uses computers with their potentials like, for example, sending an email where the users open or run the attachment. If it is human based, then the attacker talks with the user by playing a boss or having an order from the boss. If the social engineering attacks success, then, depending on the gained information, the attacker has full or partly access to the system and control the complete information gain (A1-A5).

Note, that the introduced hypothetical scenarios are exemplary selected to show and motivate the necessity of security protection mechanisms. In addition, these scenarios can be enhanced by other known attacking techniques [4] or combined together to be more efficient for the potential attacker to address one or more selected security aspects. The following Table 2 summarizes the exemplarily selected and simulated attack scenarios and discusses their impact on the five security aspects.

Table 2. Security Analysis of the Hypothetical Attack Scenarios

	Confidentiality	Integrity	Authenticity	Non-Repudiability	Availability
S1	Is not violated, because the information is public for all.	The retrieved message is not proved and therefore unknown, if it is changed.	Is not verified and unknown, who was the sender of the message.	Is not given.	Is not addressed.
S2	Is violated and private or personal data should be kept secret or protected.	It is unknown, whether the retrieved speed is correct or not.	It is not proven, whether Alice was the sender of the message or not.	Alice can dispute, that she drove too fast and it is not provable.	Is not addressed.
S3	Is not given, because the attacker reads all transmitted data.	Is violated, because the attacker modified the response.	Is not given, because Alice seems to get the data from her insurance.	It is hard to prove that Alice sent the retrieved data.	Is not addressed.
S4	Is not addressed, because the message is public.	Is not proved and can be changed by anybody.	Is not addressed and it is unknown, whether the message was send by Alice or not.	It is hard to prove that Alice sent the retrieved data.	Is not addressed.
S5	Is important to protect the owner for abuse its information.	Should be proved, to guarantee to required duration time.	Is very important to charge the right driver.	Must be given, to prove the required car insurance.	Is not addressed.
S6	Is not addressed.	Is not addressed.	Is not addressed.	Is not addressed.	Whether violated and Alice cannot use the provided services.
S7	Is not addressed.	Is not addressed.	It is hard to verify the right sender whether the malformed IP packet.	Is not addressed.	It violated and Alice cannot use the provided services.
	Confidentiality	**Integrity**	**Authenticity**	**Non-Repudiability**	**Availability**
S8	If the malicious code runs on the car IT, then this security aspect is broken and the attacker can read everything.	If the malicious code runs on the car IT, then this security aspect is broken and the attacker can modify everything.	If the malicious code runs on the car IT, then this security aspect is broken and the attacker can change the source and destination IPs.	If the malicious code runs on the car IT, then this security aspect is broken.	Depending on the payload function, services can be disabled.
S9	If the attack is successful, then this security aspect is broken by telling the attacker the requested information.	If the attack is successful, then this security aspect is broken by changing information.	If the attack is successful, then this security aspect is broken because of the attack itself.	If the attack is successful, then this security aspect is broken by the user doing something for the attacker.	If the user disables services called upon by the attacker, then it is violated.

Table 3. Safety Analysis of the Hypothetical Attack Scenarios

	SIL Association
S1	It is difficult to control S1 and without any security mechanisms, the detection of such an attack is difficult. Alice seems to be decreasing the speed. Because the integrity and authenticity of the message was not proven. Therefore, the controllability is associated to the SIL level 3.
S2	The sniffing of information in a wireless mobile ad-hoc network is easy and therefore, it is not controllable, who receives and reads the information. This is the reason, why this attack scenario is associated with the highest SIL level 5.
S3	The communication between Alice and her insurance can be protected with existing cryptographic mechanisms. Therefore, existing mechanisms are currently available to control the access by unauthorized reading, writing and modifying of the information. If the security mechanisms are implemented and configured, then we can associate this attack scenario to the SIL level 1.
S4	If this attack is successful, then Chris does not know that the sent message is interrupted by an attacker. Therefore, it is difficult for him, to control such an attack which is the reason that this attack scenario is assigned to the SIL level 4.
S5	Currently, there are different mechanisms available (like timestamp or unique packet ID) to identify a replay attack. If such mechanisms are designed, implemented and configured for this car scenario, then the attack fails. Otherwise, the attacker can perform this attack successful. It means that security mechanisms must be designed and implemented in car environment to prevent such an attack. Therefore, the SIL level is assigned to level 1.
S6	To control and prevent a denial of service attack based on flooding is difficult, if the attacker has at least a higher bandwidth. Furthermore, the source of the attacker is not known always to block and control him. The association of this attack scenario to the SIL level is 2.
S7	This attack scenario is successful, if an implementation error exists, which is used by the attacker. If the attacker knows it, then it is hard to control it, because only one IP packed is required for the attack itself. Therefore, it is hard to control and associated to the SIL level 3.
S8	This form of an attack provides the attacker many possibilities, which reach up deactivate existing security mechanisms and to remote control the infected system. If this occurs, then the system is hard to be controlled by the owner. This is the reason, why it is associated to the SIL level 4.
S9	This form of attack does not focus on the IT system itself. Therefore, it is hard for technical security mechanisms to prevent such an attack. The human is the weak point of the system and if the human is social engineered, then the car IT system cannot be only controlled by the owner. Therefore, this attack scenario is associated to the SIL level 4.

The introduced hypothetical attack scenarios do not only violate the security aspects. For cars, the safety and its implication on human health is important and discussed in the following. Therefore, the safety integrity levels are used with the attack scenarios and presented in Table 3. Note that we only focus on the controllability of the threat. The failure rate depends on the potential attacker and not on randomly occurred events. Furthermore, the main affected safety violates are discussed to introduce the controllability of them and possible combinations are neglected.

Note, that the safety levels are defined for potential attackers. Thereby, for 3 attacks, we defined the highest SIL level, because the owner does not control the car IT system anymore. Furthermore, exiting definitions of safety levels should be redefined for cars, because of other environments, requirements and its mobility. Note also, that the possibility to violate humans is much easier with cars, than with classical IT systems and networks.

From the introduced attacker scenarios for cars IT systems derived from classical IT systems and networks, the risk and threat increased and provides attacker more points of attacks. Because of it, we want to motivate to investigate to focus more on security aspects for common design criteria's for car components and car IT systems. Furthermore, the designed security mechanisms should be implemented and configured for car IT systems to protect them from potential attackers.

4 Practical Risk Assessments on a Selected Example

In this section we present one first simulation for the potential attack S8 on the IT security of an automotive system, making use of malicious code (AS7) that employs the attacking strategies introduced in chapter 2, namely, the basic attack principles A1 (read) and A4 (create/spoof) as well as the attack scenario AS4 (replay attack). Again, the violated security aspects are pointed out. Thereby, we present potential attacks on today's cars using the CAN bus and we simulate the scenario for S8. Note that cars produced today will be still found on roads in 15 years. An upgrade in a car service station in, for example, 15 years could add many car functions (like car-2-car communication and so on) by adding the IP protocol based communication. These features can increase the security threats as introduced as follows.

In our scenario S8, the attacker can attack the five security aspects, summarized within Table 4 using the basic attack principles A1 and A4 as well as the attack scenario AS4.

Table 4. Scenario S8 with its Basic Attack Principles and Attack Scenario Addressing the Security Aspects

	Confidentiality	Integrity	Authenticity	Non-Repudiability	Availability
S8	A1	A4	A4	AS4	A4

As basis for our simulation to study potential CAN bus attacks for A1, A4 and AS4 to perform S8 we use the software product CANoe from Vector Informatik [8], which is widely used throughout the automotive industry for development, testing and diagnosis of embedded automotive systems. It provides hardware interfaces to common automotive bus systems (e.g. CAN, LIN, MOST, FlexRay), which allows bus traffic analysis and the coupling of simulated and physical devices.

The standard installation provides a few automotive demo-environments, which represent a subset of a simplified car's automotive components. For our test setup we chose a demo environment that consists of two CAN networks (powertrain and comfort subnets). Both are connected together via a gateway that passes some messages from one subnet into the other, e.g. to provide the main display in the comfort bus with its needed input data from the powertrain subnet.

As exemplary target for the simulated basic attacks A1 and A4 we chose the electric window lift triggered by a corresponding switch within a console to perform the attack scenario AS4, which is used for our scenario S8.

Under normal operation, the ECU for these switches periodically (every 20 milliseconds) sends its status onto the comfort bus in form of a 32 bit data telegram,

which contains the position of the window switches as bit flags. For the driver window, "open" and "close" are represented as bit 26 and 27 (the other bits are responsible e.g. for adjusting the horizontal and vertical position of each mirror). If the responsible electronic control unit receives such a data telegram from the console (Message-ID 416) it reacts corresponding to these bit flags contained and trigger the window-lift motor according to the direction specified by the received data telegram from the console. For example, a data telegram sent while pushing the "open" switch for the driver window has the 26th bit set and pressing the close button results in a set bit in position 27 of a 32 bit datagram.

Based on this test setup, we conceived and simulated S8 as an attack on the electric window lift. To infiltrate malicious code (AS7) the attacker prepares a manipulated email attachment and transmits this email via IP based car-to-infrastructure (C2I) communication to Alice (*alice@carA.producer1.de*). By convincing Alice to read the email and execute the attachment by using known computer based social engineering approaches or by exploiting some security vulnerabilities in the ECU software the malicious code gets started in the automotive IT system. If the infected ECU is also connected to the internal CAN bus or has even gateway functionality, the malicious code from the email can also gain access to internal bus systems of the car, which might still be today's automotive bus systems like CAN in the old cars of tomorrow.
To achieve the aim of this attack, the attacker combines code for A1 and A4 in two parts of the attack:

Part 1 (A1): The malicious code (AS7) reads/sniffs (A1) messages (Message-ID 273) from the internal car communication containing information about the current speed of *carA.producer1.de*. The sniffing is repeated until some condition (in this case: the car exceeds 200 km/h) is fulfilled, which leads to activation of second part (A4, see below). During this first step, this and other sniffed sensitive information could also be transmitted back to the attacker via C2C/C2I IP communication.
Part 2 (A4): In the second part of the payload function, the attack scenario A4 is performed by creating and spoofing the data telegram for opening the driver's window in a loop. The content of the data telegrams to be generated is hard coded in the malicious code and might got acquired by the attacker prior the attack by reading (A1) such a message from another car of that series or by looking it up in a specification.

Combining A1 (read) as well as A4 (create/spoof) represents a replay attack (AS4).

By enforcing the opening of the driver window at high speed of 200 km/h, the attacker might have the intention to harm or at least frighten Alice. Obviously she would get distracted a lot which poses clear safety implications (at least SIL 1).

In our simulation (implemented by adding 16 additional lines of CAPL code, a C-like programming language within the CANoe environment) we found that during this replay attack the enforced opening of the window cannot even be stopped by manual counter steering via pressing the close-button.

By using another logical starting condition like a timer for example, the attacker might also intend to steal items out of Alice's car (carA.producer1.de), for example, parts of the entertainment system, by opening the window at an appropriate point in time, for example, when her car is usually placed at a public parking lot.

With respect to the violated security aspects, the practical implementation of S8 can be classified as shown in Table 5.

Table 5. Security Analysis of the Exemplary Practical Attack Scenario

	Confidentiality	Integrity	Authenticity	Non-Repudiability	Availability
S8	The code reads out the current speed which is sensitive information telling about the drivers habits (A1)	The integrity of the replayed messages is not violated but the integrity of the entire system is changed by injecting packets. (A4)	The authenticity of the replayed messages is violated since they do not stem from the door's console (A4)	After the spoofing attack, it is hard to prove for Alice, that the window opened without her trigger (AS4)	The availability of the real window switch is not given while it is blocked (A4)

Using the described combination of the basic attacks read (A1) and create/spoof (A4) which forms a replay attack (AS4), we simulated the attack scenario S8 on the given environment using existing automotive development and simulation software. The results of this first simulated attack show that there is an increasing importance of requirements and measures for IT security in embedded automotive application domains.

5 Conclusion

In this paper, we summarize existing security aspects and safety levels for a hypothetical application scenario the IP-enabled car. Based on Internet Protocol (IP) based communication, selected threats and vulnerabilities are introduced and discussed on example attack scenarios. Derived from classical IP based attacks, the assumption is done to map them to car IT systems in near future. Therefore, hypothetical risk assessments on selected examples are introduced and their implication on the security and safety are discussed. Our goal for this paper is the simulation of the potential attack behavior for Internet and mobile ad-hoc network enabled automotive protocol attack performance and the impact to security and safety is investigated for selected examples. If the IP protocol is considered in future automotive technology, our work should motivate to design, implement and configure methods of defense for future cars IT systems from the beginning of the developing process. To point out the need for such IT security measures in tomorrows automotive systems, we have simulated an example attack scenario practically on today's technology using a wide spread automotive development and simulation software.

Acknowledgments. The security and safety analysis and the exemplary selected, attacker scenarios of this paper are supported by the Automotive project at Otto-von-Guericke University of Magdeburg (http://www.uni-magdeburg.de/automotive/).

References

1. Howard, J.D., Longstaff, T.A.: A Common Language for Computer Security Incidents (SAND98-8667); Sandia National Laboratories. 1998 (0-201-63346-9), Forschungsbericht (1998)
2. Dittmann, J.: Course: Fundamentals of IT-Security, Summer Term 2005 (2005)

3. Innovation Electronics (UK) Ltd., Health & Safety Laboratory: A methodology for the assignment of safety integrity levels (SILs) to safety-related control functions implemented by safety-related electrical, electronic and programmable electronic control systems of machines; Health and Safety Laboratory. Forschungsbericht (2004)
4. Anderson, R.: Security Engineering. In: A Guide to Building Dependable Distributed Systems.: A Guide to Building Dependable Distributed System, Wiley, New York (2001)
5. Tulloch, M.: Microsoft Encyclopedia of Security. Microsoft Press (2003) ISBN 0-7356-1877-1
6. Kiltz, S., Lang, A., Dittmann, J.: Klassifizierung der Eigenschaften von Trojanischen Pferden. In: Horster, P. (ed.) DACH Security 2006; Bestandsaufnahme, Konzepte, Anwendungen, Perspektiven; Syssec, Duesseldorf, Germany, pp. S. 351–361 (2006) ISBN: 3-00-018166-0
7. Kaufman, C., Perlman, R., Speciner, M.: Network security: private communication in a public world. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2002) ISBN 0-13-04601-9
8. Vector [CANoe und DENoe]; (June 2007), http://www.vector-informatik.com/vi_canoe_ de,2816.html

Modelling Interdependencies Between the Electricity and Information Infrastructures

Jean-Claude Laprie, Karama Kanoun, and Mohamed Kaâniche

LAAS-CNRS, Université de Toulouse, France
{laprie,kanoun,kaaniche}@laas.fr

Abstract. The aim of this paper is to provide qualitative models characterizing interdependencies related failures of two critical infrastructures: the electricity infrastructure and the associated information infrastructure. The interdependencies of these two infrastructures are increasing due to a growing connection of the power grid networks to the global information infrastructure, as a consequence of market deregulation and opening. These interdependencies increase the risk of failures. We focus on cascading, escalating and common-cause failures, which correspond to the main causes of failures due to interdependencies. We address failures in the electricity infrastructure, in combination with accidental failures in the information infrastructure, then we show briefly how malicious attacks in the information infrastructure can be addressed.

1 Introduction

In the past decades, the electric power grid experienced several severe failures that affected the power supply to millions of customers. The most recent one occurred in November 2006 in Western Europe when a shutdown of a high-voltage line in Germany resulted in massive power failures in France and Italy as well as in parts of Spain, Portugal, the Netherlands, Belgium and Austria, and even extended as far as Morocco. About ten million customers were affected by this failure. Similar major blackouts with even more severe consequences have occurred in summer 2003 in the United States, in Canada and in Italy [1, 2]. These events highlight the vulnerability of the electric grid infrastructures and their interdependencies. The large geographic extension of power failures effects is related to i) the high interconnectivity of power grid transmission and distribution infrastructures and to ii) the multiple interdependencies existing between these infrastructures and the information infrastructures supporting the control, the monitoring, the maintenance and the exploitation of power supply systems. An *interdependency* is a bidirectional relationship between two infrastructures through which the state of each infrastructure influences or is correlated to the state of the other. Clearly there is a need to analyze and model critical infrastructures in the presence of interdependencies in order to understand i) how such interdependencies may contribute to the occurrence of large outages and ii) how to reduce their impact.

This paper focuses on two interdependent infrastructures: the electric power infrastructure and the information infrastructures supporting management, control and

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 54–67, 2007.

maintenance functionality. More specifically, it addresses modelling and analysis of interdependency-related failures between these infrastructures.

We concentrate on cascading, escalating and common-cause failures, which correspond to the main causes of interdependency-related failures. We model the infrastructures globally, not explicitly modelling their components. The models presented are qualitative ones. They describe scenarios that are likely to take place when failures occur. The models are built based on assumptions related to the behaviour of the infrastructures as resulting from their mutual interdependencies.

In the remainder of this paper, we will first address failures in the electricity infrastructure and accidental failures in the information infrastructure, considering the three classes of interdependencies, then we will illustrate briefly how malicious attacks of information infrastructures can be addressed.

Section 2 presents the background and related work. Sections 3 and 4 are dedicated to modelling interdependencies taking into account failures in the information infrastructure and in the electricity infrastructure. Section 3 addresses accidental failures in the information infrastructure while Section 4 addresses malicious attacks. Section 5 concludes the paper.

2 Background and Related Work

Interdependencies increase the vulnerability of the corresponding infrastructures as they give rise to multiple error propagation channels from one infrastructure to another that make them more prone to exposure to accidental as well as to malicious threats. Consequently the impact of infrastructure components failures and their severity can be exacerbated and are generally much higher and more difficult to foresee, compared to failures confined to single infrastructures. As an example, most major power grid blackouts that have occurred in the past were initiated by a single event (or multiple related events such as a power grid equipment failure that is not properly handled by the SCADA, i.e., Supervisory Control And Data Acquisition, system) that gradually leads to cascading failures and eventual collapse of the entire system [2].

Infrastructure interdependencies can be categorized according to various dimensions in order to facilitate their identification, understanding and analysis. Six dimensions have been identified in [3]. They correspond to: a) the type of interdependencies (physical, cyber, geographic, and logical), b) the infrastructure environment (technical, business, political, legal, etc.), c) the couplings among the infrastructures and their effects on their response behaviour (loose or tight, inflexible or adaptive), d) the infrastructure characteristics (organisational, operational, temporal, spatial), e) the state of operation (normal, stressed, emergency, repair), the degree to which the infrastructures are coupled, f) the type of failure affecting the infrastructures (common-cause, cascading, escalating). Other classifications have also been proposed in [4-6, 29]. In particular, the study reported in [29], based on 12 years public domain failure data, provides useful insights about the sources of failures affecting critical infrastructures, their propagation and their impact on public life, considering in particular the interdependencies between the communication and information technology infrastructure and other critical infrastructures such as electricity, transportation, financial services, etc.

Referring to the classification of [3], our work addresses the three types of failures that are of particular interest when analyzing interdependent infrastructures: i) cascading failures, ii) escalating failures, and iii) common cause failures. Definitions are as follows:

- *Cascading failures* occur when a failure in one infrastructure causes the failure of one or more component(s) in a second infrastructure.
- *Escalating failures* occur when an existing failure in one infrastructure exacerbates an independent failure in another infrastructure, increasing its severity or the time for recovery and restoration from this failure.
- *Common cause failures* occur when two or more infrastructures are affected simultaneously because of some common cause.

It is noteworthy that these classes of failures are not independent; e.g., common-cause failures can cause cascading failures [7].

Among the three relevant types of failures in interdependent infrastructures, the modelling of cascading failures has received increasing interest in the past years, in particular after the large blackouts of electric power transmission systems in 1996 and 2003. Several research papers and modelling studies have been published on this topic in particular by the Consortium for Electric Reliability Technology Solutions (CERTS) in the United-States [8]. A large literature has been dedicated recently to the elaboration of analytic or simulation based models that are able to capture the dynamics of cascading failures and blackouts. A brief review of related research addressing this topic is given hereafter. A more detailed state-of-the art can be found in [9, 10].

In [11, 12], the authors present an idealised probabilistic model of cascading failures called CASCADE that is simple enough to be analytically tractable. It describes a general cascading process in which component failures weaken and further load the system so that components failures are more likely. This model describes a finite number of identical components that fail when their loads exceed a threshold. As components fail, the system becomes more loaded, since an amount of load is transferred to the other components, and cascading failures of further components become likely. This cascade model and variants of it have been approximated in [13-15] by a Galton-Watson branching process in which failures occur in stages, with each failure giving rise to a Poisson distribution of failures at the next stage.

The models mentioned above do not take into account the characteristics of power systems. An example of a cascading failures model for a power transmission system is discussed in [16]. The proposed model represents transmission lines, loads, generators and the operating limits on these components. Blackout cascades are essentially instantaneous events due to dynamical redistribution of power flows and are triggered by probabilistic failures of overloaded lines. In [17] a simulation model is proposed to calculate the expected cost of failures, taking into account time-dependent phenomena such a cascade tripping of elements due to overloads, malfunction of the protection system, potential power system instabilities and weather conditions. Other examples emphasizing different aspects of the problem have been proposed e.g., in [18-21], in which hidden failures of the protection system are represented. Their approach uses a probabilistic model to simulate the incorrect tripping of lines and generators due to hidden failures of line or generator protection systems. The distribution of power system blackout size is obtained using importance sampling and Monte-Carlo simulation.

Recently, new approaches using complex networks theory have been also proposed for modelling cascading failures [22-27]. These models are based on the analysis of the topology of the network characterizing the system and the evaluation of the resilience of the network to the removal of nodes and arcs, due either to random failures or to malicious attacks).

All the models discussed above adopt a simplified representation of the power system, assuming that the overloading of system components eventually leads to the collapse of the global system. However, these models do not take into account explicitly the complex interactions and interdependencies between the power infrastructure and the ICT infrastructures. Moreover, the modelling of escalating failures is not addressed. Further work is needed in these directions. This paper presents a preliminary attempt at filling such gaps.

3 Accidental Failures in the Information Infrastructure

Our aim is to model the infrastructures behaviour together, when taking into account the impact of accidental failures in the information infrastructure and failures in the electricity infrastructure, as well as their effects on both infrastructures. Modelling is carried out progressively:

- First, we model cascading failures by analysing the constraints one infrastructure puts on the other one, assuming that the latter was in a working state when an event occurs in the other one.
- Then, we address cascading and escalating failures considering successively:
  - constraints of the information infrastructure on the electricity infrastructure,
  - constraints both ways (of the information infrastructure on the electricity infrastructure and of the electricity infrastructure on the information infrastructure).
- Finally, we address common-cause failures.

For the sake of clarity, and in order to avoid any confusion between the two infrastructures, we use specialized but similar terms for the two infrastructures states and events as indicated by Table 1.

Table 1. States and events of the infrastructures

Information Infrastructure	Electricity Infrastructure
i-failure	e-failure
i-restoration	e-restoration
i-working	e-working
Partial i-outage	Partial e-outage, e-lost
i-weakened	e-weakened

3.1 Modelling Cascading Failures

We first analyse the impact of accidental i-failures on each infrastructure assuming that the electricity infrastructure is in an e-working state, then we analyse the impact

of e-failures on each infrastructure assuming that the and information infrastructure is in an i-working state, before considering the combined impact of i- and e-failures in Section 3.2.

3.1.1 Impact of Information Infrastructure Failures (i-failures)

Accidental i-fa-ilures, hardware- or software-induced, affecting the information infrastructure can be:

- Masked (unsignalled) i-failures, leading to latent errors.
- Signalled i-failures.

Latent errors can be:

- Passive (i.e., without any action on the electricity infrastructure), but keeping the operators uninformed of possible disruptions occurring in the electricity infrastructure.
- Active, provoking undue configuration changes in the electricity infrastructure.

After signalled i-failures, the information infrastructure is in a partial i-outage state: the variety of functions and components of the information infrastructure, and its essential character of large network make unlikely total outage. Latent errors can accumulate. Signalled i-failures may take place when the information infrastructure is in latent error states. When the information infrastructure is in a partial i-outage state, i-restoration is necessary to bring it back to an i-working state.

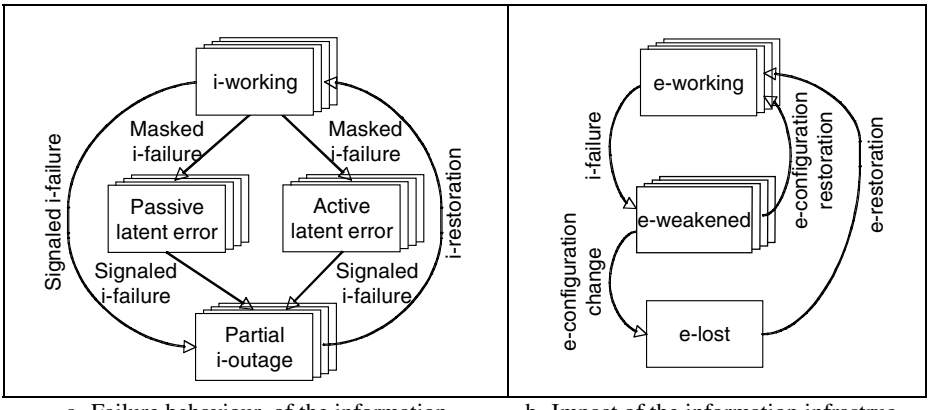
Fig. 1-a gives the state machine model of the information infrastructure taking into account its own failures. It is noteworthy that all states are presented by several boxes, meaning that a state corresponds in reality to a group of different states that are considered as equivalent with respect to the classification given in Table 1. For example all states with only one busbar isolated can be considered as equivalent irrespective of which busbar is isolated.

We assume that an i-failure puts some constraints on the electricity infrastructure (i.e., cascading failure), leading to a weakened electricity infrastructure (e.g., with a lower performance, unduly isolations, or unnecessary off-line trips of production plants or of transmission lines).

From an e-weakened state, a configuration restoration leads the electricity infrastructure back into a working state, because no e-failures occurred in the electricity infrastructure. Accumulation of untimely configuration changes, may lead to e-lost state (i.e., a blackout state), from which an e-restoration is required to bring back the electricity infrastructure into an e-working state. Fig. 1-b shows the constraint that the information infrastructure puts on the electricity infrastructure when the latter is in an e-working state.

3.1.2 Impact of Electricity Infrastructure Failures (e-failures)

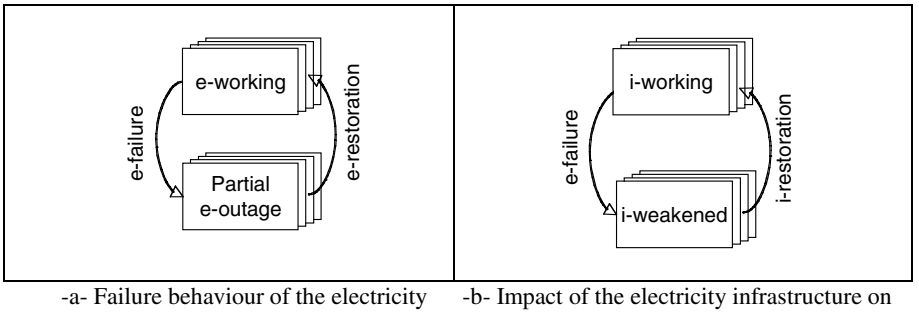
We consider that the occurrence of e-failures leads the electricity infrastructure to be in a partial e-outage state, unless propagation within the infrastructure leads to loosing its control (e.g., a blackout of the power grid), because of an i-failure (this latter case corresponds to escalating events that will be covered in the next section). Fig. 2-a gives the state machine model of the electricity infrastructure taking into account its own failures.



-a- Failure behaviour of the information infrastructure

-b- Impact of the information infrastructure on the electricity infrastructure

Fig. 1. Impact of i-failures on infrastructures behaviour



-a- Failure behaviour of the electricity infrastructure

-b- Impact of the electricity infrastructure on the information infrastructure

Fig. 2. Impact of e-failures on infrastructures behaviour

Also e-failures may lead the information infrastructure to an i-weakened state in which parts of the information infrastructure can no longer implement their functions, although they are not failed, due to constraints originating from the failure of the electricity infrastructure. Fig. 2-b shows the constraint that the electricity infrastructure puts on the information infrastructure assuming that the latter is in an i-working state.

Tables 2 and 3 summarise the states and events of each infrastructure, taking into account cascading events, as described above.

3.2 Modelling Cascading and Escalating Failures

The global state machine model of the two infrastructures is built progressively:

- Considering, in a first step, only the constraints of the information infrastructure on the electricity infrastructure.
- Considering constraints of each infrastructure on the other.

Table 2. States and events of the information infrastructure

Events	
Signalled i-failure	Detected i-failure
Masked i-failure	Undetected i-failure
i-restotation	Action for bringing back the information infrastructure in its normal functioning after i-failure(s) occurred
States	
i-working	The information infrastructure ensures normal control of the electricity infrastructure
Passive latent error	Parts of the information infrastructure have an i-failure, which prevents monitoring of the electricity infrastructure: e-failures may remain unnoticed
Active latent error	Parts of the information infrastructure have an i-failure, that may lead to unnecessary, and unnoticed configuration changes
Partial i-outage	Parts of the information infrastructure have knowingly an i-failure. Partial i-outage is assumed**:** the variety of functions and of the components of the infrastructure, and its essential character of large network make unlikely total outage
i-weakened	Parts of the information infrastructure can no longer implement their functions, although they do not have an e-failure, due to constraints originating from e-failures, e.g., shortage of electricity supply of unprotected parts.

Table 3. States and events of the electricity infrastructure

Events	
e-failure	Malfunctioning of elements of the power grid: production plants, transformers, transmission lines, breakers, etc.
e-restoration	Actions for bringing back the electricity infrastructure in its normal functioning after e-failure(s) occurred. Typically, e-restoration is a sequence of configuration change(s), repair(s), configuration restoration(s)
e-configuration change	Change of configuration of the power grid that are not immediate consequences of e-failures, e.g., off-line trips of production plants or of transmission lines
e-configuration restoration	Act of bringing back the electricity infrastructure in its initial configuration, when configuration changes have taken place
States	
e-working	Electricity production, transmission and distribution are ensured in normal conditions
Partial e-outage	Due to e-failure(s), electricity production, transmission and distribution are no longer ensured in normal conditions, they are however somehow ensured, in degraded conditions
e-lost	Propagation of e-failures within the electricity infrastructure led to loosing its control, i.e., a blackout occurred.
e-weakened	Electricity production, transmission and distribution are no longer ensured in normal conditions, due to i-failure(s) of the information infrastructure that constrain the functioning of the electricity infrastructure, although no e-failure occurred in the latter. The capability of the electricity infrastructure is degraded: lower performance, configuration changes, possible manual control, etc.

Fig. 3 gives a state machine model of the infrastructures, taking into account, only the constraints of the information infrastructure on the electricity infrastructure. The states are described in terms of the statuses of both infrastructures. Both cascading failures (states 3, 4) and escalating ones are evidenced, with a distinction of consequences of the latter in terms of time to restoration (state 6) and of severity (state 7). Dependency of the electricity infrastructure upon the information infrastructure is illustrated by the need for both i- and e-restoration from states 6 and 7.

A noteworthy example of transitions from states 1 to 2, and from 2 to 7 relates to the August 2003 blackout in the USA and Canada: the failure of the monitoring software was one of the immediate causes of the blackout, as it prevented confining the electrical line incident, before its propagation across the power grid [1].

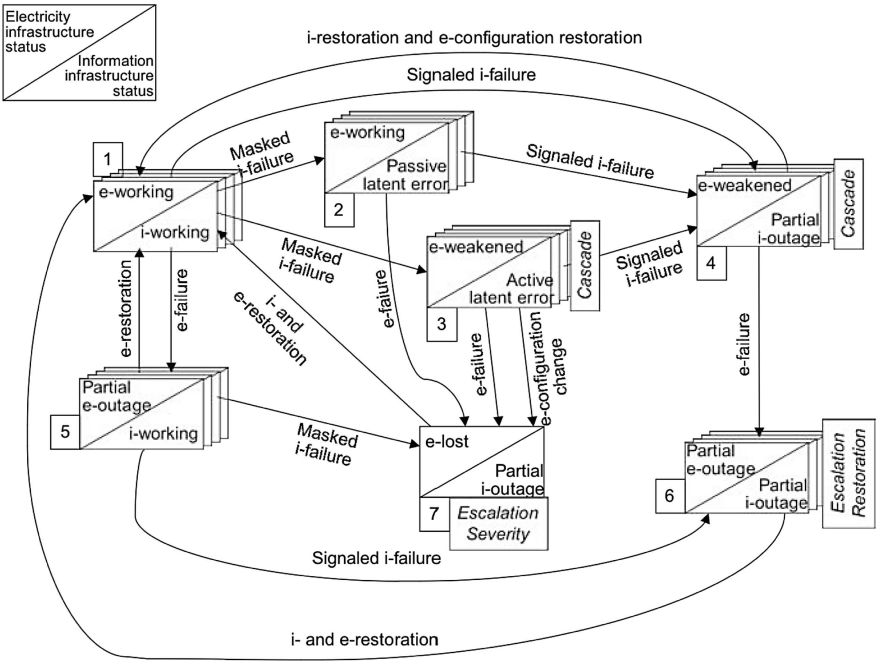


Fig. 3. State Machine taking into account constraints of the information infrastructure on the electricity infrastructure

A Petri net representation of the Fig. 3 model is given by Fig. 4 which enables to evidence the cascading and escalating mechanisms. Such mechanisms are, in Petri net terms, synchronizations between the individual events of the infrastructures. Table 4 gives the correspondence between the states and events of Figures 3 and 4.

This Petri net is deliberately kept simple. In particular, it does not distinguish the individual states within a group of states represented by several boxes in Fig. 3. For example, state 2 of Fig. 3 that represents in reality a set of states is represented by a single state in the Petri net of Fig. 4.

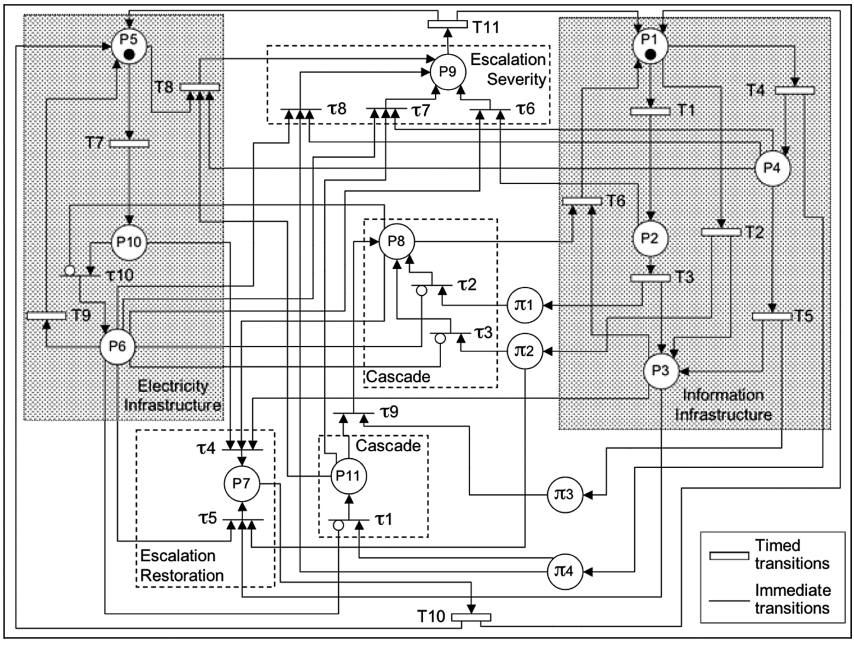


Fig. 4. Example of a high level Petri net associated to the model of Figure 3

Table 4. Correspondence between states and events of Fig. 3 and Fig. 4

States in State Machine	Markings in the Petri net
1	P1, P5
2	P2,P5
3	P4,P5,P11
4	P3,P5,P8
5	P1,P6
6	P7
7	P9

Transitions in State Machine	Transitions in the Petri net
1 → 2	T1
1 → 3	T4 - τ1
1 → 4	T2 - τ3
1 → 5	T7 - τ10
2 → 4	T3 - τ2
2 → 7	T7 - τ10 - τ6
3 → 4	T5 - τ9
3 → 7 - configuration change	T8
3 → 7 - e-failure	T7 - τ10 - τ7
4 → 1	T6
4 → 6	T7 - τ4
5 → 1	T9
5 → 6	T2 - τ5
5 → 7	T1 - τ6 or T4 - τ8
6 → 1	T10
7 → 1	T11

Fig. 5 gives a state machine model of the infrastructures, taking into account the constraints of the electricity infrastructure on the information infrastructure in addition to those of the information infrastructure on the electricity infrastructure already considered in Fig. 3. In addition, Fig. 5 assumes possible accumulation of e-failures from states 5 to 7 and from the escalation restoration state 6 to the escalation severity state 8.

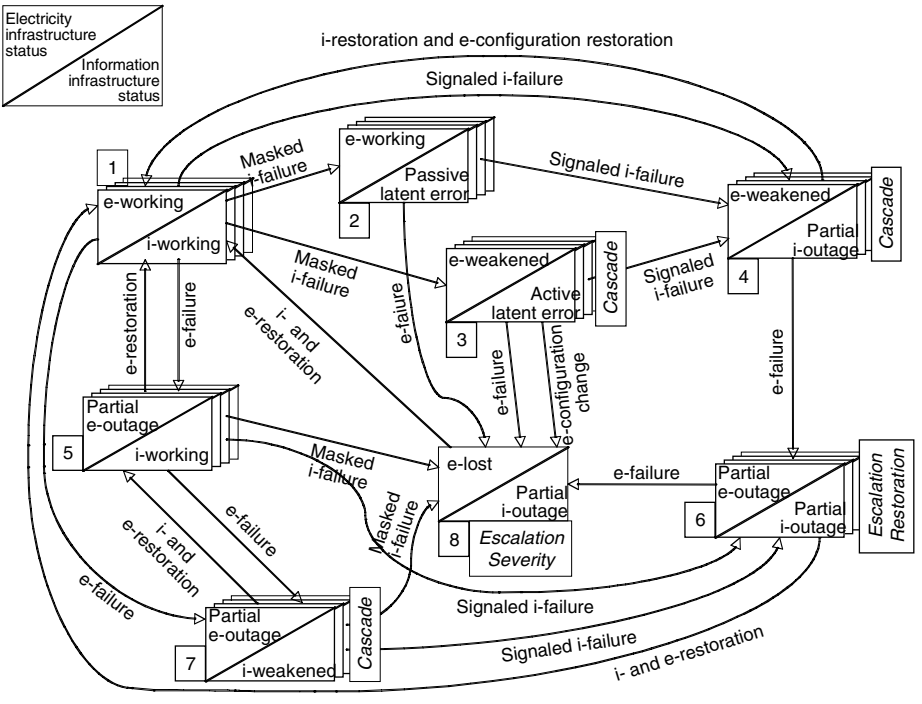


Fig. 5. Model of the two infrastructures when considering accidental failures

3.3 Modelling Common-Cause Failures

Figure 6 gives a model with respect to common-cause failures that would occur when the infrastructures are in normal operation, bringing the infrastructures into states 6 or 8 of Figure 5, i.e., to escalation. Should such failures occur in other states of the infrastructures of Figure 5 model, they would also lead to states 6 or 8.

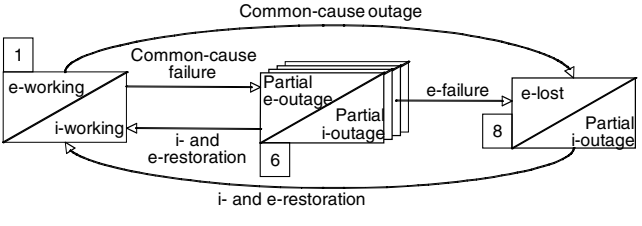


Fig. 6. Common-cause failures model

Considering common-cause failures does not introduce additional states, they however add direct transitions from already existing states that do not exist when considering only cascading and escalating failures. The states of the resulting model become almost totally interconnected.

4 Malicious Attacks of the Information Infrastructure

We consider malicious attacks of the information infrastructure and their consequences on the electricity infrastructure. Due to the very nature of attacks, a distinction has to be performed for both infrastructures between their real status and their apparent status. For the electricity infrastructure, the apparent status is as reported by the information infrastructure.

Attacks fall into two classes:

- Deceptive attacks that are provoking unperceived malfunctions, thus similar to the latent errors previously considered,
- Percep*tible attacks* creating detected damages.

Deceptive attacks can be:

- *Passive* (i.e., without any direct action on the electricity infrastructure).
- *Active*, provoking configuration changes in the electricity infrastructure.

Fig. 7 gives the state machine model of the infrastructures. This model and the previous one are syntactically identical: they differ by the semantics of the states and of the inter-state transitions. Let us consider for example states 2 and 3.

In state 2, the effects of the passive deceptive attack are: i) the information infrastructure looks like working while it is in a partial i-outage state due to the attack, ii) it informs wrongly the operator that the electricity infrastructure is in partial i-outage, and as consequence iii) the operator performs some configuration changes in the electricity infrastructure leading it to a i-weakened state. Accumulation of configuration changes by the operator may lead the electricity infrastructure into e-lost state.

In state 3, the effects of the active deceptive attack are: i) the information infrastructure looks like working while it is in a partial i-outage state due to the attack, ii) it performs some configuration changes in the electricity infrastructure leading it to a weakened state without informing the operator that the electricity infrastructure is in partial e-outage, for whom the electricity infrastructure appears if it were working. Accumulation of configuration changes by the information infrastructure may lead the electricity infrastructure into a e-lost state.

The difference between states 2 and 3 is that in state 2 the operator has made some actions on the electricity infrastructure and is aware of the e-weakened state, while in state 3 the operator is not aware of the actions performed by the information infrastructure on the electricity infrastructure.

After detection of the attack, the apparent states of the infrastructures become identical to the real ones (state 4), in which i-restoration and configuration restoration are necessary to bring back the infrastructures to their working states.

States 5, 6 and 7 are very similar respectively to states 5, 6 and 7 of Fig. 5, except that in state 6 the information infrastructure is in a partial i-outage state following a perceptive attack in Fig. 7 and following a signalled i-failure in Fig. 5.

In Fig. 7, state 8 corresponds to e-lost state but the operator is not aware, he/she has been informed wrongly by the partial i-outage of information infrastructure that it is in a partial e-outage state.

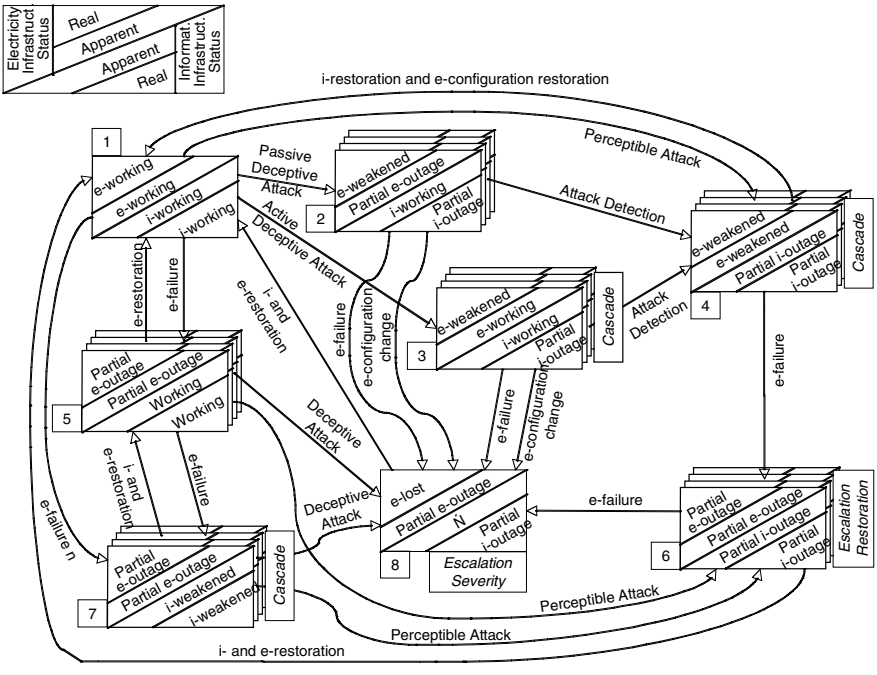


Fig. 7. Model of the two infrastructures when considering malicious attacks

5 Conclusion

In this paper we have introduced qualitative models allowing the description and the analysis of the behaviour of the information and electricity infrastructures taking into account the effect of failures of each infrastructure on the other one. These models describe, at a high level, scenarios that may occur when failures occur and the relationship between the states of the two infrastructures.

We have presented different models when considering accidental failures in the information infrastructure and when accounting for malicious attacks. Currently we are investigating a unified model for taking into account both classes of failures of the information infrastructure.

The high level models developed in this paper are to be refined to evaluate quantitative measures characterizing the impact of the interdependency-related failures on the resilience of the electricity and information infrastructures, with respect to the occurrence of critical outages and blackouts. In particular, in parallel to this work, within the CRUTIAL project, preliminary investigations have been carried out in order to develop a hierarchical modelling framework aimed at the detailed modelling of the electricity and information infrastructures taking into account their internal structure, and accidental failures [28]. When considering accidental failures, modelling techniques such as stochastic Petri nets or stochastic activity networks, complemented by data collected

from observation of the two infrastructures (or based on more general data, see e.g., [29]) are used. Consideration of malicious attacks raises some difficulties and challenges. Indeed, the very definition of security measures and evaluation has been, and is still, a research topic (see e.g., [30, 31]). Future work will be focussed on the definition of an integrated modelling approach that is well suited to take into account the combined impact of accidental as well as malicious faults.

Acknowledgement. This work is partially supported by the European project CRUTIAL (Critical Utility InfrastructurAL Resilience), IST-FP6-STREP – 027513.

References

[1] US-Canada, Power System Outage Task Force — Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations(2004)
[2] Pourbeik, P., Kundur, P.S., Taylor, C.W.: The Anatomy of a Power Grid Blackout. IEEE Power & Energy Magazineno, 22–29 (September/October 2006)
[3] Rinaldi, S.M., Peerenboom, J.P., Kelly, T.K.: Identifying, Understanding, and Ana-lyzing Critical Infrastructure Interdependencies. IEEE Control Systems Magazine, 11–25 (December 2001)
[4] Pederson, P., Dudenhoeffer, D., Hartley, S., Permann, M.: Critical Infrastructure Interdependency Modeling: A Survey of U.S. and International Research, Idaho National Lab, USA (2006)
[5] Lee, E., Mitchell, J., Wallace, W.: Assessing Vulnerability of Proposed Designs for Interdependent Infrastructure Systems. In: 37th Hawaii Int. Conf. on System Sciences (2004)
[6] Masera, M.: An Approach to the Understanding of Interdependencies. In: Masera, M. (ed.) Power Systems and Communications Infrastructures for the Future (CRIS'2002), Beijing, China (2002)
[7] Krings, A., Oman, P.: A simple GSPN for modeling common-mode failures in critical infrastructures. In: 36th Hawai Int. Conf. on System Sciences, p. 10 (2003)
[8] Dobson, I., Carreras, B.A.: Risk Analysis of Critical Loading and Blackouts with Cascading Events. In: Consortium for Electric Reliability Tech. Solutions (CERTS) (2005)
[9] Dobson, I., Carreras, B.A., Lynch, V.E., Newman, D.E.: Complex Systems Analysis of Series of Blackouts: Cascading Failure, Criticality, and Self-organization. In: IREP Symp. on Bulk Power Systems and Control - VI, Cortina d'Ampezzo, Italy, pp. 438–451 (2004)
[10] Anghel, M., Werly, K.A., Motter, A.E.: Stochastic Model for Power Grid Dynamics. In: 40th Hawaii International Conference on System Sciences, Big Island, Hawaii (2007)
[11] Dobson, I., Carreras, B.A., Lynch, V.E., Nkei, B., Newman, D.E.: A Loading-dependent Model of Probabilistic Cascading Failure. In: Probability in the Engineering and Informational Sciences, vol. 19, pp. 475–488 (2005)
[12] Dobson, I., Carreras, B., Newman, D.: A Probabilistic Loading-dependent Model of Cascading Failure and Possible Implications for Blackouts. In: 36th Hawaii Int. Conference on System Sciences (2003)
[13] Dobson, I., Carreras, B.A., Newman, D.E.: A Branching Process Approximation to Cascading Load-dependent System Failure. In: 37th Hawaii International Conference on System Sciences, Hawaii, IEEE Computer Society, Los Alamitos (2004)

[14] Dobson, I., Wierzbicki, K.R., Carreras, B.A., Lynch, V.E., Newman, D.E.: An Esti-mator of Propagation of Cascading Failure. In: 39th Hawaii International Conference on System Sciences, Kauai, Hawaii, IEEE Computer Society, Los Alamitos (2006)
[15] Dobson, I., Carreras, B.A., Newman, D.E.: Branching Process Models for the Exponentially Increasing Portions of Cascading Failure Blackouts. In: 38th Hawaii Interna-tional Conference on System Sciences, Hawaii, IEEE Computer Society, Los Alamitos (2005)
[16] Carreras, B.A., Lynch, N.E., Dobson, I., Newman, D.E.: Critical Points and Transi-tions in an Electric Power Transmission Model for Cascading Failure Blackouts. Chaos 12(4), 985–994 (2002)
[17] Rios, M.A., Kirschen, D.S., Jayaweera, D., Nedic, D.P., Allan, R.N.: Value of secu-rity: Modeling time- dependent phenomena and weather conditions. IEEE Transactions on Power Systems 17(3), 543–548 (2002)
[18] Chen, J., Thorp, J.D.: A Reliability Study of Transmission System Protection via a Hidden Failure DC Load Flow Model. In: IEEE Fifth International Conference on Power System Management and Control, pp. 384–389 (2002)
[19] Chen, J., Thorp, J.D., Dobson, I.: Cascading Dynamics and Mitigation Assessment in Power System Disturbances via a Hidden Failure Model. International Journal of Electrical Power and Energy Systems 27(4), 318–326 (2005)
[20] Bae, K., Thorp, J.D.: A Stochastic Study of Hidden Failures in Power System Protec-tion. Decision Support Systems 24, 259–268 (1999)
[21] Thorp, J.D., Phadke, A.G., Horowitz, S.H., Tamronglak, S.: Anatomy of Power Sys-tem Disturbances: Importance Sampling. Electric Power and Energy Systems 20(2), 147–152 (1998)
[22] Crucitti, P., Latora, V., Marchiori, M.: Model for Cascading Failures in Complex Networks. Physical Review E 69(045104) (2004)
[23] Kinney, R., Crucitti, P., Albert, R., Latora, V.: Modeling Cascading Failures in the North American Power Grid. The European Physical Journal B 46(1), 101–107 (2005)
[24] Chassin, D.P., Posse, C.: Evaluating North American Electric Grid Reliability Using the Barabasi-Albert Network Model. Physica A (2005)
[25] Watts, D.J.: A Simple Model of Global Cascades on Random Networks. Proceedings of the National Academy of Science USA 99, 5766–5771 (2002)
[26] Sun, K.: Complex Networks Theory: A New Method of Research in Power Grid. in Transmission and Distribution Conference and Exhibition: Asia and Pacific, IEEE/PES, 2005, Dalian, China, pp. 1–6 (2005)
[27] Motter, A.E., Lai, Y.C.: Cascade-based Attacks on Complex Networks. Physics Review E 66(065102) (2002)
[28] Chiaradonna, S., Lollini, P., Di Giandomenico, F.: On a modeling framework for the analysis of interdependencies in Electrical Power Systems. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK, IEEE Computer Society Press, Los Alamitos (2007)
[29] Rahman, H.A., Besnosov, K.: Identification of sources of failures and their propaga-tion in critical infrastructures from 12 Years of public failure reports. In: CRIS'2006, Third International Conference on critical Infrastructures, Allessandria, VA, USA (2006)
[30] Ortalo, R., Deswarte, Y., Kaâniche, M.: Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security. IEEE Transactions on Software Engineer ing 25(5), 633–650 (1999)
[31] Kaâniche, M., Alata, E., Nicomette, V., Deswarte, Y., Dacier, M.: Empirical Analysis and Statistical Modeling of Attack Processes based on Honeypots. In: supplemental volume of the int. Conf. Dependable Systems and Networks (DSN 2006), pp. 119–124 (2006)

Handling Malicious Code on Control Systems

Wan-Hui Tseng and Chin-Feng Fan

Computer Science and Engineering Dept., Yuan-Ze University, Taiwan
s929403@mail.yzu.edu.tw

Abstract. After the 911 terrorist attacks, the American government thoroughly investigated the vulnerabilities of infrastructure environment and found that the lack of security protection of most automated control systems is a vulnerable point. In order to ensure security in control systems, it is urgent to investigate the issue of potential malicious code, especially that made by insiders. This paper first discusses the undecidability of identifying all kinds of malicious code on control systems. However, effort to analyzing malicious code pays since it may increase the difficulty of insider attacks and improve the safety and security of the system. This paper then classifies malicious codes based on control system characteristics. The potential malicious code classifications include time-dependent, data-dependent, behavior-dependent, input-dependent, violation of a certain theorem, and so on. Finally, the paper presents possible approaches to prevention and detection of malicious code on control systems.

Keywords: control systems security, malicious code classification, undecidability**.**

1 Introduction

Since the 911 terrorist attacks in 2001, many critical infrastructure vulnerabilities are revealed. The American government has thoroughly investigated the infrastructure environment and found that the automated control system security is a weakness point. Control systems may have many security vulnerabilities, and thus, they can be attacked easily by terrorists. Moreover, once the terrorist attack happens, it will lead to a serious consequence.

The reasons why control systems have security weakness may be explained below. First, a control system usually executes in a closed-environment. It is difficult to intrude the independent network of a control system form outside. Second, if someone wants to take malicious actions, he or she must possess the domain knowledge to operate the control system. Therefore, compared to the security of information system, the control system security has been neglected in a long time. However, the 911 terrorist attacks show that the terrorist may resort to every conceivable means to achieve their goals.

During these years, many critical security events on infrastructures have happened [1,2,3,6,7]. The control system programming of Inalina nuclear power-plant in Lithuanian [1], was embedded the logic bomb by an insider who had a grievance against the government. Thus, it can be seen that the malicious code written by an

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 68–74, 2007.

insider may cause disastrous damages. Therefore, the detection of malicious code on control systems is an important issue which needs to be resolved immediately.

In this paper, we will first prove that to develop a malicious code detection program is undicidable. However, effort to analyzing malicious code pays since it may increase the difficulty of insider attacks. Then, our research focuses on the control system malicious code classification and their prevention and detection methods. There are many differences between control systems and general business systems. The control system emphasizes the real-time implementation, and operates with feedback loops, sensor inputs, actuator actions, and operator commands to execute the operation. Thus, the potential malicious programs on control systems are different from those on other systems [5,8].

The rest of the paper is organized as follows. Background will be briefly described first in Section 2. Section 3 presents undecidability of malicious code detection, followed by the malicious code classification as well as their prevention and detection methods. Finally, future work is given in Section 4.

2 Related Background

The research on malicious code classification includes McGraw [5] and R. W. Lo et al. [8]. R. W. Lo et al. [8] defined tell-tale signs and used them to detect the malicious code by program slicing and data-flow technique. However, all of the research on malicious code classification deals with general programming code. It seems that there is no research so far to classify malicious code specifically for control systems. Thus, differing from these studies, our research first focuses on control system malicious code classification.

According to the U.S. Secret Service and CERT Coordination Center/SEI statistics [9,10], the insider threat can lead to serious damages. They show that the insider threat can result in powerful attacks and severe loss. Thus, the insider attack consequence can not be neglected. So, to prevent the insider attack will be important research for control systems security.

3 Our Malicious Code Classifications on Control Systems

3.1 Undecidable Problem

To develop a universal program to detect and prevent malicious code on control systems is desirable. However, this section will prove that the general malicious code program does not exist. We assume that program Detect_Malware(p,x) detects whether a program p executes malicious code when running on an arbitrary data x. Now, we consider a new program *M(x)* as indicated below:

```
M(x){
  If (not Detect_Malware(x,x))= TRUE then
    malicious code
}
```

We may infer as follows:

1. Malicious code is executed when M runs on x iff not Detect_Malware(x,x)= TRUE (according to the definition of M)
2. If malicious code is executed when M runs on x, then Detect_Malware(M,x) = true (according to the definition of Detect_Malware)
3. Since *x* is an arbitrary input, it can be replaced by M. For x representing the coded representation of M, we obtain the contradiction Detect_Malware(M,M) = not Detect_Malware(M,M).

By this contradiction the original assumption on the existence of Detect_Malware can be proven to be an undecidable problem.

It can be seen that to develop the omnipotent program which can detect all kinds of malicious code written by insiders is impossible. In other words, to define complete malicious code classifications and find their detection methods is an extremely difficult research. However, it is very important. Such research may increase the difficulty of insider attacks and improve the safety and security of the system. In the next section based on control system properties, we develop the malicious code classifications as well as their potential prevention and detection methods.

3.2 Malicious Code Classifications

The malicious code classifications are widely investigated on general business systems. However, control system security has been neglected for a long time. Potential insider attacks and vulnerabilities of control systems must be identified. In order to ensure the control system code security, the malicious program issue needs to be investigated. For a control system, a single malicious action may not cause serious hazards. It always takes two or more instances of different types of malicious code to lead to a serious attack. In this section, we will analyze potential malicious code based on control system characteristics. The following distinguishing features of control systems are identified first: 1. Real-time implement action, 2. Feedback loop, 3.Sensor inputs, 4. Actuator actions, 5.Keeping stable state, 6. Require sequential control, 7.Involved with operator.

According to the above mentioned characteristics and existing research, we identified the following potential malicious code classifications for control systems.

(1) Time-dependent

Control systems always emphasize real-time operations. Hence, the timing-related problems are critical for them.

a. *Time-bomb*: the meaning of a time-bomb is that the malicious code is triggered at a certain time.
b. *Early or delayed message/action*: in generally, a message or an action directly influences the actual operations. An early or a delayed message/action will change the sequential operation of the control system. Thus, to keep the right control sequences is important.
c. *Race condition/Synchronization*: a race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

(2) *Data-dependent (value-related)*
Some of the control system operations depend on data setting. For example, for a car cruise control system, the speed set-point will influence the safety of a vehicle. The malicious code may change the critical data, and cause terrible damages.
   a. *Display inconsistency*: the distances between the plants and the control room are always very far. Therefore, the monitor in the control room is a key reference basis to the operator. Once the malicious code counterfeits the actual result, it will confuse the operator to perform right actions.
   b. *Change alert set-point*: the control systems include several critical alert set-points. The malicious code may change that setting to invalidate the alert.
   c. *Sending/receiving fake data*: on control systems the action depends on the sending/receiving data. Once the sensor and actuator send or receive the fake data, it would affect the proper actions.
   d. *Getting fake feedback data*: the control system may be open loop systems or closed loop systems. Thus, to get the fake feedback data would cause the system in an unstable state.
   e. *Independent/isolated data*: the anomalous data flow may include the potential malicious actions, such as use of undefined variables, use of extra data.

(3) *Behavior-dependent (action-related)*
Actuator actions affect the actual execution. Thus, the malicious actions will destroy the normal operation. Such actions include adding extra malicious actions, ignoring critical actions, executing wrong actions and mixing up execution sequences.

(4) *Input-dependent*
The malicious action may be triggered when certain inputs are encountered. The insider may add abnormal actions in the program triggered by the particular inputs.

(5) *Violation of a certain theorem/rule/relation/invariant*
A sound control system should always keep the system in a stable state. The system can not violate certain theorems, such as conservation of mass, the law of the conservation of energy, and so on.

(6) *Deviation errors*
In the control system, the damages may not occur immediately, but they may result from accumulated deviation actions over a long period.

In addition to these categories, we may also include results from existing research on malicious code classification. Thus, *isolated codes, change of privilege/authentication, anomalous file access, and access of text segment as data* are also considered.

3.3 Prevention and Detection Methods

In the above section, we have shown that it is not feasible to develop a general malicious code detection program. However, in order to alleviate the damages from the malicious code. We proposed offering different prevention and detection methods at software developing stages. First, in the analysis and design stages of software development, adequate constraints can be added to the system input and output data.

The constraints data may consist of safety range, type, unit, exception handling and so on. Constraints [4] keep systems in a stable and safety state, and thus, avoid unnecessary hazards. Furthermore, dynamic monitors can be used to check whether constraints are violated at run time. Consequently, the malicious code of the *Data-dependent* category can be prevented.

Next, in order to find out malicious code from source code, we suggest that program slicing can be used first to extract related data, specified commands or timing-related programming fragments. After getting the fragments, reviewers can judge whether the extracted parts are suspicious with malicious code. When this method is used, the malicious code of the *Time-dependent and Data-dependent* types can be found. In order to detect the malicious code of the *Behavior-dependent* type**,** we supposed that to utilize reverse engineering to reconstruct statecharts from source code and identify the unsafe state combination. According to the reconstructed statechart, the illegal transitions leading to unsafe states can be detected to reveal potential *Behavior-dependent* type of malicious code.

As to the malicious code of the *Time-dependent, Data-dependent, Behavior-dependent, Input-dependent, violation of a certain theorem and Deviation errors*, we advise to perform testing. Generally, testing can detect the abnormal behavior from code. However, in order to detect *Input-dependent* type of malicious code, the fault injection method can be included. Once the faults injected to the code, we can observe the possible system behavior which may show the insecure outcome. Nevertheless, testing effort can only increase the difficulty faced by the insider; it can not guarantee to detect all of malicious spots. Next, *Deviation errors and violation of a certain theorem* may lead to damages after the system has run for a long period, such malicious code may be detected through testing the system over a longer period of time, or detected through constraint checking at run-time.

Table 1 summarizes the malicious code classification and their prevention and detection methods by a cruise control system case. For example, once a time bomb planed in the code, program slicing method may extract time relevant fragments. Thus to analyze the extracted fragments, one can find the malicious code. About changing alert set points type of malicious code, constraints can check safety ranges and types and so on. For adding extra malicious actions types of problems, we can identify unsafe state, i.e. brake and accelerator actions run simultaneously, by basing on reconstructed statecharts to diagnose whether this case happens. For *Deviation errors*, the throttle delta computation exist deviate incrementally. This problem may be detected by running the system over a period time, or be caught eventually by constraint checking at run time. Thus, our prevention and detection methods may make malicious code difficult to hide, and thus, reduce the probability of potential damages.

4 Future Works

This research focuses on handling the control system malicious code based on our malicious code classification. Our classification includes time-dependent, data-dependent, behavior-dependent, input-dependent, violation of a certain theorem, deviation error, isolated codes, change of privilege/authentication, anomalous file access, access of text segment as data, and so on. Possible prevention and detection methods for these categories of malicious code were also addressed. In order to

Table 1. Cruise control system example

Combination of malicious code classification	Example (cruise control system)	Prevention and detection
(1) Time-dependent		1. Program slicing 2. Testing
a. Time-bomb	To trigger accelerator action at a certain time	
(2) Data-dependent		1. Constraint and run-time monitor 2. Program slicing 3. Testing
a. Display inconsistency	Speed display inconsistency (less actual speed)	
b. Change alert set point	Change speed set point	
c. Sending/receiving fake data	sensor send fake data	
d. Get fake feedback data	Feedback the fake vehicle speed	
(3) Behavior-dependent		1. Reverse engineering 2. Testing
a. adding extra malicious actions	clean setting data (speed set point)	
b. ignoring critical actions	Brake or accelerator fail	
c. executing wrong actions	replace brake by accelerator action	
(4) Input-dependent / Time-bomb + Behavior-dependent	Specified commands/time + Behavior-dependent	1. Program slicing 2. Testing
(5) Input-dependent / Time-bomb + Data-dependent	Specified commands/time + Data-dependent	1. Program slicing 2.Testing
(6) Behavior-dependent + Data-dependent	Send fake vehicle speed + Display computed vehicle speed	1.Program slicing 2.Testing
(7) Deviation errors	To add a random value to throttle delta computation over a long period	1. Testing

enhance the security of control systems and reduce their vulnerabilities, the final goal is to discriminate the malicious code from the benign ones. In the future, we plan to construct a systematized way to enhance system safety and security from software development early stages and develop practical tools that address the automatic detection of the malicious code on control systems.

Acknowledgement

This work was supported in part by graduate student scholarship by Atomic Energy Council, Taiwan.

References

1. Lithuanian nuclear power-plant logic bomb detected, Software Engineering Notes, vol. 17(2) (2006)
2. Major power outage hits New York, other large cities (2003)
3. Calton: Taipei Subway Computer Crash. The Risks Digest 18(17) (1996)

4. Leveson, N.G.: Intent specification: an approach to building human-centered pecifications. IEEE Transactions on Software Engineer 26(1) (January 2000)
5. McGraw, G., Morrisett, G.: Attacking malicious code: Report to the Infosec research council. IEEE Software 17(5), 33–41 (2002)
6. Poulsen, K.: Slammer worm crashed Ohio nuke plant network, SECURITYFOCUS NEWS, Security Focus (2003)
7. Matt, L.: US software 'blew up Russian gas pipeline, ZDNet, UK (2004)
8. Lo, R.W., Levitt, K.N., Olsson, R.A.: MCF: A Malicious code Filter. Computers and Security 14(6), 541–566 (1995)
9. U.S.Secret Service and CERT Coordination Center/SEI, Insider Threat Study: Computer System Savotage in Critical Infrastructure Sectors (2005)
10. U.S.Secret Service and CERT Coordination Center/SEI, Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector (2004)

Management of Groups and Group Keys in Multi-level Security Environments

Mohammad Alhammouri[1] and Sead Muftic[2]

[1] Computer Science Department, The George Washington University, Washington DC, USA
hammouri@gwu.edu
[2] Computer and System Sciences Department, Royal Institute of Technology, Stockholm, Sweden
sead@dsv.su.se

Abstract. This paper describes techniques and solutions for management of groups and cryptographic keys when sharing secure documents protected at different classification levels. Such access control environment enforces access to documents at multiple security classification levels, starting from the membership in the group, then access to particular group applications, then access to individual documents and finally even their sections.

Keywords: Access Control, Key Management, Group Management, Key Agreement Protocols.

1 Introduction

Current protocols for distribution of group messages and group keys require group members to be online in order to receive re-key and group management messages [1, 2, 3, 4, 5, 6, 7]. These are very restrictive protocols both for scheduling and distribution of group keys, as well as for deployment by secure group applications such as long-term sharing of protected documents, but they are suitable for on-line exchange of short messages within groups.

In this paper we propose a new definition of group membership by introducing two membership states: on-line and off-line, and two group memberships: group membership and session membership. Group membership is used to identify group members, which means that it is used for registration purposes. Session membership is used for communication purposes because it identifies the subset of group members who can participate in the group communication at any time.

Using our new group membership concept, we introduce the new features of secure group communication systems, i.e. the possibility to archive group messages, as well as group keys. This provides the possibility for delayed verification of messages, as well as continuous membership in a group, even after leaving the session.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 75–80, 2007.

2 Related Work

2.1 Group Key Agreement Protocols (KAPs)

Most of KAPs are based on long-term keys scheme, which assigns a long-term unique key for each member [2, 3, 4, 7, 11]. This long-term key is established by a peer-to-peer communication protocol between each group member and the Key Server (KS) or Group Controller (GC).

C. Rotaru [9] analyzed the cost of adding security to group communication systems, which includes key management and data encryption, and she showed how significant is the delay caused by rekey activities.

Based on the key creation and distribution method, the KAPs can be divided into two main types: Centralized Key Agreement Protocols and Contributory Key Agreement Protocols [2, 3, 4, 5, 8].

2.2 Document Based Collaboration and Access Control Systems

There are many systems that provide protection to documents, but some of them do not address security requirements in shared and collaborative environments [12, 13, 14], such as sharing documents between cryptographic groups.

There are several systems developed to provide the required platforms and functionalities for documents sharing and collaboration such as Microsoft SharePoint Services, IBM Document.Doc which is part of the IBM's Lotus Notes suite, and Oracle's Collaboration suite. These three systems provide access control at the folder and document levels only, but not at the section level within the document.

3 Group Membership and Rekeying in Collaborative Environments

The actual problem with the current group and key management protocols is that they do not allow group members to enter an off-line state without losing their membership. Leaving the group means that a rekey event has to occur and it occurs every time a member rejoins the group. Besides the overhead of rekeying events [10], this approach can not differentiate between new group members, who should not have access to the previous communications and the returned users who should be allowed to do so.

In this work we are trying to avoid the overhead of rekeying events that occur because of rejoin activities by minimizing the number of such activities. We are trying also to re-define the group membership states so group-based applications can allow group members to change their status back and forth between on-line and off-line while guaranteeing message delivery to off-line members when they re-enter the on-line state again. To do so we introduce two states (on-line and off-line) for group members.

In our solution we differentiate between two types of group memberships: group membership that requires key changes, in case of join and leave activities, and session membership that does not require any key changes.

The current group management and key distribution protocols allow the users to go through two states: Join Group and Leave Group. Any group member can communicate with the other group members as long as he/she is an active member which is the status between join and leave. If any member leaves the group, then he/she has to rejoin the group and this will cause a rekeying event. Also, based on groups and key management terminology, no user should have access to any data exchanged within a group after leaving that group, which means if the same user re-joins the group, then he/she will be prevented from viewing the data that was exchanged after he/she left the group.

The key management in our model uses access controls like the one described in [15], which is used also to controls and manages the access of group members to the group keys, to differentiate between authorized and unauthorized so it can provide the authorized users with the appropriate keys as we will describe later in section 5.

In our approach the user can change his/her status from on-line to off-line and vice versa without any need to change the group key every time he/she changes his/her status. The only cases in which the key changes are when any new member joins or an existing member leaves the group. In Figure 1, the first state is called Join Group. In this state the joined user becomes a group member, which requires an update and dissemination of the members list to current members. In addition, the group key must be changed and disseminated to group members. In this state the user can not exchange messages with the other members yet. The second state is the on-line state. It means that the user has already joined the group and he/she will be joining an active session to participate in the group communication. The off-line state means that the user has already joined the group, but he/she will be leaving an active session, and therefore can not participate in the group activities, but he/she will not be losing his/her group membership.

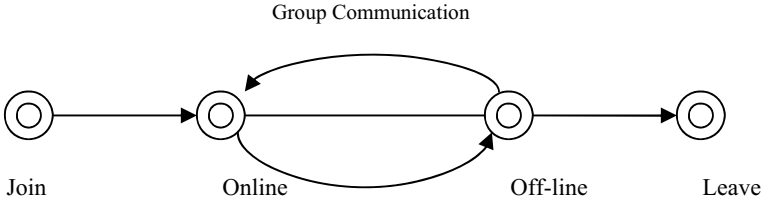


Fig. 1. On-line/Off-line States

During the off-line status of any member, other members may continue their communication. Once that member rejoins the session he/she must be updated with the current group key, in case if it has changed, as well as with all messages exchanged during his off-line mode period. The problem in this case is that this member has no knowledge about the keys used to encrypt these messages, if they were encrypted by different group keys. This means that he/she will not be able to decrypt these messages and read them.

4 Group Key Management and Document Protection and Delivery Process

In document-based sharing systems, group members are not required to be on-line all the time. In this case, a special server, equivalent to a mail–server, is needed to store and manage delayed delivery of keys and messages. Since the group keys change from time to time, there is a big chance that some archived messages will be encrypted with different keys. This leads to a conclusion that the group keys should be long-lived keys in order to use them to decrypt archived messages.

In the systems that share multi-level security documents the documents are divided into multiple sections. This means that the messages are composite and not one-unit messages. When any group member requests an access to a shared document, the authorization processor, depending on the user's authorization level, might return only portion of the document which represents the portion that the user is authorized to access. Also, since the document is a composite of multiple sections modified or added at different times, then it might have sections encrypted with different keys, because the keys change from time to time during the group life time.

To create long-life keys we divide the group lifetime into multiple time windows. Each window represents the lifetime of one key. This means that any window is opened and closed based on rekey activities. Each window is opened when new key is created and it is closed when that key is changed, and new window is opened for the new key and so on. Each key value is recorded in addition to the time period when it was effective. So when an access request is received the server will determine which keys were used to encrypt the document when it was uploaded or modified.

This approach is used to archive all keys during the group lifetime. It records the keys as well as the starting and ending time points. This information will be used later to specify the key that was effective at any moment and then use it for decryption/ encryption of the group messages that were exchanged during that key time period.

When archiving a multi-level document at the server, the server needs to know the encryption keys so they can be used later to decrypt the saved document. Our approach to protect the document is by encrypting each section in the document with the key that was in use when it was added or updated. This will produce documents encrypted with multiple keys. This means that only the updated parts of the document will be re-encrypted, which improves the performance of the system not like the approach that re-encrypts the whole document whenever any section changes.

5 The Analysis of the Solution

Encrypting the transmitted document by the current group key is not secure enough because unauthorized members can access the documents during transmission when requested by authorized members because they have the current group key. This approach protects the document against the group outsiders only since they do not have access to the group key. From this scenario we conclude that access control system by it self is not sufficient, because without encryption everybody will be able to decrypt the group communication including group outsiders. And confidentiality

only is not sufficient, because the group members will be able to view sections that they are not authorized to view. To solve this problem we propose two solutions and we analyze their advantages and disadvantages.

5.1 Subgroups vs. Members Exclusion Using Complementary Values

In the first solution, to deliver the document to the right users, we create a temporary subgroup that contains homogeneous members, i.e. those who are authorized to see the same document, and then send the document to this subgroup. In this approach the key server, based on the access control processor results, identifies the authorized users and by using the long-term keys, it establishes a new group that has the authorized members and shares a new key with them.

To deliver the document the key server encrypts the document with the subgroup key and sends it to them. The other group members will not be able to view the transmitted document because they don't have access to the new key. Therefore, we guarantee that only the authorized users will be able to read the protected document.

One disadvantage is the need to create many groups within the main group to guarantee secure delivery of the document. The worst case is to have as many groups as members. The second disadvantage is the continuous key agreement and distribution actions between subgroups.

In the second solution we exclude the members who do not have the right credentials that allow them to view the delivered sections. In this solution the key server shares a new key with only the authorized members and delivers the document using this key. As a result of this, no other members will be able to decrypt the message.

The new secret key is based on the complementary variables approach as in [3]. In this approach every group member will have access to the group current key and a set of complementary values of all other group members except his/her own. The only entity that has access to all complementary values is the key management server.

For example when member ***M1*** requests an access to any document, the access control processor analyzes the request and generates a matrix that shows the list of members who can access any section. For example, if the access control decides that the members ***M1*** and ***M2*** should be allowed to access sections **1** and **2**, then the key server creates a temporary key ***K'*** and uses it to encrypt the message. The new key ***K'*** is a function **F (K, M3', M4')** of the current group key ***K*** and all of the unauthorized members complementary values **(M3', M4').** At the client side only members ***M1*** and ***M2*** will be able to decrypt the message because they are the only members who have access to the values ***K***, ***M3'***, and ***M4'***.

This solution is based on excluding non-authorized members from the group. It works well, with one exception: if two unauthorized members collaborate with each other. There are two possible solutions to the collaboration problem: the first solution is to change all complementary values when any new member joins the group. The second solution is by preventing any communication between group members. The second solution can solve the problem by preventing group members from exchanging complementary values, but it puts more restrictions on the communication between group members.

6 Conclusions

In this paper we solved problems of sharing protected documents within a group. We described how to protect multi-layered documents, even when they are created and then edited at different times, still with the full protection for controlled sharing within a group. Since it may be expected that group keys are different at these times, the document has sections encrypted with different keys, based on the time of creation and/or editing of those sections. For such cases security system described in this paper can analyze the structure of protected documents and accordingly apply correct cryptographic keys before delivering the target document to the user. Finally, the system addresses not only protection of documents and messages within a group against external intruders, but it also prevents any unauthorized group member from intercepting and accessing those sections of a document that he/she is not authorized to access, even though the user is in possession of the current group key. This may be a problem when documents, sent to some authorized member of a group, are intercepted by any other members of the group, but with the authorization different from the original recipient of the document.

References

1. Keider., I.: GroupCommunication. In: Encyclopedia of Distributed Computing, Kluwer Academic Publishers, Dordrecht (2001)
2. Muftic, S.: A Survivable Group Security Architecture, Design and implementation Specification. In: CSPRI (2002)
3. Wallner, D., Harder, E., Agee, R.: Key Management for Multicast: Issues and Architectures. In: NSA (June 1999)
4. Harney, H., Muckenhirn, C.: Group Key Management Protocol (GKMP) Architecture. In: RFC 2094 (July 1997)
5. Ingemarsson, I., Tang, D., Wong, C.: A Conference Key Distribution System. IEEE Transactions on Information Theory 28(5), 714–720 (1982)
6. Burmester, M., Desmedt, Y.: A Secure and Efficient Conference Key Distribution System. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, Springer, Heidelberg (1995)
7. Harney, H., Meth, U., Colegrove, A., Gross, G.: GSAKMP: Group Secure Association Group Management Protocol. SPARTA, Inc., IdentAware Security (May 2005)
8. Schmolzer, G.: Secure Group Communication Systems (March 2003)
9. Nita-Rotaru, C.: The Cost of Adding Security Services to Group Communication Systems. Technical report, CNDS (2000), http://www.cnds.jhu.edu/publications/
10. Amir, Y., Kim, Y., Nita-Rotaru, C., Tsudik, G.: On the performance of group key agreement protocols. Tech. Rep. CNDS2001 -5, Johns Hopkins University, Center of Networking and Distributed Systems (2001), http://www.cnds.jhu.edu/publications/
11. Harney, H., Schuett, A., Colegrove., A.: GSAKMP Light. In: Internet Draft, IETF (2001)
12. Domino.: Doc 3.5 Administrators Guide (2002)
13. Desai, L., Schliefer, S.: Oracle Collaboration Suite, an Oracle Technical White Paper (July 2003)
14. Microsoft Windows SharePoint Services Administrator Guide 2.0 (2003)
15. Alhammouri, M., Muftic, S.: A Model for Creating Multi-level-security Documents and Access Control Policies. In: Proceedings of the 8th International Symposium on Systems and Information Security, November 2006, Sao Paulo, Brazil (2006)

Application of the XTT Rule-Based Model for Formal Design and Verification of Internet Security Systems*

Grzegorz J. Nalepa

Institute of Automatics,
AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Kraków, Poland
gjn@agh.edu.pl

Abstract. The paper presents a concept of support for the design and analysis of Internet security systems with a rule-based methodology. It considers a web security architecture, including a network and application-level firewall with intrusion detection systems. The XTT methodology allows for hierarchical design, and on-line analysis of rule-based systems. It is applied using the Unified Firewall Model, allowing for implementation-agnostic formal design and verification of firewalls. UFM extension aimed at integration with ModSecurity HTTP firewall are introduced.

1 Introduction

In order to provide security, complex Internet security systems combine number of advanced techniques from different domains [1]. In this heterogeneous environment finding an accurate approach to the problem of the design of such systems remains a challenge. This paper presents a concept of practical support of design and analysis of selected Internet security systems with a formal methodology, based on the classic rule-based programming paradigm [2,3]. The paper considers an extended security architecture for web systems security, including a network-level and application-level firewalls integrated with an intrusion detection system (IDS). The rule-based XTT methodology allows for hierarchical design, and on-line analysis of rule-based security systems [4]. The methodology is applied, using the Unified Firewall Model (UFM) [5,6], allowing for implementation-agnostic formal design and verification of firewalls. The paper presents extensions towards integration with an application-level HTTP firewall.

This paper is organized as follows: in Sect. 2 a short discussion of rule-based security systems is given, and the architecture of integrated web firewalls is discussed. In Sect. 3 important elements of rule-based systems (RBS) formalism are recalled, with the XTT design process briefly discussed. In Sect. 4 the UFM is presented, along with extensions. Elements of the visual UFM design are presented in Sect. 5. Directions for future work are presented in the Sect. 6.

* The paper is supported by the Hekate Project funded from 2007–2009 resources for science as a research project, and from AGH University Grant No.: 11.11.120.44.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 81–86, 2007.

2 Rule-Based Computer Security Systems

Network firewalls are the most common component of every network infrastructure. They are in fact optimized real-time rule-based control systems. A natural feedback for firewalls is provided by intrusion detection systems (IDS) that play a critical role in monitoring and surveillance. Network infrastructure based on network firewalls and IDS is often extended with application-level firewall solutions. These are especially important with complex web services, based on the HTTP protocol. Solutions such as *ModSecurity* [7], allow for HTTP traffic monitoring and filtering, with real-time intrusion detection. The practical development of integrated security systems is non trivial. The fact is, that it is usually a complex engineering task, often close to a hand craft activity. Improving this design and analysis process remains an area of active research.

The general idea behind this paper is to consider an integrated hybrid network and web application-level firewall model. A statefull network firewall serves as a gateway to a demilitarized zone (DMZ), where the main web server is located; it is integrated with an application-level firewall, working at the HTTP level. The infrastructure includes an intrusion detection system with several sensors for the subnetworks. In this approach an open implementation, called *ModSecurity* [7] (`www.modsecurity.org`) is considered. *ModSecurity* is an embeddable web application firewall available for the well known opensource *Apache2* webserver. Ultimately, the application of the UFM/XTT approach presented in this paper should develop into an integrated design methodology, combining both network-level and application-level firewall, and the intrusion detection system.

3 Formal Rule-Based Systems Analysis with XTT

Rule-Based Systems (RBS) [2,3] constitute a powerful AI tool for specification of knowledge in design and implementation of systems in many domains. In the formal analysis of RBS important aspects of the design and implementation are identified, such as *rulebase design*, and *inference engine implementation*. In order to design and implement a RBS in a efficient way, the knowledge representation method should support the designer introducing a scalable *visual representation*. As the number of rules exceeds even relatively very low quantities, it is hard to keep the rule-base consistent, complete, and correct. These problems are related to knowledge-base verification, validation, and testing. To meet security requirements a *formal analysis and verification* of RBS should be carried out; it usually takes place after the design. However, the XTT method allows for on-line verification during the design and gradual refinement of the system.

The main goal of the XTT approach is to move the design procedure to a more abstract, logical level. The design begins with the *conceptual design*, which aims at modelling the most important features of the system, i.e. attributes and functional dependencies among them. *Attribute-Relationship Diagrams* [3], allow for specification of functional dependencies of system attributes. An ARD diagram is a hierarchical conceptual system model at a certain abstract level.

The ARD model is the basis for the actual XTT model which allows for the *logical design*. The main idea behind *XTT* [8] knowledge representation and design method aims at providing a hierarchical visual representation of the decision tables linked into tree-like structure, according to the control specification provided. The logical design specification can be automatically translated into a low-level code, including Prolog, so that the designer can focus on logical specification of safety and reliability. The translation is the *physical design*. Selected formal system properties can be automatically *analyzed on-line* during the logical design, so that system characteristics are preserved. In this way XTT provides a clear separation of logical and physical design phases.

4 The Extension of the Unified Firewall Model

The *Unified Firewall Model* (UFM) [5] is a formal, implementation-free firewall model, build on top of the XTT methodology, providing a unified attribute specification for representing network firewalls. It is introduced as a middle-layer in firewall design process, enabling formal analysis of the created firewall. Generation of target language code for specific firewall implementation is achieved by defining translation rules from the abstract model into a specific implementation. In order to apply the UFM-based approach to firewall system design it is necessary to define: formal firewall system attributes, attributes domains, and syntax for expressing firewall policy in different implementations A full list of conditional firewall attributes is specified; they correspond to information found in network packets header. The specification is given in the Table 1, where each attribute is specified with: *Name*, *Symbol*, *Subset* (the position in inference process, specifying whether attribute is *input*, *output* or its value is defined during inference process – *middle*), and *Atomicity* (specifying whether attribute takes only *atomic* values from specified domain or also *sets* or *ranges* of these values).

In order to construct a practical firewall implementation, it is necessary to provide a formal translation from the unified model to particular implementations. Two open firewall implementations have been considered so far: Linux

Table 1. UFM Attribute Specification

Name	Symbol	Subset	Domain	Atomic
Source/Destination IP	aSIP/aDIP	input	Ipaddr	*set*
Protocol	aPROTO	input	Protocol	*set*
Destination port	aPORT	input	Port	*atomic*
Input/Output interface	aIINT/aOINT	input	Interface	*atomic*
ICMP type	aICMPT	input	Icmptype	*atomic*
ICMP error code	aICMPC	input	Icmperrcode	*atomic*
TCP flags	aTCPF	input	Tcpflags	*atomic*
Service	aSERV	middle	Service	*set*
Source/Destination group	aSGR/aDGR	middle	Group	*set*
Action	aACT	output	Action	*atomic*

NetFilter and OpenBSD PacketFilter (PF). Full translation contained in [5] is long and detailed, and is out of scope of this paper. The goal of this research is to extend the UFM with attributes needed to describe an application-level HTTP firewall. Some natural specification restrictions are considered here, such as the fact that HTTP packets are contained only in the TCP packets, so the specification restricts HTTP-related rules only with use of the TCP. It can be observed, that traffic restrictions are in practise down-to-top; that is when designing the system the packet traversal in the network stack must be taken into account. A packet blocked at the network firewall level, basing on the IP/TCP attributes does not reach the HTTP level application firewall. So the translation to the target languages is not trivial, e.g. a rule: "block the traffic from the IP w.x.y.z" can be carried out at the IP level, so the HTTP level firewall never sees the packet. The traffic could also be allowed at the IP level, with the packet blocked at the HTTP level. Practically, at the unified level more detailed rules are considered. Let's consider three general firewall rules, each one more specific:

```
1) block the traffic from the IP w.x.y.z
2) block the traffic from the IP w.x.y.z to destination port HTTP
3) block the traffic from the IP w.x.y.z to destination port HTTP
                                    with BODY containg "cmd.exe"
```

In the extended UFM, the first rule is translated into 1 network level firewall rule; the second rule generates 2 rules, the second one for the HTTP level firewall (the fact is, that the rules are redundant, but both security policies are consistent); the third rule also generates 2 target rules, the first one identical to the previous ones, and the second for the HTTP level firewall which is more specific. In this way, there is a more fine-grained control over the security policy.

ModSecurity supports number of complex rule cases, the most important and common is the basic: `SecRule VARIABLES OPERATOR [ACTIONS]`. Where `VARIABLES` together with the `OPERATOR` part specifies rule precondition, and the `ACTIONS` specifies the decision (if omitted the default decision is made). In order to extend the UFM to support HTTP-level firewall, ModSecurity-specific attributes have been introduced. The translation procedure had to be extended, taking into account a two level firewall. The ModSecurity-specific part is trigerred when the destination port 80 (service www) is encountered. Let us now move to the design and verification procedure, using the extended UFM.

5 Visual UFM Design with XTT

The UFM has been developed for the integration with the XTT design process, which in case of generic RBS consists of several basic stages. On top of this process a Unified Firewall Model is built. In this case, the RBS designed is an *abstract firewall*. The UFM model provides a well-defined attribute specification for the 1st phase. Using this specification, in the 2nd phase, a general ARD model capturing functional relationships between UFM attributes, has been built [5]. Its simplified version is presented in Fig. 1. Basing on the conceptual design,

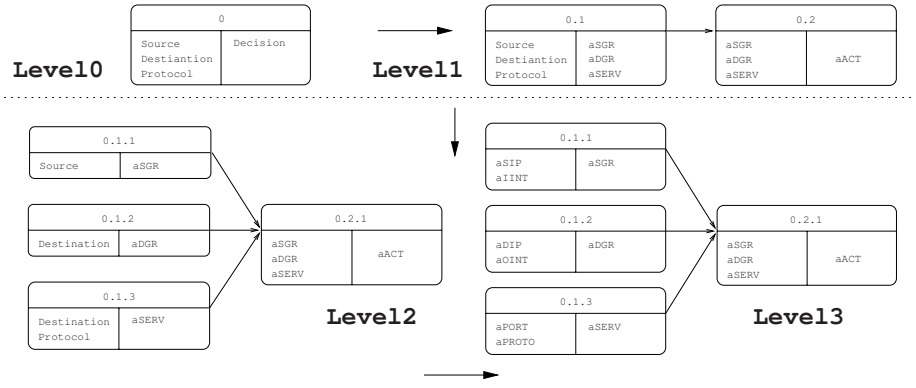


Fig. 1. The ARD Diagram for the UFM

XTT tables are built. Every row of an XTT table corresponds to a firewall rule in the UFM. Having the XTT structure partially designed, it possible to conduct a verification of the firewall structure. The last stage is the physical design, where the XTT rules are translated into a given firewall implementation language, using formally predefined translation provided by the UFM.

Let us show how rules translation to the target implementation language is performed for the an example XTT rule.

```
Precondition(aSGR=inet, aDGR=fw_inet, aSERV=www),
Retract(aDIP=f_inet, aPort=80), Assert(aDIP=d_3w, aPort=8080),
Decision(aACT=dnat)
```

This is a rule for web proxy address translation. During the system attribute specification symbolic names of given IP networks and addresses are defined, in this case these are: `f_inet`, `fw_inet`, `d_3w`. The firewall rule for the NetFilter is:

```
iptables -t nat -A PREROUTING -s 0/0 -i eth0 -d 10.10.22.129
   -p tcp --dport 80 -j DNAT --to-destination 192.168.2.2:8080
```

The full translation is discussed in [5]. It is important to point out that the expressiveness of the UFM is as high as possible, so it is closer to the more expressive target language, e.g. OpenBSD PF. However, all of the UFM syntactic structures can be translated to any firewall language, provided that the the implementation has the features represented by the UFM. The whole design process proposed in this paper is presented in Fig. 2. An important part of this process is the analysis and verification framework. The XTT approach offers a possibility of automatic, on-line formal analysis of the firewall structure *during* the logical design. The analysis is accomplished by an automatic transformation of the XTT model into a corresponding code in Prolog. An extensible Prolog-based inference engine is provided, with number of analysis modules available, for important firewall features, including completeness, or determinism.

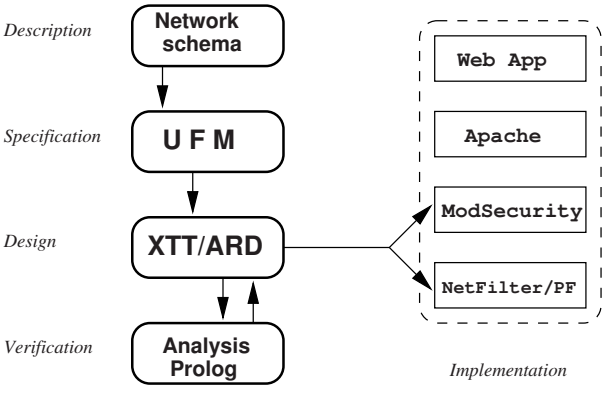


Fig. 2. UFM-based Design and Verification Process

6 Future Research

The original contribution of this paper is the extension of the formalization of the *Unified Firewall Model*, aimed at the application-level HTTP firewall, *ModSecurity* for the Apache webserver. This research should still be considered a work in progress. Future work includes: UFM application to intrusion detection systems, e.g. *Snort*, and improved verification with firewall-specific plugins. The XTT-backed UFM approach for security systems design and analysis allows for improving system quality, by introducing the formal verification during the visual design, while offering an abstract layer over common firewall implementations.

References

1. Garfinkel, S., Spafford, G., Schwartz, A.: Practical Unix and Internet Security, 3rd edn. O'Reilly and Associates (2003)
2. Jackson, P.: Introduction to Expert Systems, 3rd edn. Addison–Wesley, Reading (1999)
3. Ligęza, A.: Logical Foundations for Rule-Based Systems. Springer, Heidelberg (2006)
4. Nalepa, G.J., Ligęza, A.: Security systems design and analysis using an integrated rule-based systems approach. In: Szczepaniak, P.S., Kacprzyk, J., Niewiadomski, A. (eds.) AWIC 2005. LNCS (LNAI), vol. 3528, Springer, Heidelberg (2005)
5. Budzowski, M.: Analysis of rule-based mechanisms in computer security systems. formulation of generalized model for firewall systems. Supervisor: Nalepa, G. J., Ph. D.: Master's thesis, AGH-UST (2006)
6. Nalepa, G.J.: A unified firewall model for web security. In: The 5th Atlantic Web Intelligence Conference, Fontainebleau, France (2007) (accepted)
7. Breach Security, I.: ModSecurity Reference Manual v2.1 (2007), http://www.breach.com
8. Nalepa, G.J., Ligęza, A.: A graphical tabular model for rule-based logic programming and verification. Systems Science 31(2), 89–95 (2005)

RAMSS Analysis for a Co-operative Integrated Traffic Management System

Armin Selhofer[1], Thomas Gruber[1], Michael Putz[1], Erwin Schoitsch[1], and Gerald Sonneck[2]

[1] Austrian Research Centers GmbH - ARC, Austria
{armin.selhofer,thomas.gruber,michael.putz,erwin.schoitsch}@arcs.ac.at
[2] Tribun IT-Consulting und Softwareentwicklung GmbH & Co. KEG, Austria
gerald.sonneck@tribun.at

Abstract. The European Project COOPERS[1] aims at developing co-operative systems based on innovative telematics solutions to increase road safety. Co-operative traffic management is implemented by intelligent services interfacing vehicles, drivers, road infrastructure and operators. These services involve various types of smart systems and wireless communications and have different impact on safety. Therefore, a RAMSS analysis has been carried out in the initial phase of the project. One of the major problems faced was the lack of knowledge regarding the implementation of the system. Consequently, a holistic approach to identify the most critical parts of COOPERS had to be considered. The methods used and the results found by applying a RAMSS analysis to the specific case of co-operative road traffic management services are presented.

Keywords: RAMSS, dependability, analysis, co-operative traffic management, traffic telematics, road safety.

1 Improving Road Traffic Safety by a Co-operative Integrated Traffic Management System

In the Sixth Framework Programme of the European Commission, COOPERS [1] takes a specific position with unique ways and methods to attain road traffic safety improvement. Co-operative services will be provided through an intelligent network which exploits existing technologies and infrastructure at affordable costs.

In each car, a receiver for I2V (infrastructure to vehicle) communication and a display offer information relevant for the driver about the current traffic situation (see Fig. 1), e. g. accidents, congestions, or road work. In the reverse direction, i. e. V2I (vehicle to infrastructure), the communication channel is used for verifying infrastructure sensor data using vehicles as floating sensors.

[1] Research supported in part by COOPERS (Co-Operative Networks for Intelligent Road Safety), an integrated project funded by the EU within priority "Information Society Technologies (IST)" in the Sixth EU Framework Programme (contract no. FP6 IST 4 026814).

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 87–92, 2007.

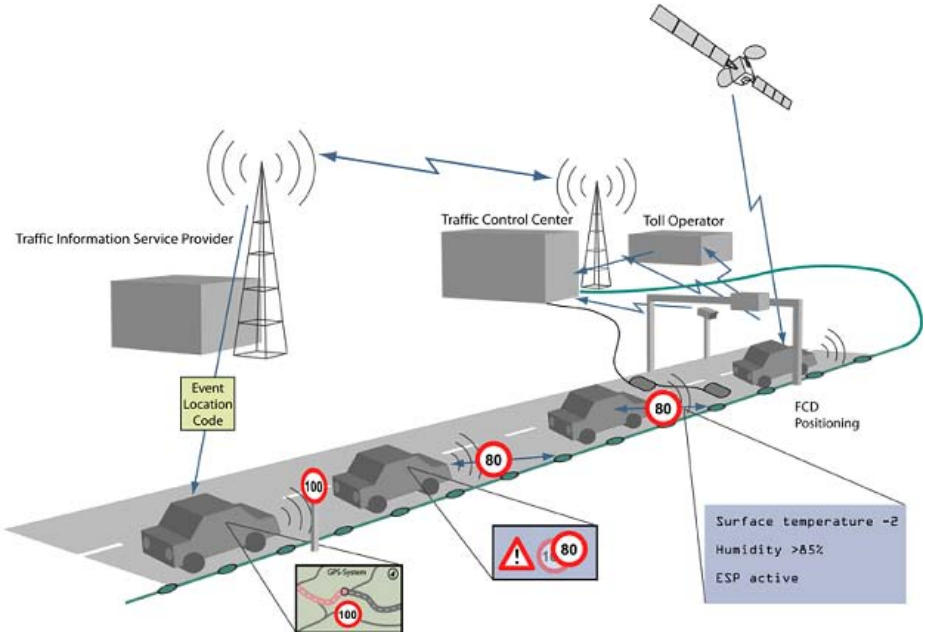


Fig. 1. Intelligent Infrastructure and Smart Cars plus individual location based services - I2V and V2I communication

COOPERS services offer information and warnings about e. g., accidents, weather condition, traffic congestion, speed limit, and road charging.

COOPERS is expected to reduce the risk in road traffic to a significantly lower value, expressed by number and severity of accidents, injuries and fatalities counts. But the implementation of the service might be faulty, and it is even possible that the driver (e. g. through distraction) is exposed to a higher risk with the COOPERS service than without. This paper describes motivation, methods used and the results of the RAMSS analysis performed for identifying such risks in an early stage of the COOPERS project.

2 Goals of a RAMSS Analysis in COOPERS

Generally, RAMSS analysis covers five different quality attributes of safety-critical systems: Reliability, Availability, Maintainability, Safety and Security (RAMSS). For COOPERS, it shall give advice on how to construct COOPERS services, regarding their functional architecture as well as the selection of appropriate technologies.

Breaking the COOPERS services down into the components of the signal flow path (see Fig. 2), it is evident that the availability of the single node functions and of each signal flow through the edges in the information flow path will play

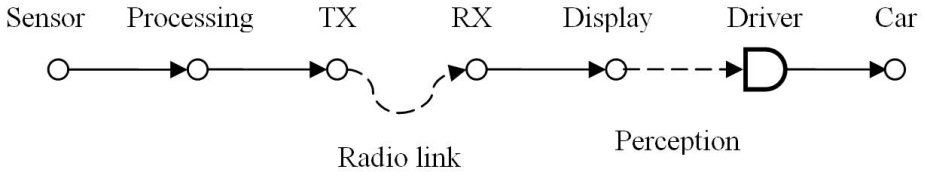


Fig. 2. Signal flow path of the data

the key role. This becomes clearer when we consider the COOPERS services as a safety function in terms of EN ISO/IEC 61508 [2], which will be explained further below.

3 Risk

For technical applications, risk is defined as "the combination of the probability of occurrence of harm and the severity of that harm" [3].

In the considerations given below, we discuss the risk to die in road traffic and neglect – for simplicity – injuries as well as material loss through road accidents.

What human beings consider a tolerable risk depends strongly on the subjective perception. Risks lower than the minimum natural risk of about 10^{-4} fatalities per year and person are commonly accepted.

3.1 Risk Reduction for Safety-Critical Systems

In EN ISO/IEC 61508-5, Appendix A, the concept of risk is explained for technical systems, which is depicted on the right hand side of Fig. 3. This description assumes a technical system that imposes an additional risk to people who are physically in the area of the system. The resulting risk is intolerable, therefore countermeasures have to be taken to reduce the risk to a residual risk level, which is below the tolerable risk.

4 Risk Reduction in COOPERS

In COOPERS, the services are intended to improve safety and therefore to reduce the risk imposed by the system "road traffic". But, unlike in usual cases of safety-relevant systems like chemical or nuclear plants, the system is expected to improve safety. In terms of ISO/IEC 61508, a COOPERS service represents a "safety function". The scope of consideration comprises therefore all involved objects and subjects: The in-car and the infrastructure equipment including the communication channel[s], as well as the driver.

Fig. 3 shows the different situation of a COOPERS service in the risk diagram. The starting point is the risk that is currently accepted ("tolerable risk"), namely

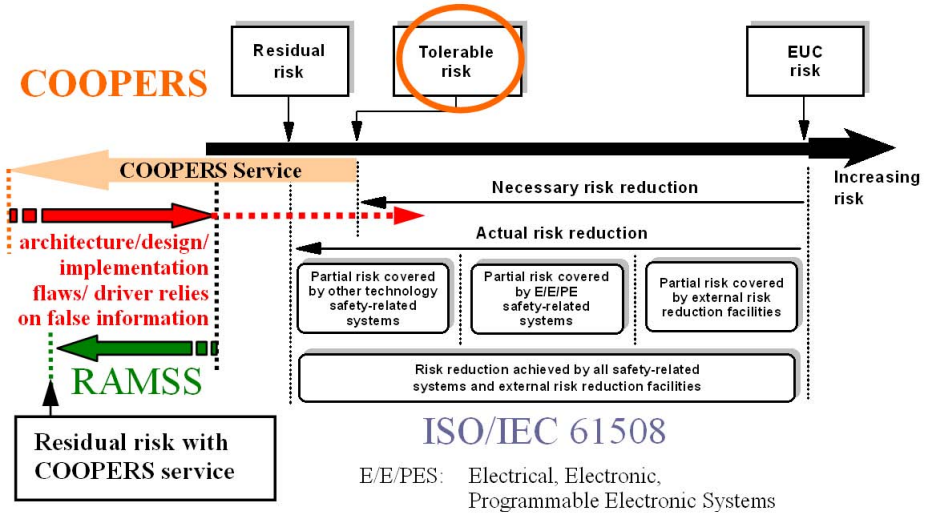


Fig. 3. Additional risk reduction by COOPERS

the risk of road traffic without COOPERS services. The EUC (equipment under control), in this case the COOPERS service, improves safety (left-oriented arrow "COOPERS Service") instead of increasing risk like in common cases.

The arrow below the latter indicates a possible loss of safety gain whose causes can be two-fold:

- Failures of the E/E/PES resulting in wrong or no information
- Driver reduces attention to the road while strongly increasing reliance on the system.

The amount of safety loss could even be larger than the safety gain by the COOPERS services and overshoot the established tolerated risk (dotted arrow). Therefore, these reasons for safety-loss have both to be treated in the RAMSS analysis (lower arrow "RAMSS") to support the goal of the project.

5 Analysis

At the early stage of the project it had been necessary to operate with assumptions about functional architecture and typical technologies. At first during a Preliminary Hazard Analysis (PHA, see also [4],[5]), the co-operative system was investigated to identify the hazards due to road traffic. Subsequently the Hazard and Operability Analysis (HAZOP) examined systematically each node of the signal path depicted in Fig. 2. Deviations from the original design intent for each node were studied and potential problems were collected to be able to tell which parts of the system have to be designed with special care.

The question whether COOPERS is safety relevant or not depends on the way how COOPERS interacts with the car and the driver:

1. If the services directly control safety related functions (e. g. steer or brake the car), COOPERS is safety relevant and the SIL (Safety Integrity Level) has to be determined.
2. If the drivers are assumed to be always finally responsible for steering the car, then COOPERS services need not to be treated as safety-relevant (like for instance car radio for traffic messages).

The recommended implementation for COOPERS is that the services do not directly control safety functions (SIL = 0) but provide information to the driver that can either improve safety (e. g. warnings) but could also reduce safety (e. g. false information, overstressing or distracting the driver). A SIL ≥ 1 would otherwise increase the development efforts significantly.

6 Results

6.1 Hazard Analysis

The PHA results in the hazards of the road traffic, of which the traffic accident is the most obvious one. Its severity is strongly related to the kinetic energy stored in the moving vehicle. Since COOPERS does not add kinetic energy or harmful substances to the system it does not influence the severity of the hazards. On contrary, the system provides information about the current traffic and road condition with the intention to improve road safety, but wrong or missing information may increase the probability of accidents.

Considering common state-of-the-art components, all electronic and electric equipment can generally be expected to have a fairly low failure rate. Hence, because no details regarding the technical implementation are known yet, the RAMSS analysis will be based on assumptions taken from comparable technology and will yield results, that may be subject to change, depending on the real implementation.

6.2 HAZOP Analysis

According to the HAZOP (for a description of the method see [5]) investigation and the signal flow path (see Fig. 2), two parts were identified to contain a relatively higher risk of failure than other parts. These are the radio link (TX and RX) and the human driver relying on the provided information in the car.

Wireless Radio Link. The wireless radio link is expected to have a remarkably higher probability of failure than other components, because of the immanent mobility of the vehicle, the change in quality of the communication media, multi-path propagation, and interference. It is therefore rated as a part within the signal path which requires particular attention. A general recommendation to

increase the availability of the wireless transmission is to use redundancy, for instance two independent technology standards (e. g. DSRC and GSM) for the radio link.

Human Machine Interface. The intention of the COOPERS services it to deliver information to the driver. The presentation of this information must be unambiguous, clearly comprehensible and consistent, least distracting as well as congruent with other, similar information sources, e. g. the perception of the surrounding by the driver. The design of the human machine interface can have a big impact on how people perform.

Comment Regarding the Results. The overall risk cannot simply be obtained by a linear superposition of the risks of the single services. COOPERS often uses a combination of multiple services where consistency plays a important role. In extreme cases, poor services implementation may end up in an even higher risk compared to the current road traffic situation (dotted arrow in Fig. 3). RAMSS analysis helped to ensure that the best possible safety gain by COOPERS services will be attained (lower arrow "RAMSS" in Fig. 3).

Finally, the COOPERS system has to detect misoperation of the services and must inform the user about it. Otherwise, the driver could assume there is no hazardous situation ahead and be exposed as well as expose others to a higher risk than without COOPERS.

7 Conclusion

In this paper the hazards and operability problems of COOPERS services have been investigated and the most relevant results of the RAMSS analysis documented. Due to the early project phase, a holistic approach and qualitative methods have been used.

The analysis identified parts that need more attention with respect to safety and reliability than others, namely the wireless connection (I2V and V2I) and the design of the human machine interface (HMI). Nevertheless there are also other parts of the system that are exposed to safety risks and therefore require attention.

References

1. Gruber, T., Schoitsch, E.: Automotive Visions beyond In-Car Driver Assistance: Integrated Traffic Managment with Coopers. ERCIM News (67), 18–19 (October 2006)
2. ISO/IEC, E.N.: 61508. Functional Safety of Electrical/Electronic/Programmable Electronic Systems, Part 1 - Part 7 (1998–2001)
3. ISO/IEC 61508-4, Definitions and abbreviations
4. US Department of Defense. Electronic Reliability Design Handbook. Technical report (October 1998)
5. American Institute of Chemical Engineers. Guidelines for Hazard Evaluation Procedures. American Institute of Chemical Engineers (1992)

Combining Static/Dynamic Fault Trees and Event Trees Using Bayesian Networks

S.M. Hadi Hosseini and Makoto Takahashi

Graduate School of Engineering, Tohoku University, 6-6-11-808, Aramaki-Aza-Aoba, Aoba-ku, Sendai, 980-8579, Japan
hadi.hosseini@most.tohoku.ac.jp

Abstract. In this study, an alternative approach for combining Fault Trees (FT) and Event Trees (ET) using capabilities of Bayesian networks (BN) for dependency analysis is proposed. We focused on treating implicit and explicit weak s-dependencies that may exist among different static/dynamic FTs related to an ET. In case of combining implicit s-dependent static FTs and ET that combinatorial approaches fail to get the exact result, the proposed approach is accurate and more efficient than using Markov Chain (MC) based approaches. In case of combining implicit weak s-dependent dynamic FTs and ET where the effect of implicit s-dependencies have to be manually inserted into the MC, the proposed approach is more efficient for getting an acceptable result.

1 Introduction

Event Tree (ET) analysis is one of the most common methods for accident sequence modeling when performing Probabilistic Safety Assessment (PRA) [1]. ET is a graphical model that depicts the response of different mitigating functions (Headings of ET) of the system to an Initiating Event (IE) and results in different sequences based on success or failure of these Headings. The failures of these Headings may be then analyzed using static and/or dynamic Fault Trees (FT) [2]. FT is a graphical model of the various parallel and sequential combinations of faults that may result in occurrence of a predefined undesired event. ET and related FTs are then combined together for further quantification of occurrence frequencies of different ET sequences. When ET branches are statistically independent (s-independent) events, frequency of each sequence will be easily quantified as the product of success or failure probabilities of the Headings along corresponding path from IE to that sequence further multiplied by IE frequency.

Despite of strong qualitative capability of ET for accident sequence analysis, one of the main limitations is the difficulty to analyze weak s-dependencies that may exist among different branches of an ET. Two branches of an ET have explicit weak s-dependency when there is at least one shared Basic Event (BE) among corresponding FTs. However, when there are implicit s-dependencies among BEs of FTs related to different ET branches, we say those ET branches have implicit weak s-dependency.

In case of combining static FTs and ET with explicit weak s-dependencies, traditional approaches result in analyzing non-coherent FTs [3] where the

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 93–99, 2007.

convergence of inclusion-exclusion expansion (exact calculation) is very slow and other approximations such as Minimal Cut Sets (MCS) upper bound approximation, coherent approximation are not accurate [4]. A new approach proposed by Andrews and Dunnett [4] using Binary Decision Diagrams (BDD) is more efficient and accurate for analyzing ETs with explicit weak s-dependent branches, but it is not capable of treating other kinds of dependencies especially implicit weak s-dependencies among different branches of an ET and analyzing ETs with multiple branches (along with weak s-dependencies).

In case of combining dynamic FTs and ET, the most efficient method is to use a combination of BDD and Markov Chains (MC) [1] for static and dynamic modules separately. However, when there are weak s-dependencies among ET branches (modeled by static and dynamic FTs), we should make a single MC for whole the s-dependent FTs (including dependent static FTs) involved in an ET that is not efficient due to the obvious disadvantage of MC that the size would face a state-space explosion problem with the increase of number of events.

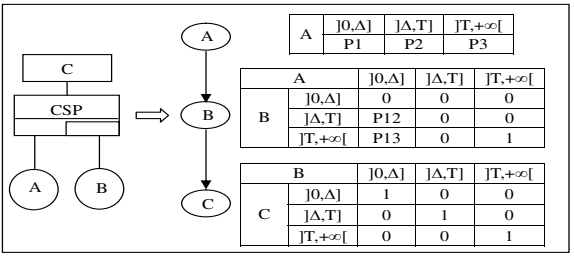
To address the above mentioned problems, we exploited a recently developed methodology [5] for mapping dynamic FT into Bayesian Networks (BN) [6] and presented an approach to treat different kinds of dependencies in ET analysis using the capabilities of BN for dependency analysis.

2 FT Analysis Using BN

Bayesian networks also known as Bayesian belief networks, is a directed acyclic graph (DAG) formed by the variables (nodes) together with the directed edges, attached by a table of conditional probabilities of each variable on all their parents [7]. It is a graphical representation that provides a powerful framework for reasoning under uncertainty and also for representing local conditional dependencies by directly specifying the causes that influence a given effect.

Boudali and Dugan [5] developed a methodology for mapping dynamic FTs into discrete-time BN for facilitating dynamic FT analysis. In this methodology, in order to consider the dynamic effects of dynamic gates (e.g. spare gates, sequential gates, etc.) the time line is divided into $n+1$ intervals. Each node (random variable) in the corresponding BN has a finite number of $n+1$ states. The n first states divide the time interval $]0,T_m]$ (T_m is the mission time for analysis) into n (possibly equal) intervals, and the last ($(n+1)^{th}$) state represents the time interval $]T_m,+\infty[$. We call this kind of nodes (with $n+1$ states and $n>1$) "dynamic node". For example, for a Cold Spare gate (CSP), considering $n=2$, the equivalent BN is shown in Fig. 1. P_1 and P_2 are the probabilities of component A, failing in the interval 1 ($]0,\Delta]$)and interval 2 ($]\Delta,T]$), respectively, and P_3 is the probability of component A, not failing in mission time (being in state 3). Suppose that time to failure distribution of component A is defined by $f(t)$. The equations for quantification of P_1, P_2, and P_3 are shown in Eq. 1.

$$P_1 = \int_0^{\Delta} f(t)dt, P_2 = \int_{\Delta}^{T} f(t)dt, P_3 = 1 - \int_0^{T} f(t)dt \tag{1}$$



A	]0,Δ]	]Δ,T]	]T,+∞[
	P1	P2	P3

A		]0,Δ]	]Δ,T]	]T,+∞[
B	]0,Δ]	0	0	0
	]Δ,T]	P12	0	0
	]T,+∞[	P13	0	1

B		]0,Δ]	]Δ,T]	]T,+∞[
C	]0,Δ]	1	0	0
	]Δ,T]	0	1	0
	]T,+∞[	0	0	1

Fig. 1. BN and CPTs corresponding to CSP Gate

The zero entries for the CPT of node *B* in Fig. 1 state that the spare *B* cannot fail before, or at the same time as, the primary *A*. Suppose that time to failure distribution of component *B* is defined by *g(t)*. The equations for quantification of P_{12} and P_{13} are shown in Eq. 2.

$$P_{12} = \frac{\int_0^{\Delta}\int_{\Delta}^{T} g(\tau - t) f(t) dt d\tau}{\int_0^{\Delta} f(t) dt}, P_{13} = 1 - P_{12} \tag{2}$$

For mapping static gates into corresponding BN nodes, using this methodology, it is just required to assign nodes with two states (state 1: failure in T_m , state 2: not fail in mission time) for them that are called "static node".

3 Combining Static/Dynamic FTs and ET Using BN

In this section, the procedure for combining FTs and ET using the described methodology is presented along with application to a redundant multiprocessor system example depicted in Fig 2 (see [8] for further details). To make a redundant bus system, an additional bus is applied, intentionally, and one power supply (*PS*) is also implemented (not depicted in Fig. 2) to supply both processors. The following s-dependencies are considered:

- Degraded mode of power supply (*PS*) increases the failure rate of P_1 and P_2 by 10% (implicit weak s-dependency).
- Failures of both buses (N_1 and N_2) will cause M_3 unavailable (time-invariant)
- M_3 is shared between two subsystems S_1 and S_2 (explicit weak s-dependency).

The ET, related FTs, and the resulted BN corresponding to failure of one of the redundant buses (N_1) (as IE) are shown in Fig. 3. To combine static/dynamic FTs and ET using BN, we define the term "dynamic events" as follows. In FTs that contains both dynamic and static gates, all the BEs, static and/or dynamic gates (subsystems) below a dynamic gate are considered as dynamic events. Also, BEs or subsystems that have time-variant weak s-dependency with dynamic events are included. To find dynamic events, the linear time FT modularization algorithm developed by Rauzy and Dutuit [9] is used with some modifications. In this algorithm, if one BE that is the

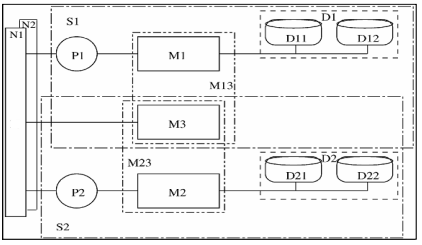


Fig. 2. A Redundant Multiprocessor System

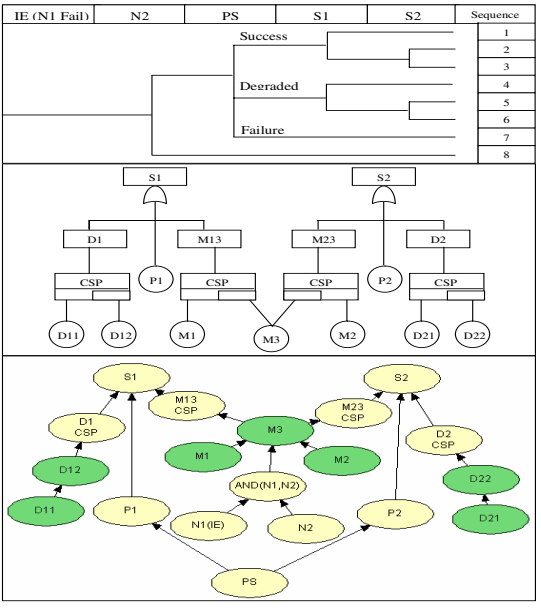


Fig. 3. FT, and Corresponding BN related to N1 Failure in Multiprocessor System

Table 1. Frequency (per hour) of ET Sequences[1]

Sequence Number	1	2	3	4	5	6	7	8
Frequency (Exact)	0.9973	0.0025	6.312×10^{-6}	9.972×10^{-5}	2.751×10^{-7}	7.611×10^{-10}	5×10^{-5}	1×10^{-5}
Frequency (n=9)	0.9973	0.0025	6.324×10^{-6}	9.972×10^{-5}	2.750×10^{-7}	7.622×10^{-10}	5×10^{-5}	1×10^{-5}
Error%	0	0	0.19	0	-0.04	0.15	0	0
Frequency (n=14)	0.9973	0.0025	6.323×10^{-6}	9.972×10^{-5}	2.750×10^{-7}	7.621×10^{-10}	5×10^{-5}	1×10^{-5}
Error%	0	0	0.17	0	-0.04	0.13	0	0
Frequency (n=19)	0.9973	0.0025	6.320×10^{-6}	9.972×10^{-5}	2.751×10^{-7}	7.618×10^{-10}	5×10^{-5}	1×10^{-5}
Error%	0	0	0.13	0	0	0.10	0	0

[1] The frequency values should be multiplied by failure frequency of N_1 (i.e. 2×10^{-9}).

input to a static gate is shared with a dynamic module, all the related static gate's inputs and output are also included in the related dynamic module. However we exclude these events from dynamic events (despite MC-based approaches).

The procedure for combining static/dynamic FTs and ET using BN along with application to the ET/FT of Fig. 3 is as follows:

1. Searching through the FTs to find dynamic events and assigning dynamic nodes to them (dark background color BN nodes in Fig.3). The remaining BEs are translated into static nodes in the corresponding BN.
2. Mapping all BEs of different FTs of ET to their corresponding BN nodes.
3. For IE, we consider a BN node with two states (occurrence and non-occurrence) with probability of occurrence of 1 (its occurrence is assumed as an evident).
4. Mapping different gates into their corresponding BN structure and applying the related conditional (or prior) probability values into the constructed BN (e.g. using equations 1 and 2 for CSP gate). In cases that we have common BEs in different FTs, their corresponding BN will be connected through those common BE nodes.
5. In order to apply implicit s-dependencies that may exist between different components, we connect related BN nodes and apply the related conditional probability values in related CPTs (e.g. PS node connected to P1 and P2 nodes in Fig. 6). Effects of *IE* on different components would be applied in the same way.
6. Deriving the logic of different sequences of the ET based on status (success, failure, etc.) of the Headings and quantifying its probability by inferring the corresponding joint probability in the resultant BN (e.g. the logic of the seventh sequence of the ET in Fig. 3 is: (*N2=success,PS=failure*)). Then multiplying the result with the occurrence frequency of IE to quantify the sequence frequency.

The results (see Table 2 for component failure data) of the quantification of different sequences of the ET using MC (exact result), and using the resulted BN considering *n=9, 14, 19* for dynamic nodes along with corresponding errors are given in Table 1.

It is clear that the maximum error for sequence frequencies is less than 0.2% for small amount of n (the relative error is about 10^{-3}). The results show that we can acquire an acceptable result with small values of n using the proposed methodology. Comparing with conventional approaches, the proposed method is more appropriate in the following cases:

- When the number of s-dependent static and dynamic modules among different ET branches increases (because of less number of dynamic nodes included).
- When no dynamic FT is included in ET, and the exact result is preferred.
- Although the process of converting dynamic gates into corresponding MC is performed automatically in codes like GALILEO [10], including the effects of implicit weak s-dependencies among ET branches should be done manually (within the state space) that is quite time-consuming and error prone.

Moreover, one of the unique features of the proposed methodology is facilitating elimination of impossible ET sequences that is one of the main concerns in constructing ETs. Using conflict measure in Eq. 2 [11], given a set of evidences $e=\{e_1, e_2, \ldots, e_n\}$, we can easily trace possible conflicts that may exist in the

constructed BN between negatively correlated evidences that result in a positive conflict measure and remove those sequences that are not possible to occur.

$$Conf(e) = \log\left(\frac{\prod_{i=1}^{m} P(e_i)}{P(e)}\right) \tag{3}$$

Where $P(e_i)$ is the probability of i^{th} evidence being in a specified state.

Table 2. Component Failure Rates and Probabilities

Component	Failure Rate / hour	Failure Probability
Processor (P_i)	5×10^{-7}	2.50×10^{-3}
Memory (M_i)	3×10^{-8}	1.50×10^{-4}
Hard Disk (D_{ij})	1×10^{-6}	5.00×10^{-3}
Bus (N_i)	2×10^{-9}	1.00×10^{-5}
Power Supply (PS)	Failure: 1×10^{-8}	5.00×10^{-5}
	Degraded: 2×10^{-8}	1.00×10^{-4}

4 Conclusions

An alternative approach for combining s-dependent static/dynamic FTs and ET has been proposed that makes use of capabilities of BN for dependency analysis. Using the proposed approach, we can acquire the exact result for combining static FTs and ET with implicit weak s-dependency among FTs along with multiple ET branches. The proposed approach is a good alternative in place of conventional MC-based approaches especially when there are shared events among static and dynamic FTs and when implicit weak s-dependencies exist across the ET branches. Capability of the new approach for elimination of impossible accident sequences is quite useful for accident sequence analysis. However, the probabilistic computational complexity of BN with increasing the number of states for dynamic nodes should be considered.

References

1. Kumamoto, H., Henley, E.J.: Probabilistic risk assessment and management for engineers and scientists. IEEE press, New York (1996)
2. Manian, R., Dugan, J.B., Coppit, D., Sulivan, K.: Combining various solution techniques for dynamic fault tree analysis of computer systems. In: Proc. 3rd Intl high-assurance systems engineering symposium, Washington D.C., pp. 21–28 (1998)
3. Barlow, R., Proshan, F.: Mathematical theory of reliability. SIAM, Philadelphia (1996)
4. Andrews, J.D., Dunnett, S.J.: Event tree analysis using binary decision diagrams. IEEE Transactions on Reliability 49, 230–238 (2000)
5. Boudali, H., Dugan, J.B.: A discrete-time Bayesian network reliability modeling and analysis framework. Reliability Engineering and System Safety 87, 337–349 (2005)
6. Jensen, F.V.: An introduction to Bayesian Networks. Springer, New York (1996)
7. Pearl, J.: Probabilistic reasoning in intelligent systems. Morgan Kaufman, California (1988)

8. Bobbio, A., Portinale, L., Minichino, M., Ciancamerla, E.: Improving the Analysis of dependable Systems by Mapping Fault Trees into Bayesian Networks. Reliability Engineering and System Safety 71, 249–260 (2001)
9. Dutuit, Y., Rauzy, A.: A linear time algorithm to find modules of fault trees. IEEE Transactions on Reliability 45, 422–425 (1996)
10. Sullivan, K.J., Dugan, J.B., Coppit, D.: The Galileo fault tree analysis tool. In: 29th Annual International Symposium on Fault-Tolerant Computing, pp. 232–235 (1999)
11. Jensen, F.V., Chamberlain, B., Nordahl, T., Jensen, F.: Analysis in Hugin of data conflict. In: Bonissone, P.P., Henrion, M., Kanal, L.N., Lemmer, J.F. (eds.) Uncertainty in Artificial Intelligence, vol. 6, pp. 519–528. Elsevier Science Publishers, Amsterdam, The Netherlands (1991)

Component Fault Tree Analysis Resolves Complexity: Dependability Confirmation for a Railway Brake System

Reiner Heilmann, Stefan Rothbauer, and Ariane Sutor

Siemens Corporate Technology
Otto-Hahn-Ring 6, 81730 München, Germany
reiner.heilmann@siemens.com, stefan.rothbauer@siemens.com,
ariane.sutor@siemens.com

Abstract. In 2006 Siemens Transportation systems had to obtain an operating license for the brake system of a newly developed train. Therefore a safety analysis for the brake system had to be performed to show that the probability of a failure of the brakes is sufficiently small, less than specified limits. The safety analysis was performed by Siemens Corporate Technology. The probability of a failure of the brake system was calculated using hierarchical fault tree analysis. The large number of different combinations of subsystems contributing to failure scenarios was managed by a specially developed program for automatic generation of combinatorial fault trees. The most important result was the proof of the quantitative safety targets of the brake system to the regulating body.

Keywords: Hierarchical Fault Trees, Combinatorial Fault Trees, Railway Brake System, Practical Example.

1 Introduction - Safety of a Railway Brake System

In order to prevent people and the environment from harm, it is required to establish that safety relevant systems meet their respective safety targets. This is especially challenging for newly developed technical systems innovating existing technical concepts. In 2006 Siemens Transportation systems had to obtain an operating license for the brake system of a newly developed train. Therefore a safety analysis for the brake system had to be performed to show that the probability of a failure of the brakes is sufficiently small, less than specified limits.

Train brake systems consist of many individual brakes of different types. The different kinds of brakes are contributing to the total brake force to varying degrees in accordance with the brake scenarios that stem from the different requirements trains are facing. Consider for example the requirements of an emergency stop caused by a passenger vs. a controlled stop of the train arriving at the station or the brake force applied to prevent the train from moving when stationed at the platform.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 100–105, 2007.

To increase the availability of the brake system and therefore enable a safe operation of the train, the total amount of brake force that the brake system is able to generate is more than 100 percent of the specified brake force of these scenarios. Therefore the failure of a certain number of individual brakes is tolerable as long as the brake force that the remaining brakes can generate is sufficient. Furthermore the train control system is able to observe which and how many individual brakes are functioning correctly. On this basis progressively more restricting measures (e. g. speed limits, kind of operation possible) may be chosen to ensure that the train is operated safely. When considering these facts, it is not surprising that the number of failure combinations which do not lead to a complete failure of the brake system is large.

In this report we describe how for a practical, not needlessly restrictive safety concept an analysis of the failure scenarios was carried out. The safety analysis was performed by Siemens Corporate Technology. The probability of a failure of the brake system was calculated using fault tree analysis. However, the special structure of the system required adequate methods to resolve the complexity for the analyses. Therefore instead of classical fault trees a new approach with hierarchical component fault trees was chosen. Finally the large number of different combinations of subsystems contributing to failure scenarios was managed by a specially developed program for automatic generation of combinatorial fault trees.

2 Traditional vs. Hierarchical Component-Based Fault Trees

FTA is an analysis technique for safety and reliability aspects that uses a graphical representation to model causal chains leading to failures. FTA has been used since the 1960s and is defined in international standards ([1], [2]). It uses a top-down approach, starting off at the undesired failure (TOP-event) and tracing its causes down to the necessary level of detail. The various conditions that have to be met for the TOP-event to happen are modelled accordingly.

As modern technical systems become more and more complex, fault trees need to be partitioned both for editing and for efficient computer analysis. Principally there are two principles for partitioning:

1. The backward refinement of the cause-effect relations as indicated by the tree.
2. The refinement by architectural components ([3]).

Traditionally the first principle is used in fault tree development, thereby the system is partitioned into independent fault trees. However some events - so called repeated events - can have effects on a number of other components, e. g. a power unit break down might influence the main CPU as well as the auxiliary CPU. It is important to know that most calculation rules for the probabilistic analysis depend on the assumption that all events are stochastically independent of each other. Therefore repeated events cannot be displayed correctly using

simple trees. Apart from repeated events, the fact that Fault Trees contain only one top-event ([2]) is also a restriction. In practice, it is often important to analyze cause-effect relations between various top-events that represent different failure modes of the same technical component.

A generalization of traditional fault trees that provides a solution to this problem leads to the notion of Directed Acyclic Graphs (DAGs) or Cause Effect Graphs (CEGs). Repeated events are only displayed once and linked correctly to all other components on which they have influence. Furthermore the CEGs can be used to follow the second refinement approach according to the physical structure of the system.

A further extension of CEGs is the concept of hierarchical fault trees ([4]). Here the different (physical) subsystems of the system are modelled separately. On the system level they are connected using input- and output-ports. Usually a quantitative evaluation of a component alone is not possible. Only on the system level the whole model becomes a CEG. Hierarchical fault trees can be evaluated using standard algorithms, e.g. BDD algorithms ([5]).

In practice the structuring of the system according to its physical subsystems has a huge advantage as it strongly supports the understanding of system developers.

3 Combinatorial Fault Trees

A wide variety of technical systems contains several equal subsystems. To increase the system's reliability and availability, often redundant subsystems are added besides the minimum required number. Further more usually more than one subsystem is involved in the performance of a system function. If multiple subsystems with redundancies are involved in a system function, a fault tree analysis of a failure of this function becomes very complicated due to the various possible combinations of working and defective subsystems, although the actual amount of different components may be manageable.

To resolve the complexity of fault tree analyses arising from a large number of different redundant components, we developed a special approach. Each actual physical component is modelled manually using component fault trees. The various possible combinations of working and defective components are generated automatically by a special designed computer program. For each combination the resulting performance of the desired function is computed by multiplying the working components with their degree of performance and then adding all of the products. The result is the degree to which the function is performed by the considered combination of working components.

In order to calculate the probability of a special scenario, all combinations which contribute to this scenario are ported into the fault tree tool using a tool with XML-interface. Each combination is represented by $m : n$-gates, one gate for each physical component type. Thereby n is the number of occurrences of the component type in the train and m is the number of working components

in this combination. For each combination the $m : n$-gates representing the different physical types of components are connected via an AND-gate. All AND-gates representing one combination then are connected via an OR-gate. This final OR-gate represents the considered scenario.

For our calculations we wrote a special computer program which generates all necessary gates and edges of the fault tree for our considered scenarios. The probability of the top-event of the fault tree is calculated using the normal functionality of the tool.

We give a simple example to describe our proceeding. A simplified railway brake system may consist of three different types of brakes. There are different numbers of brakes of the different types and they add differently to the brake force of the entire train. The total amount $\#Comb$ of different combinations of

Table 1. Brake subsystems

Brake subsystem type	Number of subsystems of this type	Contribution of single subsystem to brake force
1	4	10 %
2	5	5 %
3	5	7 %

working and defective subsystems without distinguishing between subsystems of the same type is

$$\#Comb = 5 \cdot 6 \cdot 6 = 180 \tag{1}$$

We consider the example that we have to calculate the probability that the brake force of the entire train is down to 50%. For this task all failure combinations of subsystems exhibiting 50% brake force have to be determined. For each combination the corresponding gates and edges have to be generated. Then all combinations are combined via an OR-gate in a fault tree with the top event "brake force down to 50%". The following table contains all combinations of working brake subsystems which together exhibit 50%. The result as presented in Table 2 can now be used to generate the corresponding fault tree. For every scenario describing reduced brake force the starting point of the analysis is the

Table 2. Combinations of working subsystems with 50 % total brake force

Working subsystems			Brake force
type 1	type 2	type 3	
0	3	5	50%
1	1	5	50%
3	4	0	50%
4	2	0	50%

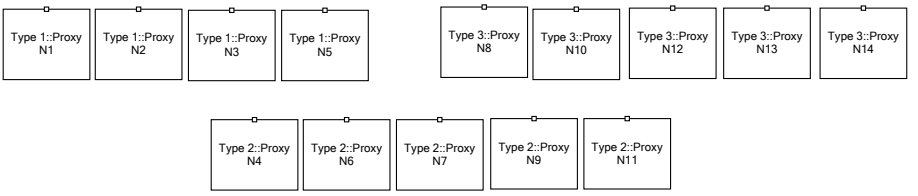


Fig. 1. Incomplete Fault Tree "Brake System"

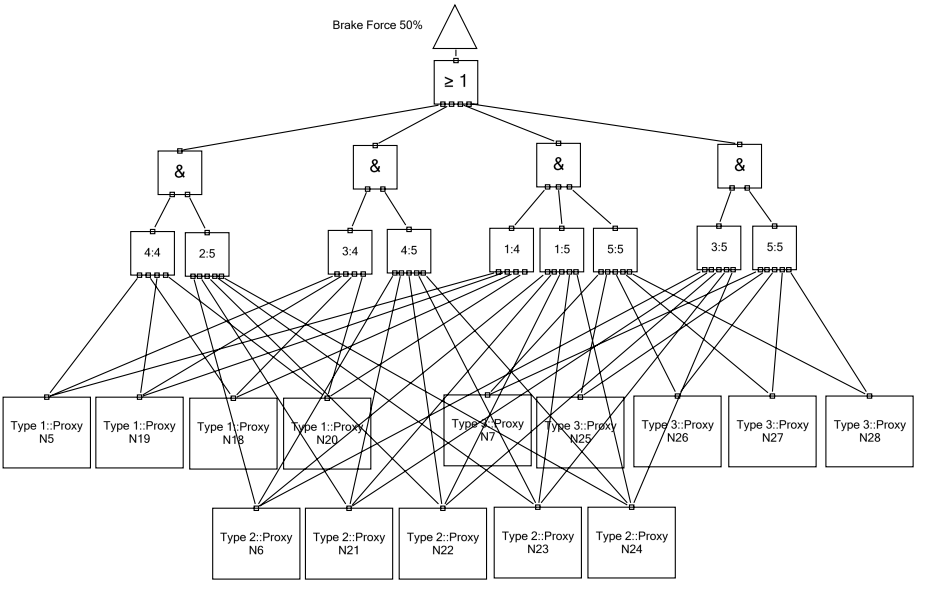


Fig. 2. Incomplete Fault Tree "Brake System"

incomplete fault tree displayed in Fig. 1. Each box in Fig. 1 contains the fault tree of one brake subsystem. Every physical component is represented by one box. We now generate the complete fault tree describing the brake force scenario automatically by reading the relevant combinations and transferring them into the fault tree tool via an XML-interface. Fig. 2 shows the result for the example brake system exhibiting 50% brake force. The fault tree is evaluated using the normal probability calculation of the tool. Due to the many combinations which can occur at practical examples it is important to use fast algorithms for the computations. There are efficient BDD algorithms which are able to handle even a large number of basic events ([3]) which also led to satisfactory results during our analysis of the railway brake system.

4 Results, Benefits Obtained and Conclusions

Using the approaches described above the brake system was modelled. After the definition of the possible failure scenarios within the lifetime of the train, the fault trees for the failure combinations were generated and evaluated automatically. By these means it was easily possible to obtain the probabilities that the trains brake system is able to provide a desired percentage of its maximum brake force. The benefits of being able to obtain these probabilities are wide-ranging:

1. Safety, reliability and availability characteristics of the brake system were obtained in a very efficient manner.
2. Strategies for a graceful degradation concept could be obtained on a reasonable basis.
3. Proof of the quantitative safety targets of the brake system could be shown to the regulating body.

Based on these results it was possible to establish a graduated approach to increase the reliability and availability of the trains brake system. The methodology used and results obtained were approved by the responsible Railway Operator. In addition to the benefits that DAGs provide, combinatorial fault trees will in praxis further enable engineers to manage the complexity of failure scenarios of technical systems based on multiple redundancy. Using combinatorial fault trees failure scenarios that up to now would not have been manageable and would therefore have to be simplified - usually leading to worse results - can be resolved and provide a real advantage in applications.

References

1. DIN 25424 Fehlerbaumanalyse (Fault Tree Analysis), German Industry Standard (Part 1 & 2) Beuth Verlag, Berlin (1981/1990)
2. IEC 61025 Fault Tree Analysis, International Standard IEC 61025. IEC, Geneva (1990)
3. Mäckel, O., Rothfelder, M.: Challenges and Solutions for Fault Tree Analysis Arising from Automatic Fault Tree Generation: Some Milestones on the Way. In: ISAS-SCI(I) 2001, pp. 583–588 (2001)
4. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new Component Concept for Fault Trees. In: 8th Australian Workshop on Safety critical Systems and Software, Canberra, vol. 33 (2003)
5. Coudert, O., Madre, J.C.: Fault tree analysis: 10^20 prime implicants and beyond. In: Proceedings of the Annual Reliability and Maintainability Symposium, Atlanta GA, pp. 240–245 (1993)

Compositional Temporal Fault Tree Analysis

Martin Walker, Leonardo Bottaci, and Yiannis Papadopoulos

Department of Computer Science, University of Hull, UK
m.d.walker@dcs.hull.ac.uk, l.bottaci@hull.ac.uk,
y.i.papadopoulos@hull.ac.uk

Abstract. HiP-HOPS (Hierarchically-Performed Hazard Origin and Propagation Studies) is a recent technique that partly automates Fault Tree Analysis (FTA) by constructing fault trees from system topologies annotated with component-level failure specifications. HiP-HOPS has hitherto created only classical combinatorial fault trees that fail to capture the often significant temporal ordering of failure events. In this paper, we propose temporal extensions to the fault tree notation that can elevate HiP-HOPS, and potentially other FTA techniques, above the classical combinatorial model of FTA. We develop the formal foundations of a new logic to represent event sequences in fault trees using Priority-AND, Simultaneous-AND, and Priority-OR gates, and present a set of temporal laws to identify logical contradictions and remove redundancies in temporal fault trees. By qualitatively analysing these temporal trees to obtain ordered minimal cut-sets, we show how these extensions to FTA can enhance the safety of dynamic systems.

Keywords: temporal fault trees, formal FTA, automated FTA, fault tree synthesis, formal safety analysis.

1 Introduction

Fault Tree Analysis (FTA) is a safety analysis technique first used in the 1960s, and since then it has been used in a number of different areas, including the aerospace, automobile, and nuclear industries. However, despite the improvements it has received over the years, it still suffers from a number of problems. One major problem is that although the analysis of fault trees has long been automated, the actual production (or *synthesis*) of fault trees has remained a manual process.

Recently, work has been directed towards addressing this problem by looking at the potential integration of design and safety analysis. In this work, fault trees are automatically produced from system models that contain information about component failures and their effects. Techniques developed to support this concept include HiP-HOPS [1] and Components Fault Trees (CFT) [2]; both support assessment processes in which composability and reuse of "component safety analyses" across applications becomes possible. In HiP-HOPS, a topological model of a system together with annotated failure data for each component is used to produce a set of fault trees and a FMEA (Failure Modes and Effects Analysis) for the system. Instead of forcing analysts to produce entire fault trees, they can focus on

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 106–119, 2007.

local failure behaviour, and HiP-HOPS then takes that focused information and shows how local failures propagate through the system to cause system-wide faults. This compositional approach also has the benefit of speeding up fault tree synthesis to a matter of seconds, so that it becomes practical to perform multiple iterations of fault tree analysis to investigate different variations on a design and to see the impact different design changes have on the system safety.

In spite of these benefits, HiP-HOPS, like traditional FTA, struggles to accurately model systems in which the failure behaviour is dependent on the sequence or timing of events. In these cases, the analyst has to work around the limitations in FTA, e.g. by representing the temporal requirements as separate events or by including them in the event descriptions themselves, but in doing so, the temporal information is effectively hidden. If the temporal information is not present in the logical structure of the fault tree, then it is unable to play a role in qualitative analysis.

One solution to this problem is Pandora [3], a recent extension to FTA and HiP-HOPS that extends the vocabulary of the fault tree with a small set of new temporal gates without altering the familiar structure of the fault tree. These new gates allow FTA to represent sequences of events and take into account any potential redundancies or contradictions that arise as a result.

In this paper, we present the formal logic that underpins Pandora (etymology: Hour or "time" - *ora* [*ώρα*] in Greek - of *Pand* gates), and show how it can be used to more effectively analyse dynamic systems. Firstly, we briefly discuss similar approaches, and then we describe Pandora itself. We also give a small example system to show how Pandora is used and finally we present our conclusions. Although much of the discussion is focused on how Pandora is usefully integrated within HiP-HOPS to enable compositional temporal FTA of dynamic systems, the principles presented in the paper are generic and directly applicable to conventional and other compositional FTA techniques, e.g. CFTs.

2 Background

An early solution to the problem of representing time and event sequences in fault trees was the "Priority-AND" (PAND) gate [4]. According to the *Fault Tree Handbook* [5], "the PRIORITY AND-gate is a special case of the AND-gate in which the output event occurs only if all input events occur in a specified ordered sequence." Unfortunately, this definition is not sufficiently precise in that it is not clear how the PAND gate behaves if no order is specified or if the inputs occur at the same time; nor does it specify whether the events have a duration, and if so, whether they can overlap. When it is used at all, the PAND gate is often treated as a simple AND gate in qualitative analysis. This ignores the potential for contradictions and redundancies that the PAND gate introduces, e.g. (X PAND Y) AND (Y PAND X) specifies that X precedes Y and that Y precedes X, which is obviously impossible, and yet (X AND Y) AND (Y AND X) is valid. For quantitative analysis, there are a number of different methods available [6], but without prior qualitative analysis, there is still the risk of obtaining invalid results.

More recent solutions include the Dynamic Fault Tree (DFT) methodology [7], which introduces a set of new dynamic gates to the fault tree, such as Spare gates,

Functional Dependency gates, and even the old PAND gates. Dynamic fault trees are quantitatively analysed using either the static BDD (Binary Decision Diagram) method or dynamic Markov analysis as appropriate. There has also been a recent proposal for performing qualitative analysis on DFTs using a variation on the BDD approach [8], but this explicitly separates the temporal and logical constraints until after the analysis is complete, potentially missing earlier opportunities for removing redundancies and contradictions during the analysis.

Temporal Fault Trees (TFT) [9] are another proposed solution. TFTs introduce a larger set of new temporal gates to the fault tree to represent various aspects of a new temporal logic (called PLTLP), including gates like "within", "sometime", and "prev". TFTs also come with algorithms for both quantitative and qualitative analysis, but they are intended to be used after the system has failed, in conjunction with the system log, as a diagnostic aid.

A much more compact solution is the AND-THEN (or TAND) gate [10], which is designed to be a building block that can represent more complex expressions. The TAND gate (indicated by Π) represents a situation where one event immediately follows the cessation of another event, so for example, to specify that Y must be true at some time after X becomes false but not immediately after, it is necessary to write X.¬Y Π ¬X.¬Y Π ¬X.Y. However, the TAND suffers from certain problems relating to its definition; in particular, it treats events more like states, and it also requires the use of NOT gates to build more complex expressions. This can lead to non-coherent fault trees, since the non-occurrence or cessation of a fault can potentially lead to the system failure, and thus the necessity for more complex analysis algorithms due to the need to represent both events and their complements [11].

A slightly different style of approach is taken by both Gorski and Wardzinski [12] and Hansen and Ravn [13], who represent timing constraints as a separate event in a more formalised fault tree structure. However, this is intended for the representation of specific real-time requirements, rather than to enable temporal qualitative analysis of fault trees. Because it expresses the temporal information as a new event rather than as a gate, that information is not part of the fault tree structure, and so the potential for removing contradictions and redundancies is lost.

Although each of these solutions is well-suited for its intended tasks, whether it be the representation of real-time requirements in fault trees or the quantitative analysis of redundant and standby components, we believe that there is still scope for a solution which offers a general and flexible approach without substantially altering the existing semantics and structure of the fault tree.

3 Pandora

Pandora allows for the general representation and analysis of event sequences without introducing many new gates or overcomplicating the process of fault tree analysis with complex new semantics. To that end, it is based around a redefinition of the long-established Priority-AND gate; the aim is to remain as close as possible to the original philosophy of simplicity and flexibility that makes FTA popular, while also solving the ambiguities that plagued the original PAND gate.

3.1 Events and Events Orderings

The Fault Tree Handbook states that for the purposes of FTA, it is the *occurrence* of faults that matters, i.e. the transition of a component to a faulty state at a certain point in time, rather than the existence of the faulty state. Pandora thus avoids the complexities of formalising time by focusing solely on events and their temporal relationships. It is based on one simple principle: like the original PAND gate, it is designed to allow the *temporal ordering of events* to be represented as part of the fault tree structure itself. Because we are interested only in the moment at which a fault occurs, we do not need to consider the duration of the fault, and instead we assume that once a fault has occurred, it persists; if necessary, the cessation of a fault (i.e. the disappearance of a fault after it has occurred) could be modelled separately as an another event. This assumption fits with the original definition in the *Fault Tree Handbook*, which states that "From the standpoint of constructing a fault tree we need only concern ourselves with the phenomenon of occurrence. This is tantamount to considering all systems as non-repairable" and "Under conditions of no repair, a fault that occurs will continue to exist." (p V-1).

In Pandora, the occurrence of an event, e.g. the transition to a faulty state, is instantaneous and can occur at most once. Therefore, given the occurrence of two distinct events X and Y, either:

- X occurs before Y
- X occurs at the same time as Y
- X occurs after Y, or equivalently, Y occurs before X

Rather than working with the date of occurrence of an event to determine which of these relationships hold, Pandora uses a more abstract approach by taking into account only the *order* of events. In Pandora, the PAND gate is used to represent "before", e.g. X PAND Y means X occurs before Y, and a "Simultaneous-AND" (or SAND) gate represents "at the same time as", i.e. X SAND Y means X occurs at the same time as Y. There is also a third gate, "Priority-OR" (or POR), that represents the situation in which X has to occur before Y if Y occurs, e.g. X POR Y means that either X occurs and Y does not, or X occurs before Y. The inclusion of both PAND and SAND gates is important to ensure that we can represent all three temporal relationships without overlap; other approaches either omit a SAND gate (which leads to a question of what happens if two events occur simultaneously, e.g. if they share a common cause), or, as in the case of the TAND approach, redefine the existing Boolean AND gate to match the definition of the SAND.

Conventional fault tree analysis using the AND and OR operators allows the specification of combinations or sets of basic events. Pandora adds the ability to specify sequences of sets of events. A sequence of sets of events, called an event ordering, is fundamental to the Pandora logic. For example, given three basic events, X, Y and Z, there are a number of possible event orderings. The empty sequence $\langle\rangle$ is the empty event ordering in which no events have occurred, $\langle\{X, Y\}\rangle$ is an event ordering in which both X and Y occurred simultaneously (this applies to all events in the same set in an event ordering), and $\langle\{X\}\{Y, Z\}\rangle$ is an event ordering in which X occurred first followed by the simultaneous occurrence of Y and Z. Thus, given a set

of basic events E, an event ordering o contains a subset of E that specifies not only which basic events have occurred, but also the sequence in which they occurred. An event cannot occur more than once in an event ordering, which prevents an event from preceding itself, and there are no empty event sets in an event ordering.

The set of all possible event orderings for a given set of events E is denoted $o(E)$, or in formal notation:

$$o(E) = \{\langle\rangle\} \cup \{o : \langle A_1, A_2, \ldots A_n\rangle\}$$
$$\text{where } A_i \neq \varnothing \wedge A_1 \cup A_2 \cup \ldots \cup A_n \subseteq E \wedge A_i \cap A_j = \varnothing,\ i \neq j, \text{ and } i, j \in 1 \ldots n.$$

A basic event, e.g. X, occurs in o if o contains a basic event set containing X, i.e. $X \in o \Leftrightarrow \exists A{:}\mathbf{P}(E) \bullet A \in o \wedge X \in A$ where $\mathbf{P}(E)$ is the power set of E. Similarly, two basic events, e.g. X and Y, occur at the same time in the event ordering o if o contains a set A which contains both X and Y, i.e. $\exists A \in o \bullet X \in A \wedge Y \in A$.

Event orderings themselves can be ordered as a precedence tree as shown in Fig. 1, which shows the precedence tree of the event orderings o({X, Y, Z}). The empty event ordering $\langle\rangle$ precedes all other event orderings, and successive event orderings are formed by appending sequences of sets of simultaneous events. Given an event ordering o, $pre(o)$ is the set of event orderings that precede o in the precedence tree, so for example $pre(\langle\{X\}\{Y, Z\}\rangle)$ contains $\langle\{X\}\rangle$ and $\langle\rangle$.

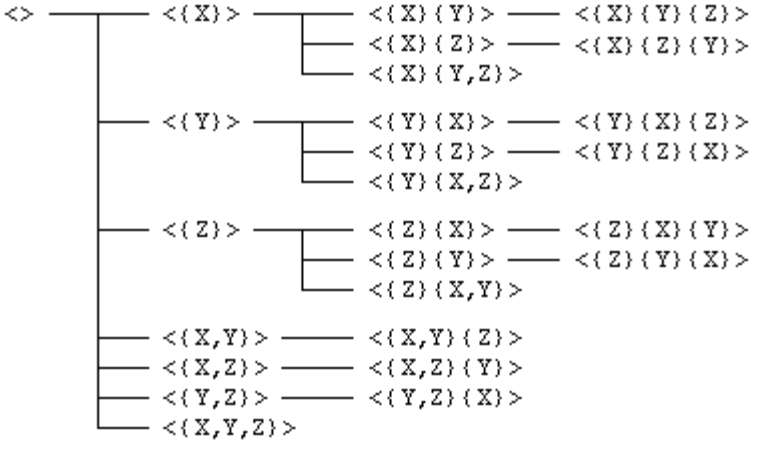


Fig. 1. The precedence tree for $o(\{X, Y, Z\})$

Each leaf of the precedence tree is a complete event ordering in that it contains every basic event. At non-leaf nodes there are incomplete event orderings that may be extended to produce "future" event orderings. The set of possible future event orderings for an incomplete event ordering o is denoted by $fu(o)$. Clearly, if an event occurs in o, it must also occur in all of $fu(o)$, and similarly, if o does not include an event, then nor do any of the event orderings in $pre(o)$.

3.2 Non-basic or Intermediate Events

A non-basic or intermediate event occurs when other combinations or sequences of events occur and is defined by a Boolean expression containing Boolean operators (gates). In order to apply any Boolean operator to events, events must be interpreted as being true or false. This is done conventionally by speaking of "X is true" to mean that X occurs and that "X is false" to mean that X does not occur. To avoid confusion between a basic event such as X and Boolean expressions about the occurrence of X, Pandora uses lower case italics, e.g. *x*, for Boolean expressions. In Pandora, Boolean expressions involving events are called *event expressions*. Basic event expressions are event expressions that contain no operators (gates). There is a 1-1 correspondence between basic events and basic event expressions; so for example, if X is a basic event, then there is a basic event expression *x* that is true of any event ordering in which X occurs. More generally, an event expression is a Boolean function on the set of event orderings $o(E)$. Note that an event expression such as *x*, which is true of some event orders in $o(E)$, implicitly defines a subset of $o(E)$, i.e. that set of event orderings for which *x* is true. For this reason, it is common to abbreviate the set of event expressions, i.e. the set of functions $o(E) \rightarrow \{true, false\}$, as the power set of event orderings, i.e. $\mathbf{P}\ o(E)$. The informal statement, "the event *x* occurs in *o*" is therefore defined formally as "the set of event orderings *x* contains the event order *o*".

Non-basic event expression are Boolean expressions that contain Boolean operators or gates. Pandora uses five gates: the conventional AND (.) and OR (+) gates, and three temporal gates, PAND (<), SAND (&) and POR (|), where the gate symbol has been shown in parentheses following the gate name. In event expressions, the operator precedence is, from highest to lowest, & (SAND), < (PAND), | (POR), . (AND), and finally + (OR). The following rules determine the occurrence of intermediate events for each of the five operators. The informal definition is given above the line and the definition in the Pandora formal semantics is given below the line. *x* and *y* are event expressions and therefore members of $\mathbf{P}\ o(E)$.

The event $x + y$ occurs in $o \Leftrightarrow$ x occurs in o or y occurs in o

$+ : \mathbf{P}\ o(E) \times \mathbf{P}\ o(E) \rightarrow \mathbf{P}\ o(E)$

$\forall x, y : \mathbf{P}\ o(E) \bullet o \in x + y \Leftrightarrow o \in x \vee o \in y$

The event $x\ .\ y$ occurs in $o \Leftrightarrow$ x occurs in o and y occurs in o

$. : \mathbf{P}\ o(E) \times \mathbf{P}\ o(E) \rightarrow \mathbf{P}\ o(E)$

$\forall x, y : \mathbf{P}\ o(E) \bullet o \in x\ .\ y \Leftrightarrow o \in x \wedge o \in y$

The event $x\&y$ occurs in $o \Leftrightarrow$ x and y in o and x occurs at the same time as y

$\& : \mathbf{P}\ o(E) \times \mathbf{P}\ o(E) \rightarrow \mathbf{P}\ o(E)$

$\forall x, y : \mathbf{P}\ o(E) \bullet o \in x\ \&\ y \Leftrightarrow o \in x \wedge o \in y \wedge \forall\ r : pre(o) \bullet r \in x \Leftrightarrow r \in y$

In the above definition, *x* is specified to occur at the same time as *y* in *o* by specifying that any event ordering in $pre(o)$ in which *x* occurs, *y* also occurs and vice versa.

The event $x<y$ occurs in $o \Leftrightarrow$ x and y occur in o and x occurs before y

$< : \mathbf{P}\, o(E) \times \mathbf{P}\, o(E) \rightarrow \mathbf{P}\, o(E)$

$\forall x, y : \mathbf{P}\, o(E) \bullet o \in x < y \Leftrightarrow o \in x \wedge o \in y \wedge \exists\, r : pre(o) \bullet r \in x \wedge r \notin y$

In the above definition, x is specified to occur before y in o by specifying that for some event ordering in $pre(o)$, x occurs but y does not.

The event $x|y$ occurs in $o \Leftrightarrow$ x occurs in o and x occurs before y if y occurs

$| : \mathbf{P}\, o(E) \times \mathbf{P}\, o(E) \rightarrow \mathbf{P}\, o(E)$

$\forall x, y : \mathbf{P}\, o(E) \bullet o \in x \mid y \Leftrightarrow o \in x \wedge \exists\, r : pre(o) \cup \{o\} \bullet r \in x \wedge r \notin y$

In the above definition, x is specified to occur in o before y or without y in o by specifying that for some event ordering in $pre(o)$ or o itself x occurs y does not.

3.3 Truth Tables

These five semantic rules for the five operators can be illustrated in a truth table containing the event orderings on two events X and Y, with 1 representing *true* and 0 representing *false*:

Table 1. Truth table demonstrating the five operators in Pandora

ordering	x	y	$x + y$	$x\,.\,y$	$x\|y$	$x<y$	$x\&y$	$y<x$	$y\|x$
<>	0	0	0	0	0	0	0	0	0
<{X}>	1	0	1	0	1	0	0	0	0
<{Y}>	0	1	1	0	0	0	0	0	1
<{X}{Y}>	1	1	1	1	1	1	0	0	0
<{Y}{X}>	1	1	1	1	0	0	0	1	1
<{X, Y}>	1	1	1	1	0	0	1	0	0

The truth table size increases rapidly with the number of basic events; for three events, the truth table would require 26 rows (see the number of nodes in the precedence tree of Fig. 1). It possible, however, to produce a more compact table by exploiting the property that event expressions are *monotonic* with respect to the set $fu(o)$ of future event orderings, i.e. if x occurs in o then it occurs in all $fu(o)$. In terms of the precedence tree of event orderings, every node below a true node must also be true. Compact truth tables list not event orderings but paths in the event precedence tree. Each path is indicated by the complete event ordering that terminates the path. For a given event expression, the entry in the compact truth table for a given path is an integer rather than a Boolean value. The integer indicates the position of the first event ordering in the path for which the given event expression is true. For example, the path to $\langle\{X\}\{Y\}\rangle$ has two predecessor orderings - $\langle\{X\}\rangle$ and $\langle\rangle$. If x means X occurs in o, then x is true in the second event ordering on this path; counting from 0, this event order is therefore indicated by 1. If y means Y occurs in o then the entry for $x\,.\,y$ on this path would be 2. Zero indicates that the event expression is not true for any event orderings in the given path. Because integers in compact truth tables indicate positions in paths of event orderings, ordered by temporal relations, we refer

Table 2. A temporal truth table demonstrating the five operators in Pandora

	x	y	$x + y$	$x \,.\, y$	$x\|y$	$x<y$	$x\&y$	$y<x$	$y\|x$
<{X}{Y}>	1	2	1	**2**	1	**2**	0	0	0
<{Y}{X}>	2	1	1	**2**	0	0	0	**2**	1
<{X, Y}>	1	1	1	**1**	0	0	**1**	0	0

to compact truth tables as Temporal Truth Tables (TTTs). Table 2 is the TTT equivalent to Table 1. Note that the temporal truth table entries for $x + y$ are the minimum of those of x and y and the entries for $x \,.\, y$ are maximum of x and y.

Two event expressions are equivalent if they are true of exactly the same set of event orderings, and this can be seen by comparing their columns in a truth table. For example, Table 2 shows the equivalence between $x + y$ and $x|y + x\&y + y|x$ as well as the equivalence between $x \,.\, y$ and $x<y + x\&y + y<x$ – for every value in the $x \,.\, y$ column, there is a corresponding value in one of $x<y$, $x\&y$, or $y<x$ (highlighted in bold), and similarly for $x + y$.

3.4 Temporal Laws

The purpose of qualitative analysis of fault trees is to obtain minimal cut sets (MCS). A MCS is a conjunctive set of basic events which are precisely sufficient to cause the top event of the fault tree to occur, i.e. if all events in a MCS occur, so will the top event. Qualitative analysis produces a disjunctive set of MCSs, so that the occurrence of all events in any one MCS is sufficient to cause the top event. In traditional analysis, the MCSs are obtained by applying Boolean laws (such as the Absorption Laws and Distributive Laws) to simplify the logical expression representing the fault tree until it is in minimal form.

In Pandora, the situation is complicated by the presence of the three temporal gates. We seek to define a *cut sequence*, analogous to a cut set, that contains temporal operators, and a *minimal cut sequence* (MCSQ), analogous to a minimal cut set, containing no redundancies or contradictions. The conventional laws for the Boolean operators AND and OR remain valid, but additional laws are required to simplify expressions that contain temporal operators. Many of these are temporal analogues to the well known Boolean laws; for example, there are Absorption laws that apply to PAND gates: ($x<y + x \Leftrightarrow x$ and $x<y \,.\, x \Leftrightarrow x<y$), POR gates ($x|y + x \Leftrightarrow x$ and $x|y \,.\, x \Leftrightarrow x|y$), and SAND gates ($x\&y + x \Leftrightarrow x$ and $x\&y \,.\, x \Leftrightarrow x\&y$), and also Distributive expansion laws (e.g. $x<(y + z) \Leftrightarrow x|y \,.\, x|z \,.\, (y + z)$ and $(x + y)|z \Leftrightarrow x|z + y|z$). Other laws may be used to detect contradictions and eliminate redundancies in expressions that contain only temporal gates. There is insufficient space to consider all of these laws in depth, but below is a brief subset containing some of the most useful laws for reduction.

Absorption

$x \,.\, (x<y) \Leftrightarrow x<y$	$x \,.\, (x\&y) \Leftrightarrow x\&y$	$x \,.\, (x\|y) \Leftrightarrow x\|y$
$y \,.\, (x<y) \Leftrightarrow x<y$	$y \,.\, (x\&y) \Leftrightarrow x\&y$	$y \,.\, (x\|y) \Leftrightarrow x<y$
$x + (x<y) \Leftrightarrow x$	$x + (x\&y) \Leftrightarrow x$	$x + (x\|y) \Leftrightarrow x$
$y + (x<y) \Leftrightarrow y$	$y + (x\&y) \Leftrightarrow y$	$y + (x\|y) \Leftrightarrow x + y$

Mutual Exclusion

$x\&y \;.\; x<y \Leftrightarrow 0$	$x<y \;.\; y<x \Leftrightarrow 0$	$x\|y \;.\; y\|x \Leftrightarrow 0$
$x\&y \;.\; x\|y \Leftrightarrow 0$	$x<y \;.\; y\|x \Leftrightarrow 0$	$x\|y \;.\; y<x \Leftrightarrow 0$

Simultaneity

$x\&x \Leftrightarrow x$	$x<x \Leftrightarrow 0$	$x\|x \Leftrightarrow 0$

The laws given above can be formally proven using one of two methods: a truth table may be used to verify that each expression is true of exactly the same event orderings, or alternatively, a proof by deduction can be done using the conventional proof rules of Predicate logic. That this is possible is clear from the formal definitions of the temporal gates given earlier in that each definition allows expressions containing temporal gates to be rewritten as expressions in Predicate logic. An unpublished technical report containing proofs of a number of laws is available from the authors.

3.5 Base Temporal Form

One approach towards simplification of the Boolean expressions that contain temporal operators is to rewrite the expression into an equivalent form in which all temporal operators are applied to basic event expressions only, i.e. the temporal operators occur only immediately above the leaf expressions of the resulting fault tree. For example, $(x\&y)<z$ is equivalent to $x\&y \;.\; x<z \;.\; y<z$, and in the case that x, y and z are basic event expressions, the temporal operators are restricted to the lowest levels of the fault tree. The resulting event expressions, consisting of two basic event expressions and a temporal operator, are called *doublets*, and are indicated by brackets, i.e. $[x\&y] \;.\; [x<z] \;.\; [y<z]$. Above the doublet level of the fault tree, the tree contains only AND and OR gates and so, providing the doublets are treated as atomic event expressions, may be analysed as a non-temporal fault tree. This makes it possible to treat a cut sequence as a normal cut set – a conjunctive set of events – albeit with a greater range of reduction methods able to be applied.

An expression in which all the temporal operators are within doublets is said to be in *base temporal form* (BTF). It is possible to prove that any temporal fault tree may be reduced to BTF. The proof is by induction on the height of temporal subtrees within the overall fault tree. Let the height of a basic event expression be 0 and the height of an intermediate event expression be one greater than the maximum height of either of its operands, then for an expression to be in BTF, the height of any temporal event expression should be 1. Consider a fault tree in which the height of the tallest temporal expression is greater than 1. Any law that distributes a temporal operator over AND and OR operators will "push" the temporal operator further down the fault tree, i.e. it produces an equivalent tree in which the height of temporal expression is reduced. By repeatedly applying such laws, a fault tree can be reduced to BTF. There are sufficient distributive laws to reduce all the temporal expressions (again, an unpublished technical report listing these laws is available from the authors). Some examples are shown below:

$x<(y + z) \Leftrightarrow x\|y \;.\; x\|z \;.\; (y + z)$	$(y + z)<x \Leftrightarrow (y<x) + (z<x)$
$x<(y \;.\; z) \Leftrightarrow y \;.\; (x<z) + z \;.\; (x<y)$	$(y \;.\; z)<x \Leftrightarrow (y<x) \;.\; (z<x)$

$x\&(y + z) \Leftrightarrow x\&y\&z + x\&y|z + x\&z|y$ $\quad (y + z)\&x \Leftrightarrow x\&y\&z + x\&y|z + x\&z|y$
$x\&(y \,.\, z) \Leftrightarrow y<x\&z + z<x\&y + x\&y\&z$ $\quad (y \,.\, z)\&x \Leftrightarrow y<x\&z + z<x\&y + x\&y\&z$
$x|(y + z) \Leftrightarrow x|y \,.\, x|z$ $\quad (y + z)|x \Leftrightarrow y|x + z|x$
$x|(y \,.\, z) \Leftrightarrow x|y + x|z$ $\quad (y \,.\, z)|x \Leftrightarrow y|x \,.\, z|x$

Because each rule above reduces the height of the temporal expression to which it is applied, any sequence of such rule applications must eventually terminate and hence lead to an expression in base temporal form.

Within cut sets, reduction is based on redundancy according to the Idempotent and Absorption laws, i.e. $x \,.\, x \Leftrightarrow x$ and $x + x \,.\, y \Leftrightarrow x$, but cut sequences also introduce the possibility of contradiction – from the Mutual Exclusion or Simultaneity laws given earlier, for example. Together with the laws $x \,.\, false \Leftrightarrow false$ and $x + false \Leftrightarrow x$, it is possible to entirely remove cut sequences containing contradictions. For example, the expression $x < (x + y)$ can be rewritten using one of the laws shown above and the usual Distributive law to obtain two cut sequences in BTF: $x|x \,.\, x|y \,.\, x + x|x \,.\, x|y \,.\, y$. Because both contain $x|x$, which is a contradiction according to the Simultaneity Law, both cut sequences are impossible (since $x.false \Leftrightarrow false$) and therefore the result is equivalent to $false + false \Leftrightarrow false$.

4 Example

As an example of how Pandora can improve the analysis of a dynamic system, consider the design pattern of Fig. 2. The pattern describes a generic primary-backup recovery system in which the backup is of simpler design and provides only a subset of the functionality of the primary. The diverse design of primary and backup components serves as a guard against common cause failures. The advantage of using such a generic example for illustration of Pandora is that the results of analysis can be easily generalised on a wider class of systems that follow this particular pattern. It can be easily imagined, for example, how specific components (e.g. a control computer in a safety critical process) and the failure modes of these components can substitute the generic references made in this section.

"Out" is the system output, and is initially the output of Algorithmic unit 1 ("A1"). A1 performs a function on the measurements provided by sensors "S1" and "S2", which monitor a common input. "M" is a monitor that detects an omission of output from A1 and activates the redundant Algorithmic unit "A2" to replace A1. In that case, the system output is provided by A2 instead. A1 is capable of detecting an omission from one of the two sensors, and can continue operation with just one sensor. Omission from both sensors will lead to an omission from A1. If the values provided by S1 and S2 differ, A1 will use an average value, so any value errors propagate through to the output. A2 is of a simpler design, and directly propagates omission and value failures received from sensor S2. A2 does not use any input from S1.

There are two possible output failures for this system: Omission (O-Out), caused by an omission of output from both A2 (O-A2) and A1 (O-A1); and Value (V-Out),

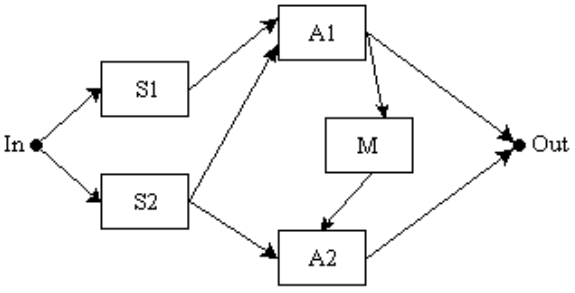


Fig. 2. Example System

caused by undetectable value failures in the outputs of either A1 or A2 (V-A1 and V-A2 respectively), depending on which is currently active. These failures are given using a simplified form of the HiP-HOPS notation. There are two types of failure in HiP-HOPS – *deviations*, which represent an error in the input or output of a component, and *internal failures*, which mean the component itself has suffered a failure. Together, these make up the local failure behaviour annotations for each component by indicating how output deviations are caused by a mixture of input deviations (due to faults elsewhere in the system, i.e. not local to the component) and internal failures (faults in the current component). The failure behaviour of the example system is:

Out:	O-Out = O-A2	V-Out = V-A2 + V-A1
A2:	O-A2 = O-M + StartA2-M.(A2 +O-S2)	V-A2 = StartA2-M.V-S2
M:	O-M = M<O-A1 + M&O-A1	StartA2-M = O-A1
A1:	O-A1= A1 + O-S1.O-S2	V-A1 = (V-S1+V-S2) |A1
S1:	O-S1 = S1f	V-S1 = S1b + V-I
S2:	O-S2 = S2f	V-S2 = S2b + V-I

On the left of each expression is the output deviation. The first letter indicates the failure class (i.e. O = Omission, V = Value) and the second part is the name of the component in question. On the right are the contributing input deviations (in the same format as the output deviations) and internal failures. Internal failures have no failure class, and are represented here by just the name of the component, e.g. M represents a failure of the monitor, and A1 represents a failure of A1. StartA2-M is not a failure but an event representing the conditions for the monitor to start A2. And since O-A2 includes O-A1, we do not repeat O-A1 in O-Out, since if O-A2 is true O-A1 must also be true.

The sensors have two internal failure modes each: S1b and S2b both indicate the values provided by the sensor in question are biased, leading to a value error; S1f and S2f mean the sensor has failed, leading to an omission. The three temporal operators are each used in the expressions to represent a more accurate model of the failure behaviour; for example, an omission from the monitor M occurs only if the monitor

fails before or at the same time as A1, and the POR is used to indicate that a value error from A1 is caused if the sensors provide incorrect output before A1 fails.

The first step in the analysis is to start with a system output deviation and substitute all input deviations for their equivalent output deviations, e.g. O-A2 is replaced by O-M + StartA2-M.(A2 +O-S2). This process continues until only internal failures remain in the expression. The resulting expression then represents the fault tree for that system output, and indicates the root causes for that failure. The two fault tree expressions for this example are as follows:

```
O-Out = M<(A1 + S1f.S2f) + M&(A1 + S1f.S2f)
      + (A1 + S1f.S2f).(A2 + S2f)
V-Out = (A1 + S1f.S2f).(V-I + S2b)
      + (V-I + S1b + V-I + S2b) | A1
```

The next step is to begin to apply the laws of reduction to these expressions in order to arrive at the base temporal form of the expressions. For example, `M < (A1 + S1f.S2f)` may be rewritten to place the OR uppermost in the expression. Three laws are needed to achieve this: $x<(y + z) \Leftrightarrow x|y \,.\, x|z \,.\, (y + z)$, which develops < over +, $x|(y \,.\, z) \Leftrightarrow x|y + x|z$, which develops POR over AND, and one of the traditional Boolean Distributive laws, $x \,.\, (y + z) \Leftrightarrow x \,.\, y + x \,.\, z$. Applying them gives the expression in base temporal form:

```
M < (A1 + S1f.S2f)
⇔  M|A1 . M|(S1f.S2f) . (A1 + S1f.S2f)
⇔  M|A1 . M|(S1f.S2f) . A1 +
   M|A1 . M|(S1f.S2f) . S1f.S2f
⇔  M|A1 . (M|S1f + M|S2f) . A1 +
   M|A1 . (M|S1f + M|S2f) . S1f.S2f
⇔  [M|A1] . [M|S1f] . A1 + [M|A1] . [M|S2f] . A1 +
   [M|A1] . [M|S1f] . S1f . S2f +
   [M|A1] . [M|S2f] . S1f . S2f
```

The last expression is in base temporal form (BTF) and contains the doublets. Once the expression is in BTF, it is possible to perform some reduction. In this case, the temporal law $x|y \,.\, y \Leftrightarrow x<y$ can be applied to reduce `[M|A1].A1` to `[M<A1]`, and similarly for S1f and S2f (i.e. `[M|S1f].S1f` ⇔ `[M<S1f]`).

The final step is to perform reduction between each cut sequence, eliminating redundancies. For example, once V-Out has been transformed into BTF, it is possible to apply temporal laws such as $x|y + x \,.\, y \Leftrightarrow x$ and the traditional absorption law $x + x \,.\, y \Leftrightarrow x$ to reduce the number of cut sequences:

```
V-Out = (A1+S1f.S2f).(V-I+S2b) + (V-I+S1b+V-I+S2b)|A1
⇔  A1.V-I + A1.S2b + S1f.S2f.V-I + S1f.S2f.S2b +
   [V-I|A1] + [S1b|A1] + [S2b|A1]
⇔  V-I + S2b + S1f.S2f.V-I + S1f.S2f.S2b + [S1b|A1]
⇔  V-I + S2b + [S1b|A1]
```

In this case, for example, `A1.V-I + [V-I|A1]` reduces to `V-I` which in turn means `S1f.S2f.V-I` is redundant. As a result, we can reduce the original seven cut sequences to just three MCSQs. The results for both O-Out and V-Out are:

O-Out	V-Out
M<A1	V-I
M&A1	S1b | A1
A1.A2	S2b
A1.S2f	
S1f.S2f	

These eight minimal cut sequences together represent all the minimal causes of the two system failures. An omission of output can be caused if: the monitor fails before or at the same time as A1, in which case it never activates the standby A2; if both A1 and A2 fail; if both sensors fail; or if S2 fails and A1 fails, in which case A2 has no input. A value failure of the output, by contrast, is caused either by a value failure of the input, a biased result from S2, or a biased result from S1 before A1 fails. If S1 provides a biased result *after* A1 fails, then it is irrelevant because the output is now being provided by A2, which takes its input only from S2. This means that a failure of A2 independent of S2b, once A2 is activated, leads to an omission from the system, rather than a value failure; this behaviour is inexpressible in classical FTA without using NOT gates.

The subtleties of these results make them more accurate than the results obtainable through purely traditional FTA. Using only Boolean operators means that M.A1 would be a cut set (which ignores the fact that if M fails after A1, the system can continue to operate) and that S1b alone would cause a value error, which is not the case if A1 has failed and A2 is providing output instead. It is not possible to represent the fact that S1b *only* causes a value failure when the system is using A1 using only AND and OR. With the temporal operators, the results are more informative and only slightly more complex.

5 Conclusion

Pandora is a new temporal extension to fault trees intended to enable them to represent the effects of sequences of events and to allow for analysis of systems in which such sequences play a role. Pandora is defined in such a way to be similar to both the structure and semantics of existing fault trees, and it introduces only three new gates that allow analysts to represent many possible combinations of events. By using a formalisation based on event orderings, which are sequences of sets of simultaneous events, it is possible to prove, within Predicate Logic, a set of laws that can be used to qualitatively analyse Pandora fault trees and obtain minimal cut sequences, analogous to minimal cut sets in ordinary fault trees. These show the minimum combinations or sequences of events necessary to cause the top event. In the future, the aim is to extend Pandora to allow for the quantitative analysis of minimal cut sequences too.

Furthermore, Pandora is compatible with HiP-HOPS, which is a hierarchical and compositional methodology for the automatic synthesis of fault trees. HiP-HOPS uses

hierarchical descriptions of component-level failure behaviour and the topology of the system to build system-wide fault trees. By extending these descriptions to include temporal expressions, it is possible for Pandora to be integrated into HiP-HOPS, enabling the compositional synthesis and analysis of temporal fault trees as well as normal fault trees. This allows HiP-HOPS to be used to more precisely analyse complex dynamic systems in which the order of faults and events is critical to the failure behaviour. The result, we hope, is a powerful tool which can produce more accurate and more informative safety analyses for a wider range of different systems.

References

1. Papadopoulos, Y.I., McDermid, J.A., Sasse, R., Heiner, G.: Analysis and Synthesis of the Behaviour of Complex Systems in Conditions of Failure. Reliability Engineering & System Safety 71(3), 229–247 (2001)
2. Grunske, L., Kaiser, B.: Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations. In: Fifth International Conference on Quality Software (QSIC 2005), 9th edn., pp. 117–123. IEEE Computer Society, Los Alamitos (2005) ISBN 0-7695-2472-9
3. Walker, M.D., Papadopoulos, Y.I.: Pandora: The Time of Priority-AND gates. In: INCOM 2006, France, pp. 237–242 (2006)
4. Fussel, J.B., Aber, E.F., Rahl, R.G.: On quantitative analysis of PAND failure logic. IEEE Trans. on Reliability R-25/5, 324–326 (1976)
5. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. US Nuclear Regulatory Commission, Washington D.C., USA (1981)
6. Long, W., Sato, Y., Horigome, M.: Quantification of sequential failure logic for fault tree analysis. Reliability Engineering & System Safety 67, 269–274 (2000)
7. Vesely, W.E., Stamatelatos, M., Dugan, J.B., Fragola, J., Minarick, J., Railsback, J.: Fault Tree Handbook with Aerospace Applications. NASA Office of Safety and Mission Assurance, USA (2002)
8. Tang, Z., Dugan, J.B.: Minimal cut set/sequence generation for dynamic fault trees. In: Annual Reliability and Maintainability Symposium Proceedings (2004)
9. Palshikar, G.K.: Temporal Fault Trees. Information and Software Technology 44, 137–150 (2002)
10. Wijayarathna, P.G., Kawata, Y., Santosa, A., Isogai, K.: Representing relative temporal knowledge with the tand connective. Eighth Ireland Conference on Artificial Intelligence' (AI-97) 2, 80–87 (1997)
11. Andrews, J.D.: To not or not to not. In: Proceedings of the 18th International System Safety Conference, Fort Worth, September 2000, pp. 267–275 (2000)
12. Gorski, J., Wardzinski, A.: Deriving Real-Time Requirements for Software from Safety Analysis. In: Proc. 8th Euromicro Workshop on Real-Time Systems, pp. 9–14. IEEE Computer Society Press, Los Alamitos (1996)
13. Hansen, K.M., Ravn, A.P.: From Safety Analysis to Software Requirements. IEEE Trans. on Software Engineering 24(7), 573–584 (1998)

Representing Parameterised Fault Trees Using Bayesian Networks

William Marsh[1] and George Bearfield[1,2]

[1] Department of Computer Science, Queen Mary, University of London, Mile End Road, London, E1 4NS
[2] Safety Policy Department, Rail Safety and Standards Board, London, UK
william@dcs.qmul.ac.uk, george.bearfield@rssb.co.uk

Abstract. Fault trees are used to model how failures lead to hazards and so to estimate the frequencies of the identified hazards of a system. Large systems, such as a rail network, do not give rise to endless different hazards. Rather, similar hazards arise repeatedly but with different frequency depending on factors such as location. Several authors have identified the need to build models to estimate both system-wide average hazard frequencies and hazard frequencies in specific situations. Fault trees can be used for this but they grow as additional factors are considered. In this paper, we describe a compact model using Bayesian networks. The fault tree notation is retained; with base events parameterised by variables in the Bayesian net to represent a mixture of related fault trees compactly. We use a simple example to describe the model structure and report on ongoing work on a model of train derailment.

Keywords: risk analysis, fault tree, Bayesian network.

1 Introduction

Quantitative risk modelling is used for safety management across a wide range of industries such as nuclear power generation, aviation, chemical processing, oil and gas, and transportation. A common approach is to use a fault tree to represent the combinations of failures leading to a hazard, together with an event tree to model the possible accidents. These models are well suited to the modelling of risk in particular locations, for example at a particular nuclear power station where the factors that influence failure rates can be investigated and specific failure probabilities determined.

The models are less well suited to industries where accidents can happen over a large geographic area. In such industries, such as railway transportation, although the accidents that can occur in different locations are similar, there will be local differences in factors that influence failure rates. Risk models in such industries can either be revised for each location or averaged over all locations. Both forms of model are used: a location specific model is used to analyse an operational or engineering change in a specific location whilst a risk model averaging over all locations is useful for monitoring system-wide risk.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 120–133, 2007.

Several groups have identified the need to build risk models for multiple locations without averaging. Since understanding the variation in risk arising from local differences is important for making decisions about how to reduce risks, it is desirable to make the factors giving rise to the variation explicit in the risk model. If the factors are used to parameterise a generic model, then the risk model for a specific location can be obtained by giving the factor value appropriate for the location and a profile of risk can be generated by location or by another causative factor.

The objective of this paper is to describe how a Bayesian network can be used to represent a risk model that is generalised in this way, equivalent to a set of fault trees, with explicit parameterisation by factors. We do this using a highly simplified case study, introduced using fault trees in Section 4 and then represented in Section 5 using Bayesian network. Before this, we cover some background: Section 2 describes fault trees, Bayesian networks and the relationship between them and Section 3 describes existing system-wide risk models. Further work and conclusions are in Section 6.

2 Fault-Trees and Bayesian Networks

In this section we review fault trees and Bayesian networks briefly, together with existing work on the relationship between them.

2.1 Bayesian Networks

A Bayesian network [1] shows the dependencies between a set of probabilistic variables. The network is made up of nodes and arcs. The nodes correspond to random variables: each variable has a finite set of possible states but we may be uncertain which state a variable is in. The arcs in the network represent probabilistic influence: the state of one node influences the probabilities of the states of another node. Each node has a table of conditional probabilities, providing the probabilities of each state of the variable for each combination of the states of parent variables. A Bayesian network can therefore be used to model relationships of cause and effect, which may be uncertain.

The network represents the joint probability distribution of all the variables. If the value of one variable is known, this evidence can be entered and 'propagated' around the network, updating the marginal probability distributions of the other variables. Although Bayesian probability theory has a long history, executing realistic models was only first made possible in the late 1980s using new algorithms [2], making it possible to apply Bayesian networks to the problems of systems engineering.

Bayesian networks have been applied to safety and risk assessment. In [3] Bayesian networks are used to model changes in an organisation's safety culture, and provide a quantified assessment of the impact of such changes on risk. In [4, 5] the use of Bayesian networks to model the technical and managerial causes of aviation accidents at airports is described.

2.2 Fault Trees

Fault trees were developed in the early 1960's by Watson and Mearns of Bell Laboratories for analysis of the Minuteman launch control system [6]. The technique

became well known after it was used to review nuclear power plant design in the WASH-1400 study [7].

A fault tree is an analytical technique for describing the combinations of failure events that cause a system failure. The system failure being analysed is called the 'top-event', and its decomposition into 'base events' is specified using logical AND and OR gates. Other types of gate can also be used, for example exclusive OR gates, or NOR gates. The probability of the top-event is calculated from the set of minimal cut sets, each set containing the base events whose simultaneous occurrence causes the top-event. This calculation is automated using computer programs such as Fault Tree + [8].

2.3 Fault Trees and Bayesian Networks

Bobbio et al [9] outline how any fault tree can be directly translated into an equivalent Bayesian network. The diagram of Fig. 1 shows networks equivalent to fault tree AND and OR gates, with the different logical relationship of each represented in the conditional probability distributions for the network nodes. Given the probabilities of occurrence of base events, the Bayesian network calculates the top event probability.

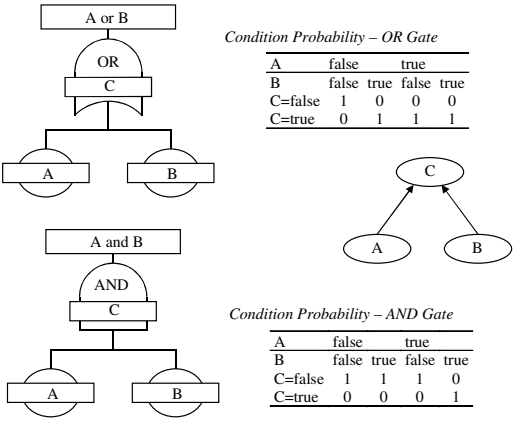


Fig. 1. Fault Tree Gates as Bayesian networks

We show below how related fault trees can be merged into a single Bayesian network. We also show how the resulting failure model can be parameterised by the factors that determine the base event probabilities. The fault trees representing a particular location can be retrieved from this failure model by entering evidence in Bayesian network to match the conditions in that location.

3 System-Wide Risk Models

We are aware of several initiatives, particularly in the railway industry, to build system wide models, modelling the risk associated with a whole system rather than just a specific location within the system.

3.1 The UK Railway Safety Risk Model

The Safety Risk Model (SRM) is a model of risk on the UK Rail Network [10]. It consists of a series of fault and event tree models for 110 hazardous events that together estimate the overall level of risk on the railway. The base event probabilities are derived from historical data of incidents and accidents. This leads to the SRM calculating averages, taken over the whole network, allowing it to be used to monitor network-wide risks. However, because the causes of varying base event probabilities are not included, the model does not profile the risk in different locations.

3.2 Parameterised Risk Models

Others have identified the need to develop models that can be used to estimate the whole system risk as well as to analyse risks in different locations. This requires the factors that determine failure probabilities to be included and their dependence on location to be modelled.

Research was commissioned by the RSSB in the UK [11, 12] to investigate a model of this type to optimise track inspection and monitoring processes in order to reduce the risk of train derailment due to track faults. The study identified nine separate track faults to be modelled with separate sets of fault trees. A prototype model of the risk due to one of these types of fault, gauge spreading, was built using fault trees. The approach used fault trees in two ways: the first use was a traditional fault tree of failures leading to a hazard; while the second use combined relevant 'environmental factors' (such as the degree of curvature, rail condition and rail type) and used these to determine the base event probabilities in the first tree.

The environment factors were parameters in the model, so that the failure probabilities could be varied without a separate fault tree for each case. However, despite this, a large number of fault trees were needed, as it was not possible to treat all factors as parameters. Separate fault trees were needed for high, medium and low severity gauge spreading faults as different control measures applied to each, and also for high and low speed conditions. This resulted in six similar, but entirely separate fault tree structures being built for this one particular type of track fault. Despite the efforts to simplify the modelling approach using parameterisation, the complete track fault derailment model would have required approximately 55 separate fault tree structures to be built and the development was not completed.

A parameterised risk model has been produced for Irish rail [13]. The model consists of fault trees and event trees modelling all of the accident sequences on the Irish railway network. Over 200 parameters were identified and sets of values for these parameters were gathered for 227 separate locations. These parameters include ratings for the design and condition of each asset, human factor performance and location factors such as train loading, frequency and topography. The model assumes that the states of parameters are fixed in any particular location and therefore locations are selected to have uniform characteristics. Cut sets generated from the fault trees are evaluated for different parameter values using Excel. The risk model can calculate risk breakdowns, such as risk by location, line and by type of asset.

In a recent review [14] of the state of the art in causal modelling in aviation, Ale et al. argue that more detailed causal models are needed of the particular circumstances

in which 'organisational accidents' occur. They outline a research project to develop causal models in the aviation industry for this purpose, bringing together existing work undertaken using fault trees, event trees and Bayesian networks. However they do not describe in detail how they propose to build such models.

4 A Simplified Case Study, Using Fault Trees

In this section we describe a simplified but illustrative failure model, using fault trees. The application is described first and then two fault trees, covering different situations (or locations) are given. Each fault tree has several different quantifications, depending on the local conditions, so that the full failure model would require multiple copies of the two fault trees. In Section 5, we show how the same failure model is represented using a single Bayesian network.

4.1 'SPAD' Events

Train separation on the UK railway network is maintained through the use of line-side signalling. A track circuit for each 'block section' of the line detects the presence of a train. A block section consists of the area of track preceding a signal, and also a small 'overlap' beyond to prevent an accident occurring if the braking is misjudged or if the train's adhesion to the track is poor.

The signal behind an occupied block section is set to red to indicate that no other train should enter the block section. Safety therefore relies on the driver reacting to the signalling indications. A protection system, called the Train Protection and Warning System (TPWS), is used at all high-risk signals automatically to apply the brakes of a train that fails to slow at a red signal. If a train passes a red signal for any reason, the incident is known as a Signal Passed at Danger (SPAD).

This case study is a highly simplified model of the causes of SPADs events. The case study is intended to illustrate the use of a Bayesian network for modelling failure: the logic of the causes of SPADs and the failure probabilities are for illustration only.

4.2 The Logic of SPADs, Using Fault Trees

Two locations are distinguished: (i) a junction with TPWS fitted, and (ii) a plain line, where TPWS is not fitted. For the purposes of the case study, we will assume that these locations cover all variations in the failure logic. Identifying a sufficient set of such variations is a necessary step in creating a failure model: in practice more cases would be needed.

Fault Tree 1: Junction with TPWS. At this type of junction, TPWS is fitted. A SPAD occurs if the driver fails to react to a signal and if TPWS fails to operate. The risk of poor train adhesion leading to a SPAD (beyond the overlap) is negligible since the overlap is greater than 200m, even though the speed is high. The fault tree is shown in Fig. 2.

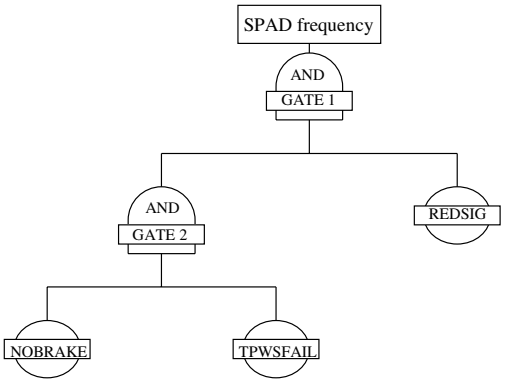


Fig. 2. FT for the Junction Signal with TPWS

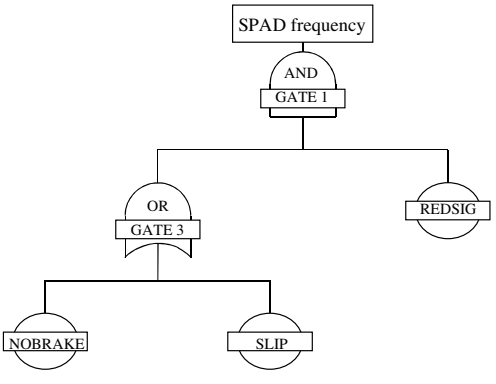


Fig. 3. FT for the Plain Line

Fault Tree 2: Plain Line. At this type of signal, TPWS is not fitted and the overlap is less than 200m. A SPAD occurs if the driver fails to react to a signal (NOBRAKE event). A SPAD, beyond the end of the overlap, may also occur if the train adhesion is poor (SLIP event), particularly if the speed is high, since in these conditions it is possible for the driver to misjudge the application of the brakes. The fault tree is shown in Fig. 3.

4.3 Fault Tree Events and Probabilities

The events in the fault trees are shown in Table 1. Except for the REDSIG event, these events are quantified as probabilities. The REDSIG event represents the number of trains per year approaching a red signal and is used to scale the failure model, so that the top-event gives an estimate of the number of SPAD events. Some fault tree software (such as FaultTree+) allows this usage although it departs from the standard definition of an event.

The probabilities of the events may depend on the conditions that can vary in the two locations analysed by fault trees. For the first fault tree, covering the junction fitted with TPWS, the signal sighting varies changing the NOBRAKE probability. In the second fault tree the signal sighting, line speed and adhesion can vary. The factors, or conditions, that cause variation and the event affected are shown in Table 2 and the probabilities are in Table 3.

Table 1. Events in the SPAD Fault Tree

Event	Description
REDSIG	Trains approaching red signal aspect (number/year)
SLIP	Train adhesion failure
NOBRAKE	Driver fails to brake at a red signal aspect
TPWSFAIL	TPWS fails on demand, i.e. conditional on train passing the signal.

Table 2. Conditions, Representing Attribute of the Rail Infrastructure

Condition	Description	Events Influenced
Signal sighting	The signal may be easier or harder to read from the driver cab. Values: poor, good	NOBRAKE
TPWS fitted	TPWS is not fitted to all signals. Values: True, False	NOBRAKE, TPWSFAILS
Overlap length	The overlap following a junction maybe shorter. Values: <200m, >200m	SLIP
Train speed	The speed of the train towards the signal varies. Values: High (>100 mph), Medium (100-60 mph), Low (<60 mph)	SLIP
Adhesion	The train may be liable to slip. Values: Good, Poor	SLIP

Table 3. Event Probabilities

Event	Probability	Condition 1	Condition 2
TPWSFAIL	0.01		
NOBRAKE	1E-08	Good signal sighting	
	1E-06	Poor signal sighting	
NOBRAKE	1E-08	Good signal sighting	
	1E-06	Poor signal sighting	
SLIP	1E-06	High speed	Good adhesion
	5E-06		Poor adhesion
	1E-07	Medium speed	Good adhesion
	5E-07		Poor adhesion
	5E-07	Low speed	Good adhesion
	5E-07		Poor adhesion

Table 4. Quantification of the REDSIG event

Fault Tree	Condition 1	Condition 2	Condition 3	REDSIG (number/year)
1	Good signal sighting			5,000,000
1	Poor signal sighting			100,000
2	Good signal sighting	High speed	Good adhesion	5,000,000
2			Poor adhesion	50,000
2		Medium speed	Good adhesion	10,000,000
2			Poor adhesion	100,000
2		Low speed	Good adhesion	10,000,000
2			Poor adhesion	500,000
2	Poor signal sighting	High speed	Good adhesion	50,000
2			Poor adhesion	10,000
2		Medium speed	Good adhesion	100,000
2			Poor adhesion	50,000
2		Low speed	Good adhesion	500,000
2			Poor adhesion	100,000
			Total	31,560,000

The variation in base event probabilities could be analysed by duplicating the fault trees and combining the different copies with an OR-gate. Given the number of values of the conditions distinguished, we would need 2 copies of the tree in Fig. 2 (given by 2 values of signal sighting) and 12 copies of the fault tree in Fig. 3 (given by 2 values of signal sighting, 3 values of train speed and 2 values of adhesion), making 14 'situations' overall. The REDSIG event also needs to be quantified by giving the number of approach events in each of the 14 situations. Table 4 shows example data.

5 The SPAD Failure Model as a Bayesian Network

In this section we show that the fault trees of Section 4 can be represented using a Bayesian Network. The main advantages of this representation are:

- The 14 variants of the fault tree are combined into a single Bayesian Network
- The conditions that cause the probabilities in the fault tree to vary are made explicit as parameters in the model.

5.1 Variables in the Bayesian Network

The network is shown in Fig. 4. The variables (or nodes) in the network are of four types:

1. A base event, with states 'true' and 'false'.
2. A gate is a variable with states 'true' and 'false'.
3. A condition, with states corresponding to the possible values of the condition.
4. The situation variable, with a state for each of the 14 situations described in Section 4.

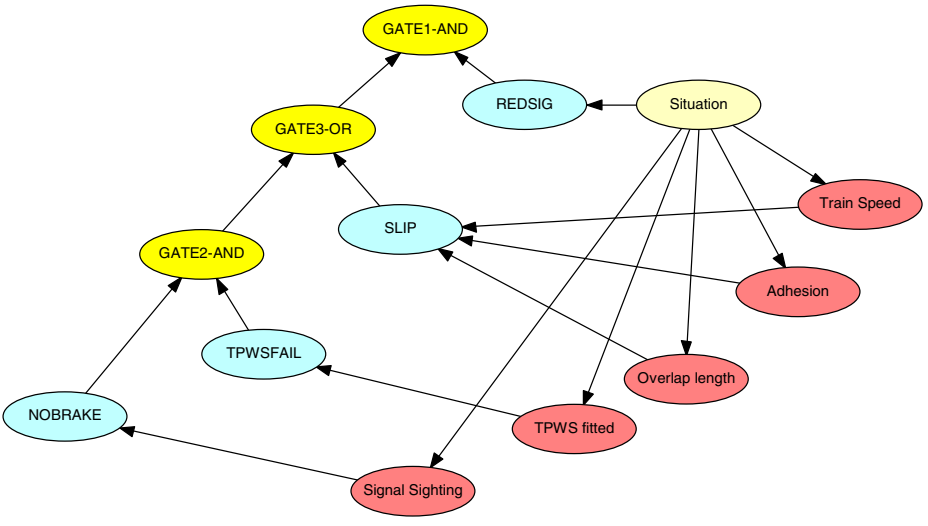


Fig. 4. SPAD Model as a Bayesian Network

5.2 Conditional Probability Tables

The probability tables for the gate variables are deterministic, corresponding to 'logical and' or 'logical or', as shown in Fig. 1. The probability table for the base events NOBRAKE, TPWSFAIL and SLIP are determined from Table 3, with the additional constraints:

1. the probability of TPWSFAIL is unity when TPWS is not fitted, and
2. the probability of SLIP is zero when the overlap length exceeds 200m.

The number of red signal approaches is represented in the Bayesian network using the 'Situation' variable and the 'REDSIG' variable, which corresponds to the 'REDSIG' event but is quantified with a probability. The 'REDSIG' variable depends on 'Situation' and the probability is the proportion of train approaching a red signal belonging to each situation, determined from the data in Table 4. For example, there are 500,000 (out of a total of 31,560,000) train approaches in the situation given by good signal sighting, TPWS not fitted, overlap less than 200m and poor adhesion, corresponding to a fraction of 1.584% of the total train approaches.

The 'Situation' determines the setting of each condition. In this case study, we have a deterministic relationship between the situations and the condition value, but a distribution over the possible values of the condition could be expressed as well. The 'Situation' node has a uniform (prior) distribution.

5.3 Viewing the Bayesian Network as a Combined Fault Tree

The Bayesian network, excluding the nodes representing conditions, can be viewed as the single fault tree shown in Fig. 5. The fault tree gives a clearer view of the logic of the failure – how the base events lead to the top event – than the Bayesian network,

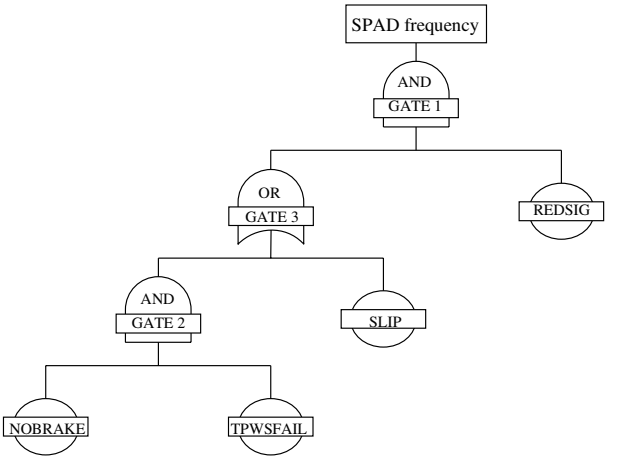


Fig. 5. BN Viewed as a Combined Fault Tree

since the fault tree notation distinguishes AND from OR. Moreover, the fault tree notation is familiar to safety engineers. It is therefore preferable to view the logical part of model as a fault tree.

As well as viewing the model as the combined fault tree of Fig. 5, it is also possible to project the original fault trees as special cases. Fig. 5 corresponds to Fig. 2 when the probability of the SLIP event is zero, which is the case when the 'overlap length' condition is set to '> 200m'. Similarly, the combined fault tree corresponds to Fig. 3 when the probability of TPWSFAIL event is unity – representing the condition of TPWS not being fitted.

5.4 Calculations Using the Bayesian Network

The Bayesian network can be used to calculate the SPAD probability in a given situation. This is done by entering evidence at the situation variable – setting it to a particular situation – and observing the marginal probability at the node 'GATE1-AND'. The result for each situation is shown in Fig. 6. The probability shown is the probability that a signal approach occurs in the situation and leads to a SPAD, thus most SPADs are estimated to occur in situations 3 and 7. This calculation allows for the varying fraction of signal approaches in each situation, so an estimate of the number of SPADs expected in each situation is obtained by multiplying the probability by the uniform scale factor of 31,650,000 signal approaches per year.

The Bayesian Network can also be used to calculate the aggregate probability by setting the 'REDSIG' node to 'true' and leaving other nodes unset. The overall SPAD probability, weighted over all the situations in 4.136e-07. Setting the 'REDSIG' node gives marginal probabilities on the 'Situation' node equal to the fraction of signal approaches in each situation. This overall probability is the sum of the probabilities in each situation, which is the correct result provided that the situations are mutually exclusive. The result given by constructing a fault tree with its top event given by the

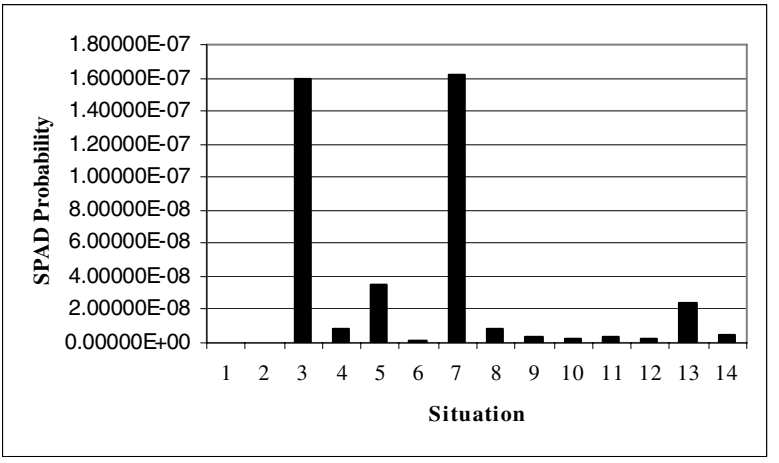


Fig. 6. SPAD Probability in Each Situation. The situation numbers follow the rows of Table 4.

disjunction of a fault tree for each situation – two copies Fig. 2 and twelve of Fig. 3 – does not assume mutual exclusion. Instead, each minimum cut set derives from one of the situation fault trees and the top event probability allows for the overlap of the cut sets; the sum of the situation probabilities is the first approximation to this calculation. The assumption of mutual exclusion is appropriate to the way we have scaled the failure model, since each approach by a train to a red signal belongs to exactly one situation.

6 Further Work and Summary

In this section we summarise the contribution of the paper and outline some areas of further work towards the goals described in Section 1.

6.1 Modelling the Infrastructure: How Factors Relate

In the failure model we have described above, the situation determines the value of the factors. This relationship between situation and factors forms a model of the infrastructure, possibility extending to the environment. The importance of the situation is to be able to calibrate (or scale) the model, with the number of events at risk – in the case study, a train approaching a red signal – counted for each situation. A simple approach is for the situation to be a geographical area, but other partitions such as routes combined with train type could be used provided that the number of events can be estimated.

In principle, the data on the infrastructure and its use needed for an infrastructure model could be taken from an asset database together with maintenance and operational records. However, there are many practical issues to be resolved including how to define situations, which factors are important and how many values of each factor should be distinguished.

In the case study, the value of each factor is uniquely determined in each situation. This restriction is unnecessary: a more general model would be for the situation to determine a distribution of the factor values. Another common simplification we are aware of in the work surveyed in Section 3 is for infrastructure factors to vary independently, given the location. In the context of our case study this implies that, for example, a signal with poor adhesion is no more or less likely to be hard to see than the average for all signals in the same geographical area. We would like to investigate how this approximation impacts the overall risk: if dependencies do exist it seems possible that this could impact the overall risk significantly.

Whether or not situations are geographical, it is likely that analyses of the top-event probabilities for different groups of situations would be useful. This could be done using a program interface to the Bayesian network, which automatically enters the appropriate evidence in the Bayesian network. We plan to develop software to assist with the construction and use of failure models of the form we have described, so that the Bayesian network is largely hidden from the user.

6.2 Modelling Train Derailment

We are currently working on a Bayesian network, of the type described above, to model derailment risk. The model covers derailments caused by track and train faults, over speeding and obstructions. The fault tree part of the model can be parameterised by the following factors:

- Rolling stock type: high-speed train, electric multiple unit, freight.
- Rolling stock fault type: break failure, axle failure, wheel failure, and suspension failure.
- Rolling stock fault severity.
- Rolling stock inspection interval.
- Effectiveness of rolling stock maintenance.
- Track type: plain line, switch and crossing.
- Track fault type: gauge spread, track twist, broken rail, buckled rail, broken fishplate, subsidence failure.
- Track fault severity and curvature.
- Location of track: in tunnel, on tunnel approach, outside rural, outside urban, in station.
- Switch and crossing fault type and fault severity.
- Effectiveness of infrastructure maintenance.
- Type of obstruction.
- Traffic density and train speed.

The Bayesian Network consists of just 64 nodes and can calculate the top event probabilities with all possible combinations of the parameter values (over 10^8 in total). The number of fault trees replaced would be the number of these combinations occurring in practice. No node in the Bayesian network has more than four parents, with the result that the model has 905 probabilities that need to be determined.

In related work [15, 16] we showed how sets of similar event trees could be merged into a single BN model and generalised in a similar way to that shown here.

We intend to link the event tree model and the fault tree model to form a single BN in which factors can affect both parts of the model, ensuring that correlations between variations in the fault tree and event tree parts of the model are effectively captured.

6.3 Building Parameterised Risk Models

As we have shown, it is possible to view the logical part of the model as a fault tree so that the Bayesian net can be hidden from the risk analyst. However, there are still some unresolved issues about how to build these models in practice. A simple approach would be to build the combined fault tree (corresponding to Fig. 5) and then to consider the failure probability for all relevant factor values. In our simple case study the fault tree of Fig. 5 can be interpreted as representing the case of TPWS fitted with overlap length '< 200m'. Since this makes sense, it is reasonable to expect a risk analyst to use the standard top-down method to build the tree. However, the combined fault tree will not always have a coherent interpretation – it could mix cases that make no sense in combination – making this approach unworkable. Moreover, it would be preferable to handle the 0 and 1 failure probabilities implicitly by building fault trees for a representative number of cases. We plan to develop prototype software to investigate these issues more fully.

6.4 Summary

We have shown how a Bayesian network can be used to represent a risk model that is generalised for multiple locations, or other subdivisions of a system, rather than averaged over all locations. The generalisation is achieved by including factors – properties of the system and its environment – that cause the failure logic and the failure probabilities to vary between different locations.

The parameterised model is equivalent to a set of fault trees but is represented by a Bayesian network. The part of the model expressing the logic of failure can be viewed as fault tree. The Bayesian network representation is compact, so that more factors to be added to the model without an exponential increase in the number of model elements. The representation will make it practical to build system-wide risk models, in the manner proposed by Ale et al. [14], for the causal analysis of the profile of risk at different locations in a system made up of repeated installations or sub-systems, such a railway network.

References

1. Jensen, F.V.: An Introduction to Bayesian Networks. UCL Press, London (1996)
2. Lauritzen, S.L., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems (with discussion). Journal of the Royal Statistical Society, Series B 50, 157–224 (1988)
3. Neil, M., Malcolm, B., Shaw, R.: Modelling an Air Traffic Control Environment Using Bayesian Belief Networks. In: Proc. 21st Systems Safety Conference, Ottawa (2003) ISBN 0-9721385-2-8
4. Roelen, A.L.C., Wever, R., Cooke, R.M., Lopuhaä, H.P., Hale, A.R., Goossens, L.H.J.: Aviation causal model using Bayesian Belief Nets to quantify management influence. In: van Gelder, B. (ed.) Safety and Reliability, Swets & Zeitlinger, Lisse, pp. 1321–1327 (2003) ISBN 90 5809 551 7

5. Roelen, A.L.C., Wever, R., Hale, A.R., Goossens, L.H.J., Cooke, R.M., Lopuhaä, H.P., Simons, M., Valk, P.J.L.: Causal modeling using Bayesian belief nets for integrated safety at airports. Risk Decision and Policy 9(3), 207–222 (2004)
6. Watson, H.A.: Launch Control Safety Study, Section VII, vol. 1. Bell Labs, Murray Hill, New Jersey (1961)
7. US Nuclear Regulatory Commission: Reactor Safety Study. WASH-1400, NUREG 75/014 (1975)
8. Isograph Software: Fault tree+ for Windows, v11.0, http://www.isograph.com/faulttree.htm
9. Bobbio, A., Portinale, L., Minichino, M., Ciancamerla, E.: Improving the Analysis of Dependable Systems by Mapping Fault Trees into Bayesian Networks. Reliability Engineering and System Safety 71, 249–260 (2001)
10. Dennis, C., Somalya, K., Small, A., Singh, P.: The Railway Safety Risk Model. The Journal of the Safety and Reliability Society 22(3), 39–48 (2002)
11. Campbell, B., Kennedy, A.: Derailment Risk Model (Track Faults), Phase 1. Report of project: T207: Development of a prototype model for managing derailment risk due to track faults. Prepared by Risk Solutions for Rail Safety and Standards Board (2003), Available from http://www.rssb.co.uk/
12. Safety and Standards Directorate: Safety Risk Model (SRM), Report No. SP-RSK-3.1.3.8 Rail Safety and Standards Board, UK (1999)
13. Sotera Ltd.: Developing a location specific risk model for Irish Rail (2006) http://www.sotera.co.uk/pdf/Location specific risk model.pdf
14. Ale, B., Bellamy, L.J., Cooke, R.M., Goossens, L.H.J., Hale, A.R., Roelen, A.L.C., Smith, E.: Towards a causal model for air transport safety – an ongoing research project. Safety Science 44, 657–673 (2006)
15. Bearfield, G., Marsh, W.: Generalising Event Trees using Bayesian Networks with a Case Study of Train Derailment. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 52–66. Springer, Heidelberg (2005)
16. Marsh, D.W.R., Bearfield, G.J.: Merging Event Trees using Bayesian Networks. In: ESREL 2007, Stavenger, Norway (to appear, 2007)

Human Error Analysis Based on a Semantically Defined Cognitive Pilot Model

Andreas Lüdtke and Lothar Pfeifer

OFFIS Institute for Information Technology, 26121 Oldenburg, Germany
{luedtke,pfeifer}@offis.de

Abstract. In this paper an approach to formal analysis of potential human errors in the interaction with mode-based systems in modern aircraft cockpits is presented. We developed a cognitive model of pilot behaviour that is integrated with system design models in order to predict human errors and the resulting safety impact due to cognitive adaptation to frequently experienced flight scenarios during pilot-cockpit interaction. The paper focuses on the definition of a formal semantics for the pilot model as a basis for formal verification of pilot-system interaction. It is shown how formal verification can support debugging formal specifications of nominal flight procedures as well as producing Human Error Fault Trees.

1 Introduction

Modern interactive avionics systems like Autopilots or Flight Management Systems are equipped with a huge number of different modes. Generally a mode may be understood as a system configuration with a specific functionality. The mode concept allows the use of systems in a variety of different operating conditions but at the same time it becomes difficult for the operators to retain "mode awareness". Mode related problems have been identified by numerous researches, e.g. by Sherry et al. [1] in the Vertical Navigation function of Flight Management Systems. A study conducted by the Federal Aviation Administration (FAA) Human Factors Team highlighted a lack of mode awareness as one major concern in the current aviation system [2]. Lack of mode awareness may lead to mode errors where an action is performed that is correct in some modes but not in the present one. Mode errors lead to "automation surprises", where an operator no longer understands what the system is doing. During the design process the need for modes has to be balanced against the probability of mode errors.

In the industry human errors are often considered very late in the system development process when a prototype is available for flight simulator studies with test and line pilots. At this stage changes to improve the usability are very time-consuming and expensive. New approaches to an earlier human error analysis (HEA) provide techniques to automatically analyse system design models and rely on Norman's [3] assumption that operators use mental models to guide their interaction with machines. In design-centred approaches, formal verification is used on a combination of mental user models and system models

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 134–147, 2007.

to identify potential human errors. Examples of these are [4] [5]. The advantage is that a complete analysis can be performed, but the question is, how to generate psychological plausible mental models. In user-centred approaches, human simulation is performed to predict user behaviour. These are discussed in [6] [7]. The advantage here is that human cognitive processes can be considered explicitly; a disadvantage is that simulation can never be complete. We suggest the integration of both approaches by (1) generating psychologically plausible mental models through simulation of cognitive learning processes, (2) automatically translating these models into a design notation in order to (3) perform formal verification to automatically compute potential pilot errors (presented as a Fault Tree). In this way, simulation and verification complement each other and the mental model is the mediating concept. Parts of this work have been carried out in the ISAAC (Improvement of Safety Activities on Aeronautical Complex Systems) project[1], funded by the European Commission under the 6th Framework Programme.

We understand mental models as knowledge about how to operate a system during concrete flight procedures (e.g. takeoff). Operators adapt their mental models while they gain experience of a particular system. The psychological theory "Learned Carelessness" [8] states that humans have a tendency to neglect safety precautions if this has immediate advantages, e.g. it saves time. Careless behaviour emerges if safety precautions have been followed several times but would not have been necessary, because no hazards occurred. Then, people deliberately omit safety precautions because they are considered a waste of time. Learned Carelessness is characteristic for human nature because we have to implicitly simplify in order to be capable to perform efficiently in a complex environment. Unfortunately this may be disastrous in non-routine scenarios. In the context of mode based avionics systems safety precautions may be understood as checking the current mode before performing critical actions (e.g. pressing buttons). We think it is crucial to consider the cognitive process of Learned Carelessnes during system design. Cognitive architectures, where the human cognitive system is understood as an information processing system, provide a framework for building models of human cognitive processing. Well-known examples are ACT-R [9] and SOAR [10]. From an engineering point of view, they can be understood as vehicles to run and modify mental models. We developed a cognitive model (implemented in PROLOG and C) in which we used concepts from ACT-R but modified them to model Learned Carelessness. The model is capable to interact in a closed-loop simulation with formal system designs developed with the commercial case tool StateMate. The simulation starts with a normative mental model of a flight procedure (called procedure model). During simulation, the procedure model is modified/simplified by the cognitive learning process. At any time, the current status of the procedure model can be translated to the state-based input language of the OFFIS model checker tool called ModelCertifier. Model checking allows (1) to identify inconsistencies, incompleteness and errors in the normative procedure model, and (2) to generate a Fault Tree

[1] www.isaac-fp6.org

that points out possible safety requirement violations due to pilot errors caused by Learned Carelessness.

The paper describes the role of formal verification during our HEA methodology (Section 2) and presents a formal semantics for the pilot model (Section 3). The semantics serves as a reference for the algorithm that translates the procedure model to the state-based input language of the OFFIS model checker. Our work is guided by the semantics for StateMate defined in [11]. Similar to that approach we use synchronous transition systems. The paper closes with briefly presenting some results of an evaluation of the HEA methodology (Section 4).

2 Human Error Analysis (HEA) Methodology

This section describes the seven steps of our HEA methodology. Steps 1 – 4 serve to prepare the needed input models for the simulation in step 5. Step 6 and 7 analyse and then harvest the results of the simulated learning process.

In step 1 (*System Design*) a formal system design has to be modeled in StateMate. In ISAAC a highly reusable modeling structure encompassing the mode logic (set of modes and mode transitions) and the corresponding control laws was developed. It allows to describe the system from the pilot's point of view. In step 2 (*Procedure Model*) a procedure model is produced that prescribes how to operate the system during a certain flight task. It is a normative model. The main components are Goals, Operators, Conditions, and Rules (similar to the famous GOMS [12] notation). Figure 1a shows a simple rule example with the informal meaning: "If you want to fulfill the goal PRESS_VERTICAL_SPEED_BUTTON and the current VERTICAL_MODE as retrieved from memory is 4, then press the VERTICAL_SPEED_BUTTON". Further details will be given in the next section. A procedure can be applied in many operational situations. These different situations are termed "scenarios". The methodology foresees to model scenarios in step 3 (*Scenario Model*) using StateMate. In ISAAC a modeling structure for scenario templates has been developed. A scenario is defined by an initial state and a list of events (e.g. ATC clearances) and describes day-to-day normal revenue flights. Instead of defining concrete values for the variables representing events and initial state data it is possible to define probability distributions over possible values. Before the simulation can be started it has to be verified in step 4 (*Debugging*) that the procedure model is indeed normative, in the sense that the application of the rules does not violate functional requirements. In this phase model checking technology is used to analyse all possible interactions between procedure model and system model. To allow this investigation the procedure model is automatically translated into the state based design notation and tightly integrated with the system model. In this form the integrated model can be analysed using the OFFIS ModelCertifier:

– Reachability analysis: to analyse if there is any goal or operator that can never be applied.

- Completeness analysis: to analyse if there is any subgoal g for which there is no rule with g as a main goal.
- Functional requirements: to analyse if the top goal of the modeled procedure may be reached, e.g. during a descend procedure the top goal is to reach and maintain a lower altitude cleared by the ATC. The requirement to be checked can be derived by formalising the desired state in LTL (Linear Temporal Logic): G(CLEARANCE_DESCEND $\Rightarrow$ FG(CURRENT_ALTITUDE = CLEARED_ALTITUDE)), where G stands for *globally* and FG for *finally globally*. In case of a possible violation the model checker generates the result *false* and a simulation run with a counter example.

For step 5 (*Human Simulation*) we implemented a platform that integrates the pilot model and system model with a flight simulator software to allow a closed loop simulation. The flight simulator software provides the environment model, including the aircraft dynamics. Before the first simulation run is started, the procedure model has to be uploaded to the cognitive architecture. The platform allows to expose the pilot model to a large number of scenarios without user interaction (batch mode). This is necessary to generate realistic results for the Learned Carelessness process implemented in the cognitive architecture. At the beginning of each run a scenario is randomly chosen. Afterwards, all scenario variables are randomly instantiated with initial values according to the defined probability distributions. During simulation the Learned Carelessness mechanism of the cognitive architecture modifies the procedure model according to the simulated scenarios. This means that new rules are added to the procedure. These rules may be incorrect, meaning that they derive pilot actions in flight situations where they are not allowed, e.g. pressing a button in a certain mode, where it is prohibited. For example, the rule in Figure 1a may be simplified by deleting the memory-retrieval and the condition (see Figure 1b). Based on this new rule the pilot model would press the VERTICAL_SPEED_BUTTON independent of the current vertical mode. This simplification is generated if repeatedly the memory retrieval of the current mode delivered the same value and subsequently the condition was always true. According to Learned Carelessness the pilot will in such cases consider checking the current mode a waste of time. Further details can be found in [13].

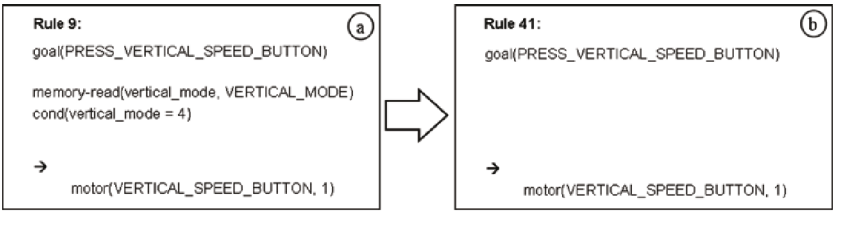


Fig. 1. Example of a normative rule (a) and simplified rule (b)

After the learning process has reached a stable state, step 6 (*Human Error Fault Tree*) is performed to check if there are scenarios in which the learned procedure model leads to pilot errors that are not covered by the system model. In order to exhaustively analyse the impact of such errors we apply model checking techniques again. In a subtheme of ISAAC, dealing with automating traditional safety analysis methods, an automatic Fault Tree Analysis based on model checking for StateMate models was developed [14]. Apart from the StateMate model this analysis needs a set of failure modes and a formal safety requirement (in LTL) as input. The failure modes are automatically injected in the model producing an *extended model*. Model checking is then used to analyse if one of the failure modes or an arbitrary combination (potential basic events) leads to a violation of a safety requirement (top level event). In the case of Human Error Analysis the set of failure modes is generated by treating every learned rule as a potential failure mode. The safety requirements are derived by formalising the top level procedure goal. Different from the analysis in step 4, here the goal is used in a negative version, e.g. $\neg(G(\text{CLEARANCE\_DESCEND} \Rightarrow FG(\text{CURRENT\_ALTITUDE} = \text{CLEARED\_ALTITUDE})))$ – roughly this means: "if an altitude clearance is emitted the aircraft must not under- or overshoot the cleared altitude". Like in step 4 the procedure model is automatically translated into the state based design notation and tightly integrated with the system model. The generated tree presents causal relationships between a violated safety requirement (depicted as top level event) and learned rules (depicted as basic events). The fault tree in Figure 2 indicates that the learning process produced two rules (rules 41 and 42) that may cause an altitude over- or undershoot. For example, rule 41 (see also Figure 1) represents the behaviour in which the pilot presses the vertical speed button without checking the current mode. In step 7 (*Design or Procedure Improvement*) either the system or the procedure design has to be improved in order to prevent or mitigate indentified pilot errors. The analysis has to be repeated iteratively until no more errors can be found: Fault Tree with no basic events.

Model checking technologies are applied in step 4 and 6. Since realistic system models are very complex, these models have to be abstracted for this analysis: a

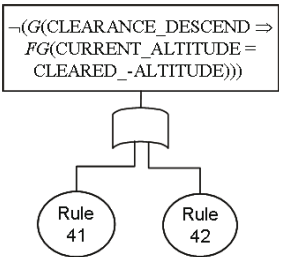


Fig. 2. Example of a Human Error Fault Tree

certain resolution has to be defined for each floating point variable and bounds have to be defined for each integer. Moreover, a simple StateMate model of the environment has to be added to the system model to replace the flight simulator software. We used very simple environment models that are tailored to the safety requirements under investigation. The procedure model is automatically translated into the state based input language of the model checker. The technical challenge was to define the translation in a way that preserves the model semantics. In order to guide the development of the translation algorithm we defined a reference semantics for the pilot model. This semantics is introduced in the next section.

3 Formal Semantics for the Cognitive Pilot Model

In this section we define a formal semantics of the cognitive pilot model. The main parts of the model are a percept component (to retrieve values from the environment), a motor component (to manipulate variables in the environment), a short-term memory (in which values from the environment are stored), a long-term memory (in which procedure rules are stored) and a knowledge processing component (which selects and fires rules from the long-term memory according to the current state of the short-term memory). The semantics below describes the details of the knowledge processing component.

The syntax grammar for rules is given in Figure 3. In general, a rule consists of a goal-part, a state-part and a means-part. Rules are selected and fired by

```
<rule> ::=  <goal-part>  <state-part>  '→'  <means-part>

<goal-part>    ::=  'Goal' ( GOAL_NAME )
<state-part>   ::=  { 'Memory-Read' '(' (INSTRUMENT_NAME), ( VARIABLE_NAME) ')' }
                    [ 'Cond' ( BOOLEAN_EXPRESSION ) ]
<means-part>   ::=  { 'Memory-Store' '(' (INSTRUMENT_NAME), ( VARIABLE_NAME) ')' }
                    { 'Percept' '(' (INSTRUMENT_NAME), ( VARIABLE_NAME) ')' }
                    { 'Motor' '(' ( INSTRUMENT_NAME ), ( VARIABLE_NAME ) ')' }
                    { 'Goal' ( GOAL_NAME ) }
```

Fig. 3. Grammar for procedure rules

the knowledge processing component in a so called *cognitive cycle*. This cycle is started as soon as one of the goals is triggered by an event or condition in the environment. During the cycle rules are selected to fulfill the goal. This leads to actions and new goals (subgoals). The cycle is repeated until all derived subgoals have been fulfilled. The semantics of the knowledge processing component is defined as a synchronous transition system (STS). In the sequel, we first introduce the STS, afterwards an abstract syntax of the main procedure model constituents is defined. Next, the concepts needed to specify the individual steps

of the cognitive cycle are handled, before finally the execution of the cognitive cycle is defined.

3.1 Synchronous Transition System

A transition system

$$\Phi = (V, \Theta, \rho)$$

is given by

- a typed set of variables V and a typed data domain $\mathcal{D}$,
- a set $\Theta \subseteq \Sigma(V)$ of initial valuations,
- a transition relation $\rho \subseteq \Sigma(V) \times \Sigma(V)$.

The valuation of a variable v is a type preserving mapping $\sigma : V \to \mathcal{D}$, $\Sigma(V)$ denotes the set of all valuations on V. The domains used in the pilot model are of the types Integer and Float.

A run π of a transition system Φ is defined as a finite or infinite sequence of valuations

$$\pi = \sigma_0 \to \sigma_1 \to \sigma_2 \to \ldots$$

with $\sigma_0 \in \Theta$ and $\forall i \in \mathbb{N} : \ (\sigma_i, \sigma_{i+1}) \in \rho$.

In the context of the pilot model V consists of the following subsets

- $V_{interface}$ is defined as the set of variables representing the values provided by the system and environment model.
- V_{local} represents the short-term memory of the pilot model.
- V_{tmp} represents variables that are local to the rules.

3.2 Formal Definition of the Pilot Model Key Concepts

The key concepts of the pilot model are goals, operations, conditions and rules.

Goals. G is a set of goal names. Every $g \in G$ comes equipped with two sets:

- A set of predecessor goals $pre(g)$ with $pre(g) : G \to P(G)$. During the cognitive cycle every predecessor goal has to be fulfilled before the goal g can be selected.
- A set of successor goals $post(g)$ with $post(g) : G \to P(G)$. During the cognitive cycle these are goals that have to be fulfilled before g can be considered as fully achieved.

A subset of G called G_{init} denotes the set of initial goals of the pilot model. For every goal in G_{init} a trigger is defined: $trigger : G \to Bool$. This trigger is a Boolean condition specifying situations in which g becomes a candidate for goal selection.

Operations. Operations are the functional elements of the pilot model. The pilot model supports four types of operations:

- Memory-read operations represent the ability of the pilot model to *recall* perceived data. $memory\_read(u, d)$ is an expression of type $\mathcal{D}$, where $\mathcal{D}$ is the type of u, $u \in V_{tmp}$ and $d \in V_{local}$. The operation assigns the value of d to u:

$$[\![memory\_read(u, d)]\!](\sigma)(v) = \sigma[memory\_read(u, d)](v),$$

  with $v \in V$, is the state that agrees with σ except for u where its value is d. OP_{mrd} is the set of all memory-read operations.
- Memory-write-operations represent the ability of the pilot model to *memorize* perceived data.
  $memory\_write(d, u)$ is an expression of type $\mathcal{D}$, where $\mathcal{D}$ is the type of d, $d \in V_{local}$ and $u \in V_{tmp}$. The operation assigns the value of u to d:

$$[\![memory\_write(d, u)]\!](\sigma)(v) = \sigma[memory\_write(d, u)](v)$$

  with $v \in V$, is the state that agrees with σ except for d where its value is u. OP_{mwr} is the set of all memory-write operations.
- Percept-operations: Percept-operations represent the pilot model's ability to *perceive* data from the environment and system model. $percept(u, d)$ is an expression of type $\mathcal{D}$, where $\mathcal{D}$ is the type of u, $u \in V_{temp}$ and $d \in V_{interface}$. The operation assigns the value of d to u:

$$[\![percept(u, d)]\!](\sigma)(v) = \sigma[percept(u, d)](v)$$

  with $v \in V$, is the state that agrees with σ except for u where its value is d. OP_p is the set of all percept operations.
- Motor-operations: Motor-operations represent the pilot model's ability to *manipulate* the system model. $motor(d, u)$ is an expression of type $\mathcal{D}$, where $\mathcal{D}$ is the type of d, $u \in V_{temp}$ and $d \in V_{interface}$. The operation assigns the value of u to d:

$$[\![motor(d, u)]\!](\sigma)(v) = \sigma[motor(d, u)](v)$$

  with $v \in V$, is the state that agrees with σ except for d where its value is u. OP_m is the set of all motor operations.

Rules. The dynamic behavior of the pilot model stems from firing rules. The set of rules is denoted by R. Every rule r comes equipped with a set of relations providing the individual elements of the various parts:

- The function $goal : R \to G$ provides the goal in the goal-part. This is the goal which shall be achieved by the rule.

- The function $mem\_read\_ops : R \rightarrow \mathcal{P}(OP_{mrd})$ provides the memory-read operations. These operations retrieve variables from the short-term memory. The memory represents a mental "image" of the actual environmental state. This relation is defined on sets of rules as follows:
  $mem\_read\_ops : \mathcal{P}(R) \rightarrow \mathcal{P}(OP_{mrd})$ with

$$mem\_read\_ops(R_i) = \bigcup_{r \in R_i} mem\_read\_ops(r) \ .$$

- The function $cond\_op : R \rightarrow Bool$ provides the Boolean expression. The condition specifies constraints over the mental environment that must hold in order to apply the rule. A condition contains only variables from V_{tmp}. For every variable in a condition there has to be a corresponding memory-read operation assigning the value of a local variable to a corresponding tmp variable.
- The function $mem\_write\_ops : R \rightarrow \mathcal{P}(OP_{mwr})$ provides the set of memory-write operations. These are used to store values in the memory component.
- The function $percept\_ops : R \rightarrow \mathcal{P}(OP_p)$ provides the set of percept operations. During rule execution percept-operation are sent to the percept-component.
- The function $motor\_ops : R \rightarrow \mathcal{P}(OP_m)$ provides the set of motor operations. During rule execution motor-operations that are sent to the motor component.
- The function $subgoals : R \rightarrow \mathcal{P}(G)$ provides the set of subgoals in the means-part. This set is partially ordered defined by the relation $\sqsupseteq$.

Subsymbolic Concepts. On a subsymbolic level the strength of a rule is defined by the number of successful applications (*successes*) and the number of erroneous applications ($failures$). These parameters are used in the cognitive cycle to select the rule with the highest probability of success. During the simulation these values are adapted by the learning process according to the success or failure of a run. Successes and failures are used to compute the strength of a rule as

$$rule\_strength : R \rightarrow [0, \ldots, 1]$$
$$rule\_strength(r) = \frac{successes(r)}{successes(r) + failures(r)} \ .$$

3.3 The Dynamic Behaviour of the Pilot Model

The following concepts are used to define the dynamics of the cognitive cycle.

Execution Concepts. Every valuation σ contains exactly one dedicated set called goal agenda GA, where

- GA is a subset of G. It denotes the set of goals that are candidates for goal selection.

– The goals in GA form a set of trees with partially ordered leaves. Goals in $G_{init} \cap GA$ are defined as roots. The tree structure is derived from the post sets

$$\text{for } g_i, g_j \in G: \ child(g_i) = g_j \text{ iff } g_j \in post(g_i)$$

The goal agenda contains those goals that the pilot model has to fulfill during execution of a task. In every valuation only a subset of GA is ready to be chosen during goal selection. A goal $g \in G$ is selectable in σ, denoted by $\sigma \models selectable(g)$, iff all predecessor goals have been fulfilled before and no successor goals have been derived so far:

$$\sigma \models selectable(g) \text{ iff}$$

– $g \in \sigma(GA)$ and
– the set of predecessors of g is empty: $\sigma(pre(g)) = \emptyset$ and
– the set of successors of g is empty: $\sigma(post(g)) = \emptyset$.

Every valuation σ contains exactly one dedicated goal g_{active} where

– $\sigma \models selectable(g_{active})$ or $g_{active} = undef$.

Every valuation σ contains exactly one dedicated set called conflict set CS, where

– $CS \subseteq R$ and
– either $CS = \emptyset$ or
– $CS = \{r \in R | goal(r) = g_{active}\}$ or
– $CS = \{r \in R | goal(r) = g_{active} \wedge [\![cond\_op(r)]\!]\sigma = \text{true}\}$.

A rule $r \in R$ is selectable in σ, denoted by $\sigma \models selectable(r)$, iff the goal in the goal-part of r is g_{active} and the condition evaluates true:

$$\sigma \models selectable(r) \text{ iff}$$

– $goal(r) = g_{active}$ and $[\![cond\_op(r)]\!]\sigma = \text{true}$.

Every valuation σ contains exactly one dedicated rule $r_{selected}$, where

– $\sigma \models selectable(r_{selected})$ or $r_{selected} = undef$.

Cognitive Cycle. The cognitive cycle describes the procedure processing of the cognitive architecture. In the sequel, we define six transition subrelations. These are used afterwards to compose the main transition relation for the cognitive cycle. The six relations correspond to the individual steps of the cognitive cycle described above.

1. Select a goal $g \in GA$: $\sigma \quad \rho_{select\_goal} \quad \sigma'$ iff
   – $\sigma \models selectable(g_{active})$ and
   – $\sigma'(GA) = \sigma(GA)$ and $\sigma'(CS) = \sigma(CS)$ and

- $\sigma'(r_{selected}) = \sigma(r_{selected}) = undef$ and
- $\sigma'(v) = \sigma(v)$ for all $v \in V$

2. Build conflict set: $\sigma \quad \rho_{build\_CS} \quad \sigma'$ iff
   - $\sigma'(CS) = \{r | goal(r) = g_{active}\}$ and
   - $\sigma'(GA) = \sigma(GA)$ and
   - $\sigma'(g_{active}) = \sigma(g_{active})$ and $\sigma'(r_{selected}) = \sigma(r_{selected})$ and
   - $\sigma'(v) = \sigma(v)$ for all $v \in V$
3. Perform all memory-read operations to prepare subsequent conflict resolution: $\sigma \quad \rho_{exe\_read} \quad \sigma'$ iff
   - $\sigma'(v) = \sigma[mem\_read\_ops(CS)](v)$ and
   - $\sigma'(GA) = \sigma(GA)$ and $\sigma'(CS) = \sigma(CS)$ and
   - $\sigma'(g_{active}) = \sigma(g_{active})$ and $\sigma'(r_{selected}) = \sigma(r_{selected})$

   with $\sigma'(v) = \sigma[mem\_read\_ops(CS)](v)$ denotes the subsequent execution of every memory-read operation of every rule $r \in CS$.
4. Conflict resolution: In order to be able to chose a rule from CS two actions are performed during conflict resolution:
   (a) CS is reduced by evaluating the conditions of every rule in CS and removing those rules whose conditions evaluate to $false$. In this way the set of selectable rules is generated.
   (b) Of the remaining rules in CS the rule with the highest rule strength is chosen. If more than one rule in CS have the same maximal strength value one of these rules is randomly chosen.

   The transition relation is defined as: $\sigma \quad \rho_{conflict\_res} \quad \sigma'$ iff
   - $\sigma'(CS) = \{r | \sigma \models selectable(r)\}$ and
   - $\sigma'(GA) = \sigma(GA)$ and
   - $\exists r \in \sigma'(CS) : \forall r_j \in \sigma'(CS) :$
     $strength(r) \geq strength(r_j) : \sigma'(r_{selected}) = r$ and
   - $\sigma'(g_{active}) = \sigma(g_{active})$ and
   - $\sigma'(v) = \sigma(v)$ for all $v \in V$.
5. Fire rules: $r_{selected}$ is fired: $\sigma \quad \rho_{fire\_rule} \quad \sigma'$ iff
   - $\sigma'(CS) = \sigma(CS)$ and
   - $\sigma'(v) = \sigma[percept\_ops(r_{selected})](v) \circ \sigma[motor\_ops(r_{selected})](v) \circ$
     $\sigma[mem\_write\_ops(r_{selected})](v)$ and
   - if $subgoals(r_{selected}) \neq \emptyset$ then
     $\sigma'(GA) = \sigma(GA) \cup subgoals(r_{selected})$ and
     $\sigma'(post(goal(r_{selected}))) = subgoals(r_{selected})$ and
     $\forall g_i, g_j \in subgoals(r_{selected})$ and $g_i \sqsupseteq g_j : g_i \in \sigma'(pre(g_j))$
   - if $subgoals(r_{selected}) = \emptyset$ then
     $\sigma'(GA) = \sigma(GA) \setminus goal(r_{selected})$ and
     $\forall g \in \sigma(GA)$ and $goal(r_{selected}) \in \sigma(post(g)) :$
     $\sigma'(post(g)) = \sigma(post(g)) \setminus goal(r_{selected})$ and
     $\forall g \in \sigma(GA)$ and $goal(r_{selected}) \in \sigma(pre(g)) :$
     $\sigma'(pre(g)) = \sigma(pre(g)) \setminus goal(r_{selected})$.

6. Trigger goals: At the beginning of the cognitive cycle GA is empty. As long as no goal $g \in G_{init}$ is triggered the pilot model remains in its initial state. In every step the pilot model checks the trigger conditions of all goals in G_{init} and adds those whose trigger evaluates to *true* defined by the transition relation $\rho_{add\_triggered}$.

The main transition relation ρ for the cognitive cycle is formally defined as the product of the subtransition relations:

$$\begin{aligned}\rho = \rho_{select\_goal} || \rho_{add\_triggered} \circ \rho_{build\_CS} || \rho_{add\_triggered} \\ \circ \rho_{exe\_read} || \rho_{add\_triggered} \circ \rho_{conflict\_res} || \rho_{add\_triggered} \\ \circ \rho_{fire\_rule} || \rho_{add\_triggered}\end{aligned}$$

Initially there is no goal in GA and CS is empty, the valuation of all variables in V_{temp} and V_{local} are undefined and the variables in $V_{interface}$ are given by the state of the environment model.

Related Work. At the end of this section two related approaches to defining a formal semantics for cognitive architectures (SOAR and ACT-R) shall be briefly described. Heise and Westermann [15] conducted a structuralist analysis of an early version of ACT-R called ACT* in order to clearly extract the unique contents of ACT*. They used set-theoretic axioms to define the concepts and structural relationships. For our purpose of formal verification this semantics is too informal and lacks a definition of the dynamics. Milnes [16] constructed a formal specification of SOAR in order to guide a reimplementation of the architecture. He used the formal specification language Z to define a state-transition system. The major difference to our synchronous transition system semantics is the level of detail. Milnes defines a level of detail that allows direct mapping of the specification into executable code. Our semantics abstracts from execution details, because for the purpose of formal verification we only need to define the transition between states but not how the transitions are actually performed.

4 Evaluation Results

In the ISAAC European Project the pilot-model-based HEA methodology as described in Section 2 was evaluated by the industrial partners with two case studies: one general auto pilot model (plus airport arrival procedure) and a reduced version of the Airbus A340 auto flight system (plus takeoff procedure). Due to space limitation we can only highlight the most important findings. More details can be found in [13].

Building the models (system, procedure, scenarios) that are necessary as input for the HEA methodolgy was a very time consuming process. The complexity of the models had to be reduced due to constraints imposed by the methodology. Some constraints stem from the current state of development of the pilot model. In the current development state it does not allow multitasking. All (sub-)tasks

have to be performed in sequence. This is no problem when dealing with short subtasks. But, especially the takeoff procedure requires longer periods of observing the flight mode or speed annunciation. The lack of multitasking leads to some overshoots of speed and altitude limits. Thus we had to broaden the accepted envelops in the safety requirements. Multitasking is planned for the next extension of the architecture. Further limitations due to the intended model focus are: no crew task sharing, no memory effect (like forgetting), no sophisticated perception.

It was acknowledged that a number of erroneous actions due to learned rules have been observed and then discussed with pilots who have good knowledge of the scenarios that were simulated. As an example with reference to the arrival procedure, altitude selection errors were observed; analysis showed that such errors were due to simplification of normative procedure induced by the "Learned Carelessness" mechanism. Results gathered so far appear to be plausible in terms of observable behaviour. However, subject matter experts highlighted that "Learned Carelessness" may be a cause for such errors only under specific environmental circumstances (e.g. high workload, rush operations). This allows deriving an indication for improving the cognitive architecture with regard to workload and workload inducing factors. Testers stated that though the pilot model focuses on "Learned Carelessness" only, it is possible to derive indications that may be useful to support the identification of system improvements. But it was noted that due to the scope of the current method it is necessary to thoroughly discuss and evaluate the results with experts.

5 Summary and Future Work

In this paper we presented a new methodology for Human Error Analysis during system design. The focus was on defining a formal semantics for an executable cognitive pilot model that forms the basis for the methodology. Based on this definition it is possible to translate the pilot model into the state-based input language of a model checker tool and to apply formal verification to analyse human errors (here caused by Learned Carelessness) in the pilot-system interaction in a mathematical exhaustive way. In future work we intend to extend the pilot model to include multitasking aspects and further cognitive error mechanisms like incorrect prioritizing of goals. In parallel we will extend the formal semantics in order to provide a clear reference for formal analysis algorithms like model checking.

Acknowledgements. The definition of the HEA methodology was done in cooperation with Cavallo, Fabbri, Christophe, Cifaldi and Javaux in ISAAC. The semantics definition and formal verification was performed by the authors of the paper. The ISAAC project was funded by the European Commission in the 6th Framework Programme, Aeronautics and Space, FP6-2002-Aero-1-501848.

References

1. Sherry, L., Feary, M., Polson, P., Mumaw, R., Palmer, E.: A cognitive engineering analysis of the vertical navigation function. Technical report, NASA Ames Research Center (2001)
2. Lyall, B., Wilson, J.: Flight deck automation issues. Technical report, Oregon State University and Research Integrations, Incorporated (1997)
3. Norman, D.A.: The psychology of everyday things. Basic Books, New York (1988)
4. Rushby, J., Crow, J., Palmer, E.: An automated method to detect potential mode confusions. In: 18th Digital Systems Avionics Conference (1999)
5. Degani, A., Heymann, M.: Formal verification of human-automation interaction. In: Human Factors, vol. 44(1) (2002)
6. Corker, K.M.: Cognitive models and control. In: Sarter, N.B., Amalberti, R. (eds.) Cognitive Engineering in the Aviation Domain, Mahwah, NJ, LEA (2000)
7. Freed, M.A., Remington, R.W.: Making human-machine system simulation a practical engineering tool: An apex overview. In: Proceedings of the International Conference on Cognitive Modelling, Groningen, Holland (2000)
8. Frey, D., Schulz-Hardt, S.: Eine Theorie der gelernten Sorglosigkeit. In: Mandl, H. (ed.) H. Mandl (Hrsg.), Bericht über den 40. Kongress der Deutschen Gesellschaft für Psychologie, Göttingen: Hogrefe Verlag für Psychologie, pp. 604–611 (1997)
9. Anderson, J.R., Bothell, D., Byrne, M.D., Douglass, S., Lebiere, C., Qin, Y.: An integrated theory of the mind. Psychological Review 111 4, 1036–1060 (2004)
10. Lewis, R.: Cognitive theory, soar. In: International Encylopedia of the Social and Behavioral Sciences, Pergamon (Elsevier Science), Oxford (2001)
11. Damm, W., Josko, B., Hungar, H., Pnueli, A.: A compositional real-time semantics of STATEMATE designs. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 186–238. Springer, Heidelberg (1998)
12. Card, S.K., Moran, T.P., Newell, A.: The Psychology of Human-Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, NJ (1983)
13. Lüdtke, A., Cavallo, A., Christophe, L., Cifaldi, M., Fabbri, M., Javaux, D.: Human error analysis based on a cognitive architecture. In: Reuzeau, F., Corker, K., Boy, G. (eds.) Proceedings of HCI-Aero 06, pp. 40–47. Cépaduès-Editions (2006)
14. Peikenkamp, T., Cavallo, A., Valacca, L., Böde, E., Pretzer, M., Hahn, E.M.: Towards a unified model-based safety assessment. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 275–288. Springer, Heidelberg (2006)
15. Heise, E., Westermann, R.: Anderson's theory of cognitive architecture (act*). In: Westmeyer, H. (ed.) Psychological theories from a structuralist point of view, pp. 103–127. Springer, Berlin (1989)
16. Milnes, B., Pelton, G., Doorenbos, R., Laird, M., Rosenbloom, P., Newell, A.: A specification of the soar cognitive architecture in z. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA (1992)

Safety Analysis of Safety-Critical Software for Nuclear Digital Protection System

Gee-Yong Park[1], Jang-Soo Lee[1], Se-Woo Cheon[1], Kee-Choon Kwon[1], Eunkyoung Jee[2], and Kwang Yong Koh[2]

[1] Korea Atomic Energy Research Institute, 150 Deokjin, Yuseong, Daejeon, 305-353, Korea
{gypark,jslee,swcheon,kckwon}@kaeri.re.kr
[2] Korea Advanced Institute of Science and Technology, 373-1 Guseong, Yuseong, Daejeon, 305-701, Korea
ekjee@dependable.kaist.ac.kr, goeric1@kaist.ac.kr

Abstract. A strategy and relating activities of a software safety analysis (SSA) are presented for the software of a digital reactor protection system where software modules in the design description are represented by function blocks (FBs). The SSA, as a part of the verification and validation activities, was activated at each phase of the software lifecycle. For the SSA of the FB modules, the software HAZOP was performed and then the SFTA (Software Fault Tree Analysis) was applied. Both methods are redundant and complementary because the software HAZOP is a forward broad-thinking analysis method and the SFTA is a backward step-by-step local analysis method. The software HAZOP with qualitative properties for a deviation evaluated all the software modules and identified various hazards. The SFTA with well-defined FB fault tree templates was applied to some critical modules selected from the software HAZOP analysis and it identified some hazards that had not been identified in the prior processes of the document evaluation and the formal verification.

Keywords: Software Safety Analysis, Software FTA, Software HAZOP, Function Block Diagram, Nuclear Reactor Protection System.

1 Introduction

A fully-digitalized reactor protection system (RPS), which is called the IDiPS, is being developed under the KNICS (Korea Nuclear Instrumentation & Control Systems) project in order to be used in newly-constructed nuclear power plants and also in the upgrade of existing analog-based RPSs [1]. The IDiPS has four channels which are located in electrically and physically isolated rooms. The IDiPS generates the reactor trip signals and the engineered safety features (ESF) actuation signals automatically whenever the monitored process variables reach their predefined setpoints. Fig.1 shows the overall architecture of a single channel of IDiPS. A single channel IDiPS is composed of bistable processors (BPs), coincidence processors (CPs), an automatic test and interface processor (ATIP), and a cabinet operator

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 148–161, 2007.

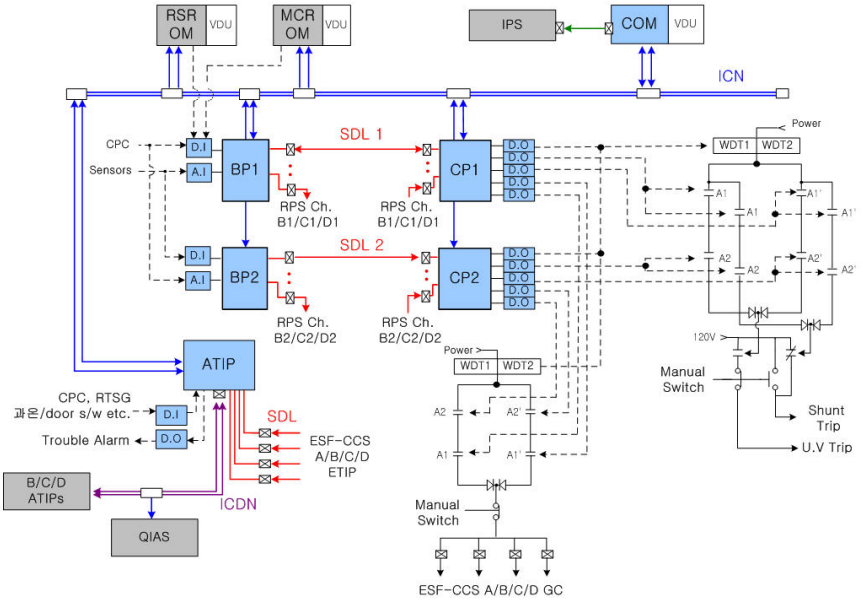


Fig. 1. Architecture of the KNICS reactor protection system, IDiPS

Note - RSR: Remote Shutdown Room, **MCR:** Main Control Room, **OM:** Operator Module, **IPS:** Information Processing System, **CPC:** Core Protection Calculator, **QIAS:** Qualified Information & Alarm System, **WDT:** Watchdog Timer.

module (COM). It contains three different networks such as the intra-channel network (ICN), safety data link (HR-SDL), and inter-channel data network (ICDN).

The BP determines the trip state and generates a trip signal by comparing the measured process variables with the predefined trip setpoints. The CP generates a final hardware-actuating trip signal by a two-out-of-four voting logic. The ATIP generates the test signals for a manual test and a manual initiated automatic test. Moreover, the ATIP performs IDiPS status indications and also performs integrity tests to verify the operational status of the BP and CP. The COM comprises of two parts: (a) a computer based part that provides and displays the status information regarding the overall IDiPS equipments, and (b) a hardware based part that performs protection-related controls such as a channel bypass and an initiation circuit reset.

In the IDiPS, the trip functions such as a signal comparison in the BP and a voting logic in the CP are implemented in the software. Hence, the software in the IDiPS is crucial to the safety of a nuclear power plant in that its malfunction may result in irreversible consequences. The software in the BP and CP of IDiPS is classified as a safety-critical class and it is mandatory that the software safety analysis be performed on all the safety-critical software. In the KNICS project, the software used in the IDiPS is being developed under a rigorous procedure [1]. Also, independent verification & validation (V&V) activities are being arranged [2][3]. The IDiPS is configured based on the POSAFE-Q PLC-based platform. The software of the IDiPS

is programmed by the use of a function block diagram (FBD) which is compliant with the standard of IEC 61131-3 [4]. And the software modules in the detailed design description are represented by the FBD. In order to follow the nuclear regulation and also to improve the software quality, the software safety analysis (SSA) is being performed as a part of the V&V activities. This paper describes the software safety analysis performed on the FBD modules.

2 Strategy of SSA

It is recommended in the code and standards that the SSA shall be performed during the development of the software used for a safety system of nuclear power plants [5], [6]. For the strategy for performing the SSA, various methods have been proposed in the literature and in the research reports. The report by LLNL (Lawrence Livermore National Laboratory) [7] suggested that the SSA start from the hazard analysis of the target system and a software hazard analysis be performed at each phase of a software lifecycle. For the software hazard analysis to be applied to the phases of a software lifecycle, the software HAZOP (Hazard and Operability) was proposed. The strategy proposed by Leveson [8] has a distinctive characteristic in the safety verification. It is suggested that the safety analysis at any phase is to check against the safety requirements and constraints, being independent of the software requirements and design specifications, rather than checking the consistency such that the current implementation is in compliance with the specifications concerning the safety at the previous phase.

For the KNICS project, Lee and his colleagues have proposed a proper software safety lifecycle [9] by investigating various standards including IEC and IEEE standards. This safety process was adopted by the KNICS project where not only a digital reactor protection system but a programmable logic controller (PLC), POSAFE-Q, with proprietary operating software is being developed. Fig.2 depicts the software safety lifecycle for the KNICS project. As can be seen in Fig.2, the SSA is activated at each software lifecycle, starting from the establishment of a software safety plan where the SSA is planned as a part of the system safety analysis. The software-contributable hazards necessary for the SSA are extracted from the system hazard analysis by FMEA (Failure Modes and Effects Analysis). For the SSA at the requirements phase, the software HAZOP is employed in the hazard analysis. Also the software HAZOP and SFTA techniques are used for the SSA at the design and implementation (code) phases.

3 Identification of Software-Contributable Hazards

For the SSA of the FBD modules for a detailed design description and also for a FBD program, the software HAZOP is performed at first and then SFTA is applied. For these two techniques to be applicable, the software-contributable system hazards and the interface points between the system hazard and the application software must be identified. Based on these hazards and interface points, the software HAZOP identifies a software hazard, which can affect a certain system hazard, by applying a qualitative deviation into a FBD module. Also the interface points provide the

necessary information about the top node event of the SFTA for the FBD modules that are found to be defective, resulting from the software HAZOP analysis.

The system hazards were identified from an FMEA for the IDiPS hazard analysis by a digital system safety engineer. The FMEA identified some failure modes due to the trip-functioning application software which can affect the system safety. The interface points between the trip-functioning safety-critical software and the system hazards were identified by examining the analysis results of the FMEA. Table 1 represents the software-contributable system hazards and their criticality level. The criticality level implies the severity of a hazard on the system and plant safety and it is divided into the four levels. Level 4 means the most critical hazard that can induce the severe plant accident resulting in a disaster. Level 3 implies a hazard that can induce a significant impact on the system operation but does not lead to an accident of a nuclear power plant, and level 2 means a hazard that can affect more or less the system operation. And finally, level 1 indicates an insignificant hazard.

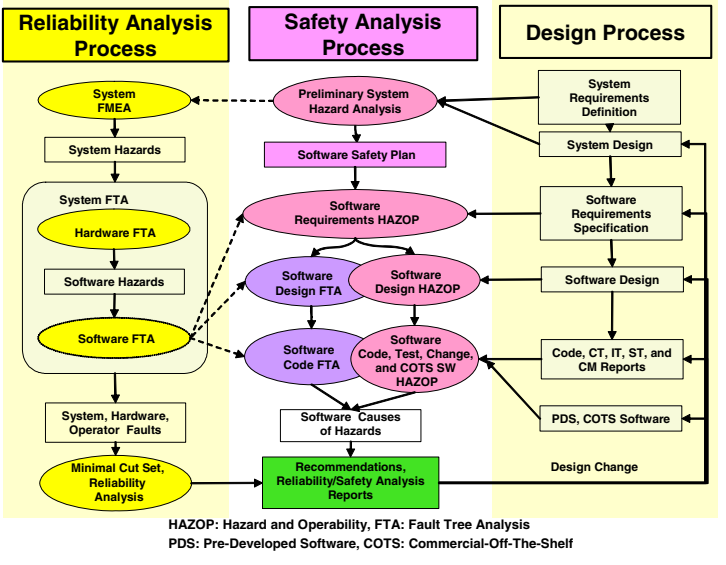


Fig. 2. Software safety lifecycle for KNICS RPS and PLC systems

Table 1. Software-contributable system hazards and criticality level for IDiPS RPS

Item No.	Software Contributable Hazards	Criticality Level
1	IDiPS cannot generate a trip signal when a trip condition for a process variable is satisfied.	4
2	IDiPS generates a trip signal when it should not generate a trip signal.	3
3	IDiPS cannot send qualified information of its operating status to the main control room for plant operators.	2

The first hazard item contradicts the purpose of the IDiPS which is to protect the nuclear reactor core at any situation and this hazard can drive a nuclear power plant into a fatal unsafe state. The second hazard item does not affect the overall plant safety but it influences adversely on the economic operation of the system. The third hazard item may be induced from an omission of some necessary information or a transfer of incorrect information during the execution of the application software. This hazard can also be generated from a malfunction of the communication software, but this is not the scope of the SSA for the IDiPS. Instead, the hazards related to the communication software are analyzed in the SSA of PLC operating software.

The software modules of the BP which is a safety-critical processor of IDiPS are presented in Table 2. The software modules in the Trip_Logic module are all the trip-functioning modules and hence, all the final output variables in these modules are the interface points that affect the system hazard items 1 and 2 in Table 1. Some software modules in no.1 and no.3 also affect the hazard items 1 and 2 through the Trip_Logic module. The software modules in no.5, no.8, and no.9 affect the hazard item 3. The rest of the software modules are insignificant because the failures from these modules can be accommodated by the fault-tolerant functions such as a watchdog timer.

Table 2. Software modules in BP

NO	Module		Description
1	Receive_Signal		HW/SDL/ICN Receive Module
2	PAT_Scheduler		Automatic Test Scheduler
3	Test_Selection		Test Selection Module
4	Trip_ Logic	PZR_PR_Hi Trip	Pressurizer Hi Pressure Trip
		SG1_LVL_Lo_RPS Trip	SG-1 Low Level Trip
		SG1_LVL_Lo_ESF Trip	SG-1 Low Level Trip for ESF
		SG1_LVL_Hi Trip	SG-1 Hi Level Trip
		SG1_PR_Lo Trip	SG-1 Low Pressure Trip
		CMT_PR_Hi Trip	Containment Hi Pressure Trip
		CMT_PR_HH Trip	Containment Hi-Hi Pressure Trip
		SG1_FLW_Lo Trip	SG-1 Low Coolant Flow Trip
		PZR_PR_Lo Trip	Pressurizer Low Pressure Trip
		VA_OVR_PWR_Hi Trip	Variable Over Power Hi Trip
		SG2_LVL_Lo_RPS Trip	SG-2 Low Level Trip
		SG2_LVL_Lo_ESF Trip	SG-2 Low Level Trip for ESF
		SG2_LVL_Hi Trip	SG-2 Hi Level Trip
		SG2_PR_Lo Trip	SG-2 Low Pressure Trip
		SG2_FLW_Lo Trip	SG-2 Low Coolant Flow Trip
		LOG_PWR_Hi Trip	Log Reactor Power Hi Trip
		DNBR_Lo Trip	Low DNBR Trip
		LPD_Hi Trip	Hi LPD Trip
		CPC_CWP Trip	CPC CWP
5	Test_Results_Handler		Test Results Handling Module
6	HB_MONITORING		Heartbeat Monitoring Module
7	HB_Gen		Heartbeat Generation Module
8	Ch_Bypass_Send_Receive		Channel Bypass Transfer Module
9	Send_Signal		HW/SDL/ICN Sending Module

4 Software Safety Analysis for the FBD Modules

4.1 Software HAZOP

A HAZOP study is for the identification of a hazard in a target system represented by a so-called P&ID (Pipes and Instrumentation Diagram) by investigating a plausible deviation of a quantity or attribute and then seeking out the cause that is capable of inducing this deviation and the consequences resulting from this deviation. For the deviation to be performed systematically, well-defined guide words are established and then some essential questions suitable for a specific application area are devised based on these guidewords. This hazard analysis has been applied successfully to processes such as chemical plants, nuclear power plants, and so forth. And, there have been a few applications on computers and software systems [10], [11]. For HAZOP studies on the software systems, there are few papers if any that describe a systematic procedure for a HAZOP to be applied to all the software lifecycles.

In the KNICS project, a software HAZOP was developed for use in all the phases of the software lifecycle. The software HAZOP has different aspects from the conventional HAZOP applied to power plants and even to software designs, in that the quantity or attribute to be deviated is not a quantitative quantity such as a temperature value or a input data value but a qualitative functional characteristic of the software, and, moreover, the guide phrases are established for a systematic deviation rather than a set of guidewords. Thus, the software HAZOP is performed to identify some software hazards or defects that can induce one of the system hazards when a certain deviation for each functional characteristic is applied to the software system. The software functional characteristics are those proposed in NUREG0800 BTP/HICB-14 [12] such as accuracy, capacity, functionality, reliability, robustness, security, and safety. The concept of using the guide phrases was originally proposed by the LLNL report [7]. The guide phrases suited for the safety-critical software of the KNICS RPS and PLC systems and applicable to the requirements, design, and implementation phases were devised carefully [13].

While the V&V activities are performed on all the software of the IDiPS, the software on which a safety analysis is performed is the safety-critical software of the BP and CP of IDiPS. The form of the software module representation is based on the FBD. Among all the guide phrases [13], the guide phrases useful for these software representations were selected. Table 3 represents the guide phrases for the FBD-based software module descriptions and the corresponding checklist.

In Table 3, the functional characteristics have the deviations whose causes are external faults except for the four items from the bottom of the table. For the analysis of the “functionality” characteristic, the function blocks and their logical connections are inspected over a FBD module. This procedure is similar to a walkthrough during tests, possibly combined with a prior individual desk checking, but this is carried out briefly. At the test phase, this type of test is facilitated in order to find an error in the code for satisfying a specified software behavior. Being different from the test phase, the purpose of the inspection of the functionality characteristic is to find an error or hazard that can ultimately lead to one of the system hazards defined in Table 1.

Table 3. Guide phrases and checklist for the FBD-based software modules of IDiPS

Characteristic	Guide Phrase	Deviation Checklist
Accuracy	Below minimum range	What is the consequence if the sensor value is below its minimum range?
Accuracy	Above maximum range	What is the consequence if the sensor value is above its maximum range?
Accuracy	Within range, but wrong	What is the consequence if the sensor value is within its physical range but incorrect?
Accuracy	Incorrect physical units	What is the consequence if the input has an incorrect physical unit?
Accuracy	Wrong data type or data size	What is the consequence if the input has a wrong data type or data size?
Accuracy	Wrong physical address	What is the consequence if the input variable is allocated to a wrong physical address?
Accuracy	Correct physical address, but wrong variable	What is the consequence if a wrong input variable is allocated to a correct physical address?
Accuracy	Wrong variable type or name	What is the consequence if wrong type or name for an input /output/internal variable is used in the FBD module?
Accuracy	Incorrect variable initialization	What is the consequence if the input/output/ internal variables are initialized incorrectly?
Accuracy	Wong constant value	What is the consequence if the internal constant is given a wrong value?
Accuracy	Incorrect update of history variables	What is the consequence if the variable is updated incorrectly?
Accuracy	Wrong setpoint calculation	What is the consequence if the procedure for calculating a setpoint is incorrect?
Capacity	Erroneous communication data	What is the consequence if there is an error in the ICN data?
Capacity	Erroneous communication data	What is the consequence if there is an error in the SDL data?
Capacity	Unexpected input signal	What is the consequence when an unexpected input signal is arrived?
Capacity	Untimely operator action	What is the consequence if the operator commences a setpoint reset or an operating bypass function untimely?
Functionality	Function is not carried out as specified	What is the consequence if some portions in the FBD module have a defect or cannot perform the intended behavior?
Reliability	Data is passed to incorrect process	What is the consequence if the data is passed to an incorrect process?
Robustness	Incorrect selection of test mode	What is the consequence if the test mode is selected or changed unexpectedly?
Robustness	Incorrect input selection	What is the consequence if the input selection is incorrect?

The software HAZARD analysis evaluates all the software design modules with respect to all the hazards in Table 1 by applying iteratively all the items in the deviation checklist to each module, reflecting the system safety and availability. Hence this process requires a considerably large amount of time and efforts of the HAZOP members. It is very important to draw a good corporation of members to obtain a successful result from the software HAZOP. Through the software HAZOP with the checklist in Table 3, various hazards affecting the availability of the system were identified and also the software HAZOP was proven to be useful in identifying a software hazard affecting the system safety.

One of the software HAZOP analysis results is presented in Table 4 where a trip logic module, “SG1_FLW_Lo Trip” (which indicates the module of Steam Generator

Table 4. Software HAZOP analysis for SG1_FLW_Lo trip

Fun. Charac.	Deviation Checklist	Cause	Analysis	Effect	C	Suggestion
Accuracy	What is the consequence if the sensor value is below its minimum range?	Sensor Failure	The TRIP_DECISION sub-module handles properly an out-of-range value, but it is carried out after all logical operations are done.	No effect on safety, but operability is poor.	2	It is desirable that a trip signal occurs at the front when an out-of-range sensor input value exists.
Accuracy	What is the consequence if the sensor value is above its maximum range?	Sensor Failure				
Accuracy	What is the consequence if the sensor value is within its physical range but incorrect?	Sensor Failure or, Input Conditioner Malfunction	This is the problem at input conditioning processor.	Severe effect on safety	4	Measures should be provided at input processor.
Accuracy	What is the consequence if the internal constant is given a wrong value?	Wong constant value allocation	If MAXCNT is set to 0, the trip signal is always ON regardless of the trip condition status.	Poor Operability	3	Need careful attention when assigning a value.
			If MAXCNT is too large, the trip signal is generated at much later time.	Violating the system response time	4	
Capacity	What is the consequence when an unexpected input signal is arrived?	ATIP Error	No part performs an exceptional handling when ATIP sets up an erroneous test operation.	Wrong test execution	1	Augment test mode selection.
Functionality	What is the consequence if some portions in the FBD module have a defect or cannot perform its intended behavior?	Error in Logic Operation	Pretrip is cancelled whenever it is triggered at the pretrip sub-module.	Pretrip is never functioning	3	Modify a pretrip logic.
			The hysterisis is not reflected in the trip logic sub-module because of using 19th previous value.	Inducing a trip malfunction	4	Modify trip logic.

#1 Low Coolant Flow Trip) was analyzed. Table 4 shows the deviations, the causes that induce these deviations, the hazards analyzed, the effects of hazards, the criticality levels, and the suggestions for their hazards.

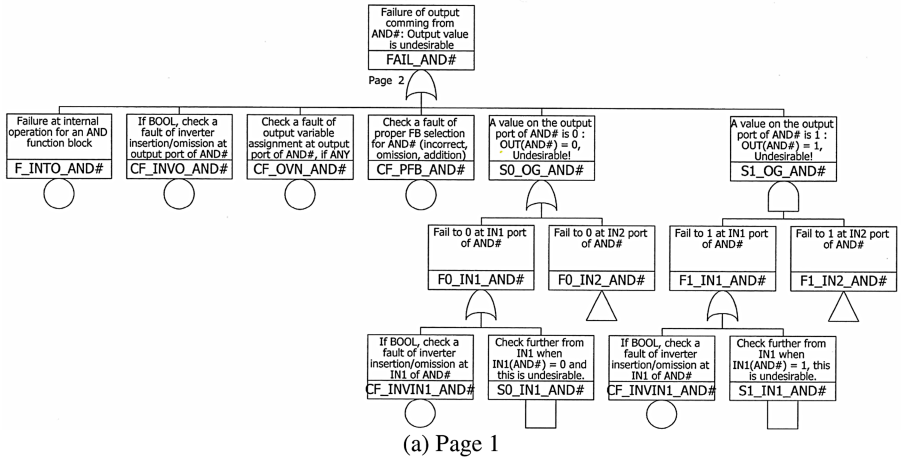
4.2 SFTA

In the activities of the SSA for the IDiPS software module descriptions, the SFTA was employed after the software HAZOP was carried out in the hazard analysis. As Leveson and Shimeall [14] mentioned the procedure of the SFTA, it is hypothesized that the software has produced an unsafe output and it is shown that this could not happen because the hypothesis leads to a contradiction. The SFTA was applied to a part of the software module descriptions with some critical defects identified by the "functionality" of the software HAZOP. The top node of the SFTA was only related to the most safety-critical hazard. Thus, the SFTA was used in a detailed analysis for a specific area with the consideration of a software defect that can affect the most significant system hazard. The software defect identified by the SFTA was mainly a software logic error or a certain input condition for the occurrence of a hazard.

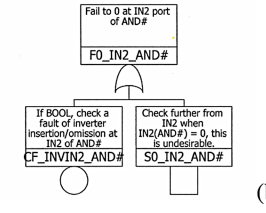
Both methods, the software HAZOP and the SFTA, are supposed to be redundant and, at the same time, complementary because the software HAZOP is a forward broad-thinking analysis method and the SFTA is a backward step-by-step local analysis method. This redundancy obviously requires an additional overlapping work on the SSA but this type of overlap is recommended by a regulatory agency and some standards because all of the safety analysis methods have their own advantages and disadvantages. For the complementariness, the HAZOP study is actually a bidirectional analysis method in that, at the point of a plausible deviation, the analysis is carried out backwardly from the deviation to find a cause or causes for this deviation and also it is searched to identify the consequence(s) and the effect on hazards due to this deviation. In the application of the software HAZOP to the BP/CP FBD modules

of IDiPS, searching for a consequence from a deviation was much more weighted than searching for a cause of a deviation. Hence, in this study, the software HAZOP was thought of as a forward analysis method. The SFTA is, needless to say, a backward analysis method. It begins from the top node which represents an unsafe state and searches for the causes of the top event through logical paths of a FBD module, up to its inputs and input conditions. Moreover, the SFTA is usually performed by an individual expert rather than through a meeting of analysis team members. Although a part of functional characteristics adopted the form of a code walkthrough, the analysis results of the software HAZOP in this study were brief with broad descriptions. But the SFTA could pinpoint a local defect or some logical error.

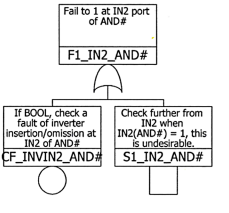
Because the software is based on the logistical constructs and its behavior is deterministic, the fault tree analysis for the software is slightly different from that for the process systems where fault trees are based on a probabilistic nature. The SFTA for a code has been constructed based on the fault tree templates [14], [15]. The fault tree templates for the function blocks (FBs) in the FBD program had been proposed in an earlier research [16], but those templates considering both the fault-oriented view and the cause-effect view were proven to be inefficient in a real implementation. In this study, the templates for the FBs were refined to be more fault-oriented in order for the templates to be implemented easily in the SFTA. The types of function blocks used for the FBD modules were divided into 5 classes: Logic Operation FB (AND/



(a) Page 1



(b) Page 2



(c) Page 3

Fig. 3. Fault tree template for the AND function block

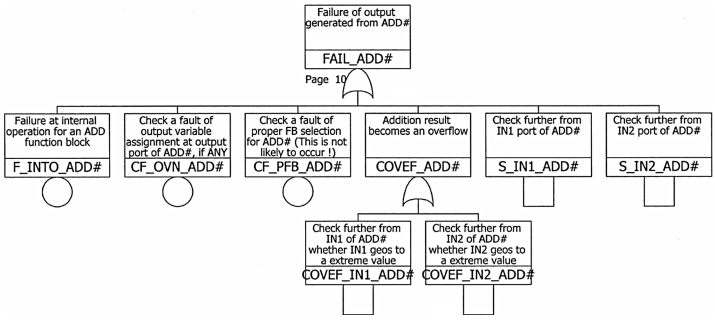
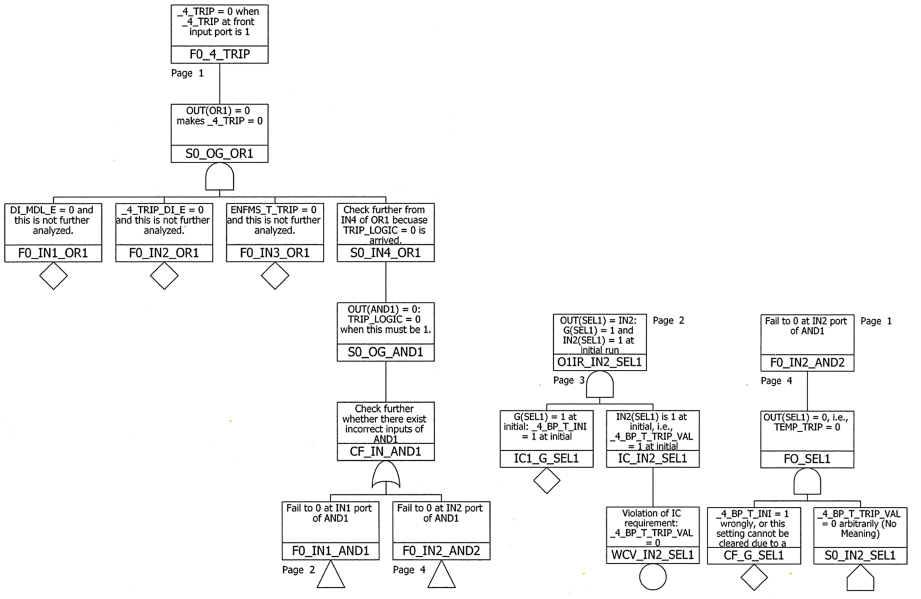


Fig. 4. Fault tree template for the ADD function block

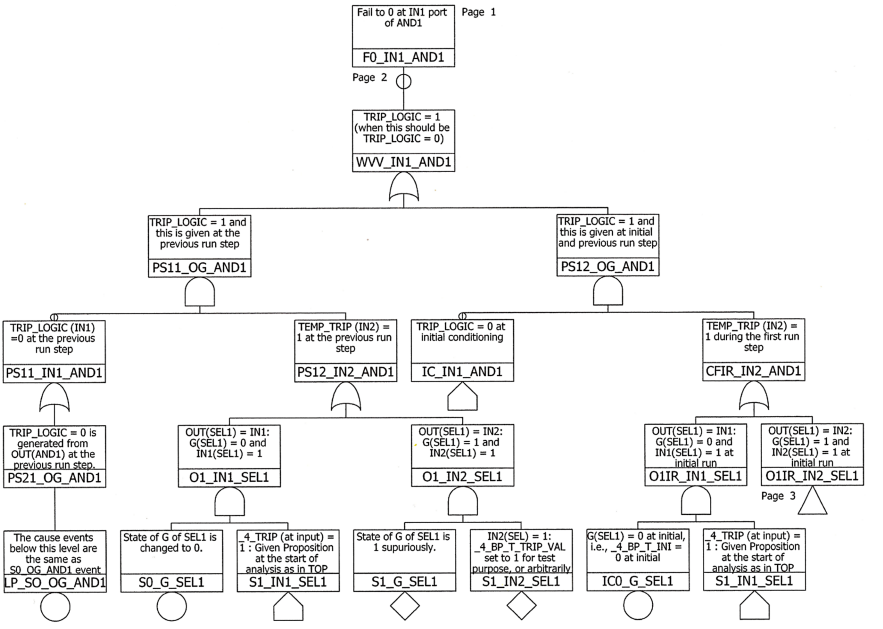
OR), Comparison FB (GE/GT/LE/LT/EQ), Selection FB (SEL), Algebraic Operation FB (ADD/SUB/MUL/DIV/ABS), and Timer FB (TON). Among the function blocks, for the limitation of this paper, the fault tree templates of the function blocks, AND and ADD are presented in Fig. 3 and Fig.4, respectively. Beside of the templates for the function blocks, the fault tree template at the final output port and also the templates for the input variables at a leaf node were devised. In Fig.3 and Fig.4, the box with a circle at the bottom of it means a basic event and one with a triangle represents an event that has more trees presented at another page. The box with a rectangle indicates that a further analysis could be progressed through this event by pasting a low-level fault tree template to this rectangle. That is, the fault tree for an FBD module is constructed by connecting the fault tree templates associated with each other through the event with a rectangle box.

The software modules selected from the results of the software HAZOP for the case of the BP FBD modules as in Table 2 are SG1_FLW_Lo Trip (Steam Generator #1 Low Coolant Flow Trip), PZR_PR_Lo Trip (Pressurizer Low Pressure Trip), VA_OVR_PWR_Hi Trip (Variable Over-Power High Trip), and DNBR_Lo Trip (Low DNBR Trip). The SFTA could systematically identify the software hazard that induced a critical system hazard. A simple example of the SFTA is shown in Fig.5 where the SFTA for the trip function of DNBR_Lo trip was pruned to leave meaningful trees. The trip logic module for a DNBR trip is depicted in Fig.6 where the function blocks related with the trip functions are “SEL1”, “AND1”, and “OR1”. The sequence of an execution is from left to right and then from top to bottom (i.e., SEL1 → AND1 → OR1 for the trip functions of a DNBR). In Fig.5, an event box with a diamond symbol represents that event is not analyzed further. An event box with a house symbol means that this event is natural.

The event in the top node of Fig.5(a) (Page 1) reflects the fact that the software cannot generate a trip signal when a trip condition is triggered and, for the case of DNBR_LO trip, it is described such that the final output value of _4_TRIP (at OR1) is 0 when an input trip variable, _4_TRIP, at the front input port (at SEL1) becomes 1. In this analysis, other trips induced from the external hardware and interface failures are precluded and thus, a fault-propagating path is progressed through an input TRIP_LOGIC at the IN4 port of OR1. This input variable is again the output variable



(a) Page 1 (c) Page 3 (d) Page 4



(b) Page 2

Fig. 5. SFTA for DNBR_LO trip

of AND1 and the SFTA is extended into the AND1 by lowering its tree level. The fault trees regarding the AND1 are depicted in the second picture (Page 2) of Fig.5.

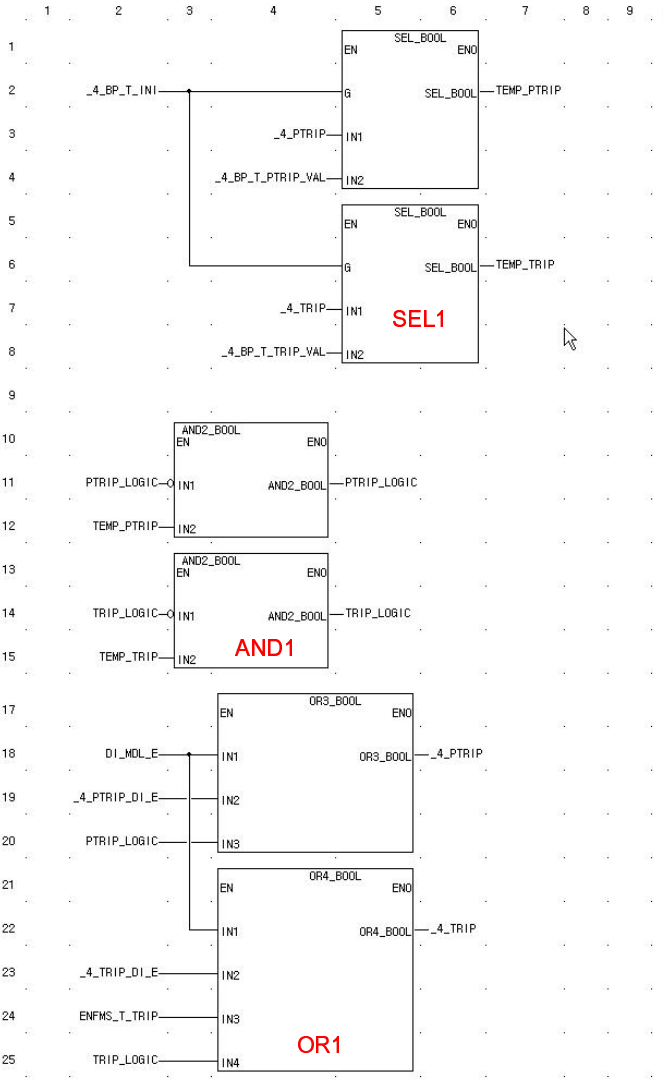


Fig. 6. FBD module of DNBR_LO trip

There are two possible failures: One is at the input variable TRIP_LOGIC and the other at the TEMP_TRIP. With the value of _4_BP_T_TRIP_VAL at SEL1 being 0, one failure mode concerning TEMP_TRIP, leading to the top event, is the wrong input selection during the real trip operation, that is, _4_BP_T_INI = 0 (but it must be 1) as in Fig.5(d). This input selection failure was not progressed further in this example because this is not the scope of the FBD module in Fig.6. For reasoning the failure modes about the TRIP_LOGIC in AND1, the tree logic is divided into two cases. One is the fault trees when an unsafe event at the top node occurs at the

execution immediately after the initial execution was finished and the other is the fault trees when the top event occurs at an arbitrary time instance. As can be seen in Fig.5(b), the event named "LP_S0_OG_AND1", where the name starting with a prefix LP_ means there is a loop within the fault trees, indicates the events occurring immediately below this event are the same as those below an upper event named "S0_OG_AND1". This means that the values of TRIP_LOGIC are toggling, which results in the coming-and-going trip signals between 0 and 1.

The logic error described above had not been detected even in the formal verification process where a file containing a FBD module was converted into the Verilog language format by the use of a conversion tool [17] developed proprietarily for the KNICS project and the model checking by a SMV mode checker was performed on the converted Verilog specification based on the specified computational temporal logic properties. Some results from the formal verification are presented in Table 5.

Table 5. Formal verification results for DNBR_LO trip

Label	Attribute	Attribute Description	Result
_4_P3	if (!TRIP_LOGIC && (_4_BP_T_INI && _4_BP_T_TRIP_VAL)) assert _4_P3: TRIP_LOGIC_1 == 1	If the current state is non-trip, the test mode is activated, and the test value is 1, then the trip output variable is 1.	TRUE
_4_P4	if (DI_MDL_E || _4_TRIP_DI_E || ENFMS_T_TRIP || TRIP_LOGIC_1) assert _4_P2: _4_TRIP_out == 1	If there is any hardware error or a logical trip, then trip signal is generated.	TRUE

In fact, the hazard identified by the SFTA of Fig.5 may be found in the code inspection or through a unit test at the implementation phase. Moreover, the software HAZOP is capable of identifying such a hazard in a small FBD module as in Fig.6. Regardless of this fact, the application of the SFTA is thought to be necessary to find a local defect. The SFTA for the other FBD modules had a tremendously complex and lengthy tree structure where the software HAZOP seemed to be impossible to draw out a logical defect. For the case of a testing, a good set of test cases or a paramount test scenario needs to be designed carefully.

5 Conclusions

For the SSA of a digital reactor protection system in the KNICS project, the strategy and methods are presented in this paper. As techniques, the software HAZOP and the SFTA are employed in the SSA for the design description represented by FBD modules (and they can also be used for the FBD program). Because of a different viewpoint from the V&V activities, the SSA could produce some valuable results that had not been identified through a previous rigorous V&V procedure including a tool-based formal verification.

Acknowledgments. This work has been performed for the research entitled "Development of the Licensing Technology for Digital I&C" as part of the KNICS project which is under the auspices of the Ministry of Commerce, Industry and Energy in Korea.

References

1. Park, J.H., Lee, D.Y., Kim, C.H.: Development of KNICS RPS Prototype. In: Proceedings of ISOFIC (International Symposium on the Future I&C for NPPs) 2005, Session 6, Tongyeong, Korea, pp. 160–161 (2005)
2. Koo, S.R., Seong, P.H., Yoo, J., Cha, S.D., Youn, C., Han, H.-C.: NuSEE: An Integrated Environment of Software Specification and V&V for PLC based Safety-Critical Systems. Nuclear Engineering and Technology 38, 259–276 (2006)
3. Kwon, K.C., Lee, J.S., Cheon, S.W.: Software Qualification Strategy for the Digital Protection Safety Systems in KNICS. In: American Nuclear Society Winter Meeting, Albuquerque, NM, USA, November 12–16, 2006, pp. 109–110 (2006)
4. IEC 61131, Part 3, International Standard for Programmable Logic Controllers: Programming Languages, International Electrotechnical Commission (1993)
5. Regulatory Guide 1.168, Verification, Validation, Reviews and Audits for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, U.S. Nuclear Regulatory Commission (2004)
6. IEEE Std-1228, Software Safety Plan (1994)
7. Lawrence, J.D.: Software Safety Hazard Analysis, UCRL-ID-122514, Lawrence Livermore National Laboratory (1995)
8. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley Inc., Reading (1995)
9. Lee, J.S., Lindner, A., Choi, J.G., Miedl, H., Kwon, K.C.: Software Safety Lifecycles and the Methods of a Programmable Electronic Safety System for a Nuclear Power Plant. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 85–98. Springer, Heidelberg (2006)
10. Redmill, F., Chudleigh, M.F., Catmur, J.R.: Principles underlying a Guideline for Applying HAZOP to Programmable Electronic Systems. Reliability Engineering and System Safety 55, 283–293 (1997)
11. McDermid, J.A., Pumfrey, D.J.: A Development of HAZARD Analysis to Aid Software Design. In: Proceedings of the 9th Annual Conference on Computer Assurance, Gaithersburg, MD, USA, pp. 17–25 (1994)
12. NUREG-0800, Standard Review Plan: BTP HICB–14, Guidance on Software Reviews for Digital Computer-Based Instrumentation and Control Systems, U.S. Nuclear Regulatory Commission (1997)
13. Lee, J.S., et al.: HAZOP Method for Safety Analysis of Software Requirements Specification (in Korean). In: Proceedings of the Korean Nuclear Society Spring Meeting, Gyeongju, Korea, May 2003, vol. 87 (2003)
14. Leveson, N.G., Shimeall, T.J.: Safety Verification of Ada Programs using Software Fault Trees. IEEE Software, 48–59 (1991)
15. Cha, S.S., Leveson, N.G., Shimeall, T.J.: Safety Verification in MURPHY using Fault Tree Analysis. In: Proceedings of 10th International Conference on Software Engineering, Singapore, April 1988, pp. 377–386 (1988)
16. Oh, Y., Yoo, J., Cha, S., Son, H.S.: Software Safety Analysis of Function Block Diagrams using Fault Trees. Reliability Engineering and System Safety 88, 215–228 (2005)
17. Jeon, S.: Verification of Function Block Diagram through Verilog Translation, M.S. Thesis, Computer System Division, EECS Department, Korea Advanced Institute of Science and Technology, Korea (2007)

Specification of a Software Common Cause Analysis Method

Rainer Faller

exida.com GmbH, Birkensteinstr. 53,
83730 Fischbachau, Germany
Rainer.Faller@exida.com

Abstract. Electronic safety systems for applications with a high level of safety integrity as in nuclear plants use hardware redundancy extensively. By implementing identical or similar software in the redundant hardware channels, systematic software failures may become a vital origin of common cause failures. The paper specifies a Software Common Cause Analysis allowing a well-documented judgment whether the likelihood of dangerous common cause failures in the conjunction of the system environment with the embedded software is adequately low, or which initiating events cannot be adequately controlled and measures on system level must be taken in order to prevent the initiating event or diversify the subsystems. The paper specifies an extensive list of common cause initiators from the environment onto software and combines them with fault avoidance and control measures in an event tree method.

Keywords: Software Common Cause Initiators, Event Tree, Fault Avoidance, Fault Control.

1 Scope and Introduction

This paper solely regards software-related common cause failures, i.e. the combination of specification- and design weaknesses, which were not detected by verification- & validation measures, with initiating events outside the software like rare transients of processed signals and data, evoked by system status, human action or hardware failure of the automation equipment. The paper does not regard hardware-related production failures.

2 Software Common Cause Analysis

Common Cause Analysis (CCA) analyses the influences of similar events on the channels of redundant systems. Common Cause Analysis (CCA) is mostly executed on system level. In order to support the system-level CCA and to determine a hardware common cause factor, IEC 61508-6 offers an evaluation table, which leads to a ß-factor estimation.

By implementing identical or similar software in the redundant hardware channels, systematic software failures may become a vital origin of common cause failures.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 162–171, 2007.

Systematic software failures are primarily avoided by quality assuring measurements in development. Not detected systematic failures, however, may, in combination with unusual external events lead to similar failures in redundant applications. Typically these are rare external events, as expected external events would already have been detected during the tests in the safety-related software development.

2.1 Definitions

Common Cause Failure (CCF) (CDV1 IEC 62430:2005)
Failure of two or more structures, systems or components due to a single specific event or cause.

Note 1 – The coincidental failure of two or more structures, systems or components is caused by any latent deficiency from design or manufacturing, from operation or maintenance errors, and which is triggered by any event induced by natural phenomenon, plant process operation or a man cased action or by any internal event in the I&C system.

Note 2 – Coincidental failure is interpreted in a way that also a sequence of system or component failures is included when the time interval between the failures is too short for repair measures.

Software Common Cause Failures are coincidental failures of several redundant software-based subsystems for which on software failure is causal in the chain of events. The triggering event itself may be a rare external incident, such as environmental conditions (EMC, lightning, radiation), hardware failures of other system outside the subsystem under evaluation, human failure, human maintenance failure.

IEC 60880-2: Software by itself does not have a CCF mode. CCF is related to system failures arising from faults in the functional requirements, system design, or in the software.

2.2 Method

The Software Common Cause Analysis proposed herein examines possible chains of events and conditions. For this purpose, triggering events are defined, which may act as common cause initiator, and the software behaviour in respect of these common cause initiators is analysed. For the common cause initiator to lead to a safety critical system behaviour, the software must meet three conditions:

I - Impact: The potential common cause initiator must influence the software in the considered operation mode.

A - Avoidance: The software has *not* been analyzed or tested in terms of the initiator during development;

M - Mastering: The software does *not* contain measures[1] which control or mitigate the negative effects of the initiator at runtime.

[1] Typical measures according to IEC 61508 are: (1) Logical program flow monitoring, (2) assertions / plausibility check and (3) data/ time redundancy.

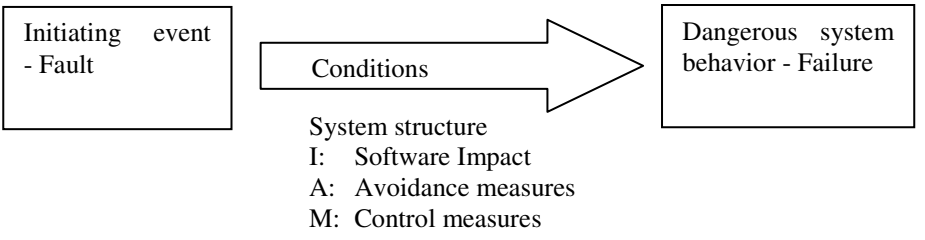


Fig. 1. Initiating event leading to a dangerous system failure

Only if these conditions are met and no measures on system level are taken (e.g. functional diversity), the initiating event will lead to a dangerous system behaviour.

The following event flow results from the conditions described:

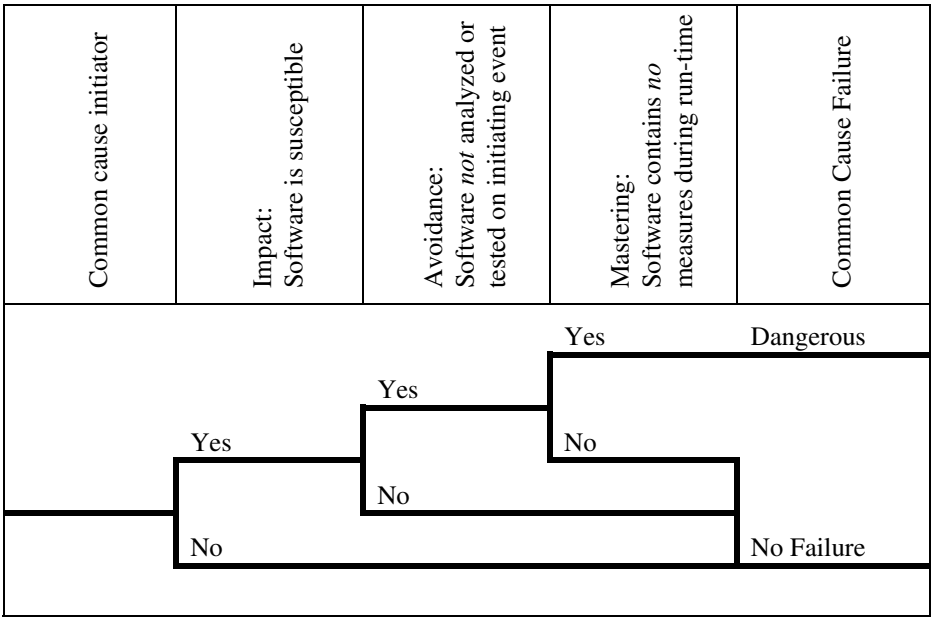


Fig. 2. Event flow from initiating event to system failure

For easier application, the event tree may also be implemented as a table.

It is important for the analysis to define the initiating events to be regarded. A proposal for a comprehensive list is shown in chapter „Initiating events to be regarded“.

If the event flow shows that the consequence associated with an initiating event must be considered as critical, measures on system level must be taken, in order to:

- Avoid the initiating events; or
- Strengthen weak system parts; or
- Diversify the weak sub-systems.

Table 1. Example: Initiating event - Too high interrupt frequency by input signal

Conditions	Description
E: Initiating event is common cause initiator	Too high interrupt frequency Interrupt is formed in both channels from different, but similar signals
I: Event is relevant for the operation mode regarded	Yes, safety-relevant software parts are interrupt-driven
A: Software not analyzed or tested on initiating event	Explicit stress tests with excessive interrupt frequencies
M: Software contains no runtime measures	Logical and temporal program flow monitoring
Consequence	No Failure

It might be useful to fill in probabilities instead of "Yes" and "No", as described in chapter "Exemplary quantification".

2.3 Independence

The independence of the conditions is essential for the event tree modelling. Therefore only few conditions (I, A, M) are defined. Further refinement of the conditions is only possible if the independence is preserved. The conditions used in this paper can be justified as follows:

- Avoidance vs. Mastering and Impact

  Avoidance is a development time issue whereas the other conditions are run time issues.

- Mastering vs. Impact

  Different parts of the system determine these conditions. The possibility for an impact of a common cause initiator is determined by the inputs to the software thru interaction with other subsystems. The mastering of the impact is determined by the run time measures implemented in the software.

2.4 Initiating Events to Be Considered

The method follows the Stress versus Strength concept, which is the basis for many common cause analysis methods. It is assumed that a component had been developed against frequently occurring assumed external stress factors. If these stress factors are stronger than assumed or if not assumed stress factors occur then redundant components exposed to this stress fail in common.

The following table of initiating events lists influencing factors (stress) which may adversely influence the Software, but are rarely examined in software reviews and tests. The table is supposed to portray an adequate checklist for a software common cause analysis. The table considers and refines the requirements from CDV1 IEC 62340 chapter 8, 9.3 and 9.6 with the exception of 8.E, as this requirement can only be analyzed for specific plant software.

In order to graduate the analysis effort, recommendations using the IEC 61508 nomenclature are given with each common cause initiator. The SIL-classification stems from the safety function(s) executed by the device / system.

HR: The common cause initiator is highly recommended to be considered for this SIL. If this common cause initiator is not considered then the rationale behind not using it should be detailed and agreed with the assessor.
R: The common cause initiator is recommended to be considered for this SIL.

Table 2. List of initiating events

Initiator **SIL**	**1**	**2**	**3**	**4**
Initiators via signals and data				
Influences from non-examined combinations of input signals	--	R	R	HR
Influences from non-examined signal form of input signals (transients, non-examined frequency spectrum)	--	R	R	HR
Exceeding of boundaries	R	HR	HR	HR
Incorrect communication data	R	HR	HR	HR
Initiator via lack of independence				
Lacking independence of the data of safety-relevant and non-safety-relevant program parts	--	HR	HR	HR
Lacking temporal independence of safety-relevant from non-safety-relevant program parts	--	HR	HR	HR
Lacking independence of the processing of different safety functions	--	R	HR	HR
Non-availability of several safety functions at failures of input signals of a single safety function	--	R	HR	HR
Illegal Pointer	--	R	HR	HR
Initiator via temporal influence				
Non-deterministic or not monitored time behaviour				
Interrupt-driven instead of cyclical processing	--	R	HR	HR
Interrupts dependent on process data	R	HR	HR	HR
Interrupt-driven scheduling without logical and temporal program flow supervision	--	R	HR	HR
Event-driven communication of safety-relevant data	--	R	HR	HR
Influences from not regarded combinations of input events / communication events / interrupts	--	R	HR	HR
Utilization of the time behaviour of Source Code constructions (e.g. program loops for time generation)	R	HR	HR	HR
Violation of worst-case time-/frequency assumptions (e.g.. sampling theorem, interrupt frequency)	--	R	R	HR
Asynchronous access to common resources	--	R	HR	HR
Calendar-dependent processing	--	R	R	HR
Initiator via human influences				
Erroneous parameterization due to complexity or ambiguity of parameters	HR	HR	HR	HR

Table 2. *(continued)*

Initiator SIL	1	2	3	4
Erroneous interference in parameterization	--	R	HR	HR
Illegal change of software or parameterization	--	R	R	HR
Initiator via influences from Coding				
Dangerous source code constructions – General e.g. substantial use of global variables, platform-dependent data types, recursion, heap und queue elements of different length.	R	HR	HR	HR
Dangerous source code constructions – C / C++ specific e.g. variable allocations in conditions, complex pointer calculations, Union, macros.	R	HR	HR	HR
Not intercepted run-time failures like overflow, divide-by-zero, illegal pointer access.	R	R	HR	HR
Changing memory load				
Dynamic objects explicitly created during run-time by the application software	R	HR	HR	HR
Dynamic objects implicitly created during run-time like Standard C/C++ library functions with implicit memory allocation, library functions for lists, queue, heap management	--	R	HR	HR

3 Exemplary Quantification

The Software Common Cause Analysis shows which initiating events, whose occurrence probability cannot be regarded as negligible, cannot be sufficiently controlled. Alternatively, the method can show that the probability of dangerous common cause failures in connection with software is sufficiently low. For the definition of „sufficiently low“, a comparison of the probability of dangerous common cause failure of a single event against the allowed Probability of Failure on Demand (PFD) per IEC 61508 for the affected safety function is proposed.

The probabilities resulting from the event flow and the effectiveness of the failure avoidance and failure control measures shall presently serve only for evaluating different software solutions and improvements (Sensitivity) within the Software Common Cause analysis.

In order to better judge event flows, the initiating events and the conditions should be associated with probabilities. The probability of the initiating event can be quantified from previous experience as explained in [9] and IEC 61508-7 Annex D.2. If no initiating event has been observed during N statistically independent runs with input distribution equal to the distribution for demands during operation, the probability of the initiating event can be estimated for the confidence level ß:

$$P_{Initiating\ Event} \leq -\frac{\ln(1-\beta)}{N} \ . \tag{1}$$

3.1 Quantification of Typical Measures of Fault Control

As there are presently no generic statements regarding software-related failure probabilities, three very conservative probability classes are proposed:

- High Effectiveness = 0,99;
  The effectiveness is high, if there is a coherent justification, that the event or its effect will be mastered for all scenarios.
- Medium Effectiveness ≥ 0,9;
  The effectiveness is medium, if there is a coherent justification, that the event or its effect will be mastered, but only under rare circumstances.
- Low Effectiveness ≥ 0,7;
  The effectiveness is low, if the measure is judge to be helpful but there is no coherent justification, that the event or its effect will be mastered.

The ranges were chosen as they seem to be analytically justifiable even without statistical evidence. Experience data should be collected at a later stage.

Table 3. Measure to master initiating events or their effect

Measure	Qualitative description of effectiveness	Effectiveness
A: Stress Testing	Can provide evidence for compliance with the safety related process time conditions and memory constraints.	Medium ≥ 0,9
M: Logical and temporal program flow monitoring	Can monitor all program states and the compliance with the safety related process time conditions.	High = 0,99
M: Time Fence	Can monitor the compliance with the safety related process time conditions.	Medium ≥ 0,9
M: Pre-assertions on data input	Monitors the input space. Effective for algorithms without history.	High = 0,99
	Less effective for algorithms with history.	Low ≥ 0,7
M: Memory Protection	Covers corruption of data due to interference from other non-safety software parts	Medium ≥ 0,9
M: Safe Data Types	Covers corruption of data by hardware failures and interference from other (non-safety) software parts.	High = 0,99

Examples for Safe Date Types are: A logical variable is stored and processed in two representations as bit and integer; a numerical variable is stored and processed as float and normalized integer or the data content of the variable is secured by a redundancy code.

Exemplary limit: The probability of the dangerous common cause failure resulting from a single initiating event shall not be greater than 1% of PFD as specified by IEC 61508.

Table 4. Example 1: Initiating event - Too high interrupt frequency by input signal

Condition	Description	Probability
E: Initiating event is common cause initiator	Too high interrupt frequency Interrupt is formed in both channels from different but similar signals Data from operating experience	$P_{Initiator} = 2*10^{-2}$
I: Event is relevant for operation mode regarded	Safety-relevant software parts are interrupt-driven	1
A: Not suitable tests	Explicit stress tests with excessive interrupt frequencies	1 – 0.95
M: Independent runtime measures	Logical and chronological program flow monitoring	1 – 0.99
Probability of a dangerous CC failure		1e-5
Exemplary limit (1%): Suitable for SIL		SIL 3

Alternatively shown as graphic event flow:

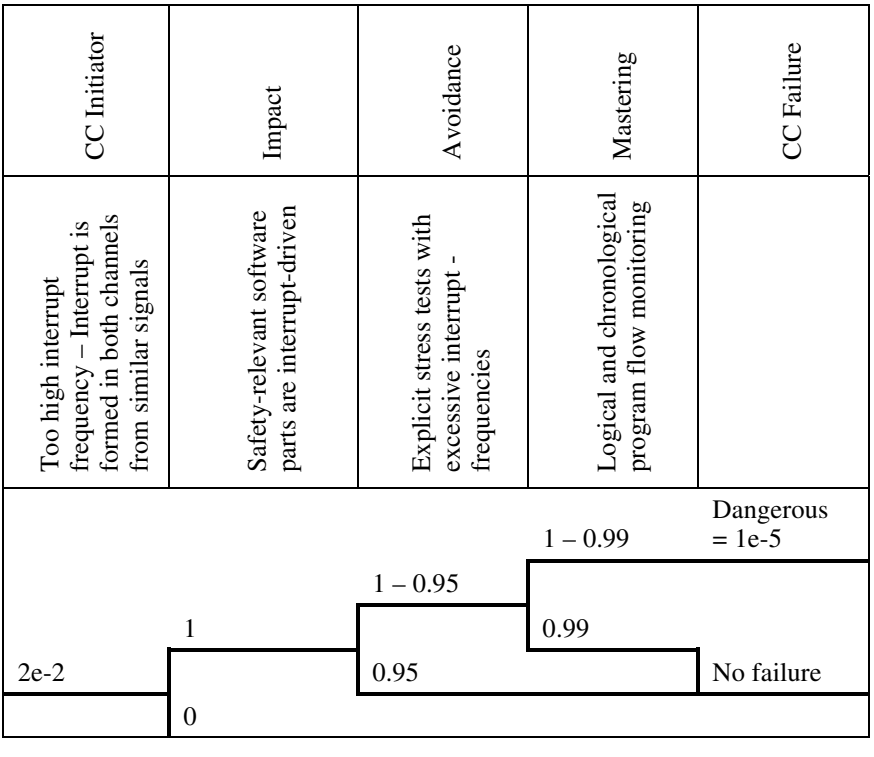


Fig. 3. Example 1: Initiating event - Too high interrupt frequency by input signal

Table 5. Example 2: Initiating event - Incorrect scheduling

Condition	Description	Probability
E: Initiating event is common cause initiator	Incorrect scheduling Common operating system Data from operating experience	$P_{Initiator}$ = 1e-3
I: Event is relevant for operation mode regarded	Safety-relevant software is controlled by OS	1
A: Not suitable tests	Scheduling tested under normal conditions, but no evidence for extreme conditions	1 – 0.7
M: Independent runtime measures	Logical and chronological program flow monitoring	1 – 0.99
Probability of a dangerous CC failure		3e-6
Exemplary limit (1%): Suitable for SIL		SIL 3

Alternatively shown as graphic event flow:

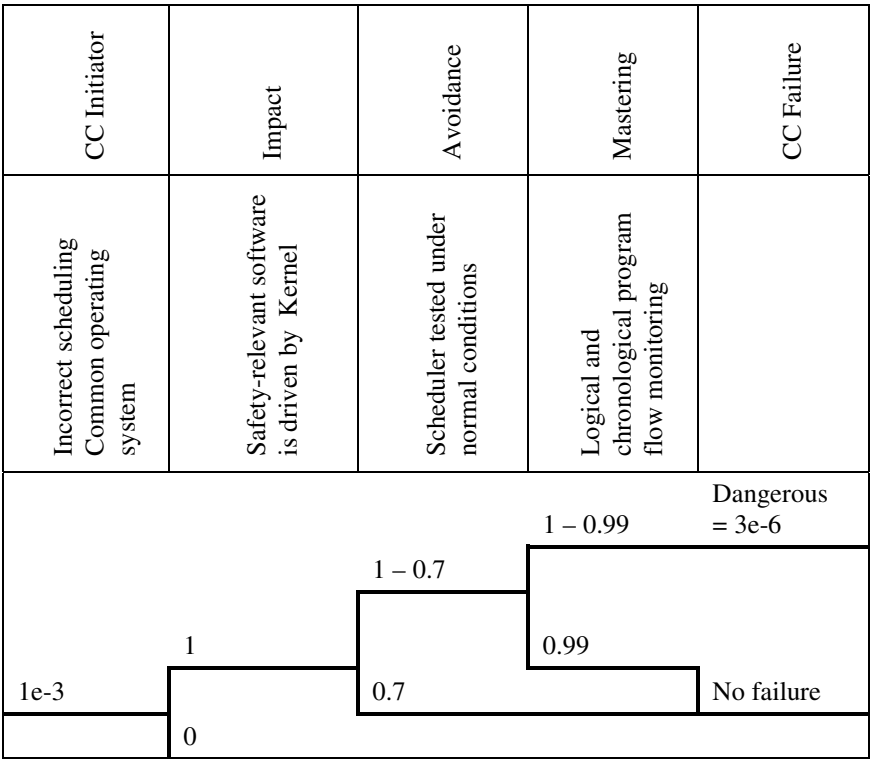


Fig. 4. Example 2: Initiating event - Incorrect scheduling

4 Outlook

The qualitative part of the method stems from the safety evaluation of the common cause potential of identical field devices or identical software parts in different field devices such as operating system kernels and libraries in nuclear power plants.

The quantitative part of the method will be tested during the application of the qualitative part. The comparison of qualitative and quantitative approach shall help to gain experience with the assumed probabilities. On a longer term, it shall justify conservative software failure quantification and widen the view on software-related common cause in respect of safety standards such as IEC 61508 and IEC 60880 which concentrate common cause considerations to the system level and development process level.

References

1. IEC 61508-2:2000: Functional safety of electrical/electronic/programmable electronic safety-related systems; Hardware
2. IEC 61508-3:1998: Functional safety of electrical/electronic/programmable electronic safety-related systems; Software
3. IEC 61508-6:2000: Functional safety of electrical/electronic/programmable electronic safety-related systems; Guidelines on the application of IEC 61508-2 and -3
4. IEC 60880:2006: Software for computers important to safety for nuclear power plants
5. CDV1 IEC 62430:2005: Nuclear power plants – Instrumentation and control systems important to safety – Common Cause Failure (CCF)
6. European Cooperation for space standardization, Draft ECSS-Q-80-03A: Methods and techniques to support the assessment of software dependability and safety
7. Chillarege, R., Bassin, K.A.: Software Triggers as a function of time (1995)
8. Chillarege, R., Biyani Sh., Rosenthal J.: Orthogonal Defect Classification. In: Measurement of Failure Rate in Widely Distributed Software (1999)
9. Ehrenberger W.: Software Verifikation – Verfahren für den Zuverlässigkeitsnachweis von Software, Hanser-Verlag (2004)
10. Friedman M. A., Voas J. M.: Software Assessment (1995)
11. Musa, L.M., et al.: Handbook of Software Reliability Engineering (1996)
12. Sullivan, M., Chillarege, R.: Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems (1991)

Combining Bayesian Belief Networks and the Goal Structuring Notation to Support Architectural Reasoning About Safety

Weihang Wu and Tim Kelly

Department of Computer Science, The University of York, York YO10 5DD
{Weihang.Wu,Tim.Kelly}@cs.york.ac.uk

Abstract. There have been an increasing number of applications of Bayesian Belief Network (BBN) for predicting safety properties in an attempt to handle the obstacles of uncertainty and complexity present in modern software development. Yet there is little practical guidance on justifying the use of BBN models for the purpose of safety. In this paper, we propose a compositional and semi-automated approach to reasoning about safety properties of architectures. This approach consists of compositional failure analysis through applying the object-oriented BBN framework. We also show that producing sound safety arguments for BBN-based deviation analysis results can help understand the implications of analysis results and identify new safety problems. The feasibility of the proposed approach is demonstrated by means of a case study.

1 Introduction

Designing dependable software systems requires early prediction of critical system properties so that effective architectural feedback can be generated as part of the evolutionary design process. The elaboration of the linkage between architectural design and safety poses challenging research problems of reasoning – reasoning about safety properties of an architecture given current level of design detail and items of evidence available in the development process. By formalising and articulating the reasoning behind architectural design, we believe that there can be an increased level of confidence in architecting safety-critical software applications. Experience has shown that most design methods predominantly rely on implicit human reasoning and judgements to inform design decisions.

In this paper we propose an approach to compositional failure analysis of system and software architectures through the application of the object-oriented Bayesian Belief Network (OOBBN) framework and articulating analysis results through the Goal Structuring Notation (GSN). We argue that BBN/OOBBN models can provide a cost-effective medium for architects to analyse critical system properties on the basis of limited evidence available, only if the underlying assumptions and implications of the BBN models developed are fully understood and properly consolidated. We therefore distinguish two essential steps of architectural reasoning from the perspective of safety: compositional failure analysis with the aid of OOBBN tools, and comprehensible safety argumentation based upon the explanation of BBN-based

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 172–186, 2007.

failure analysis results. We illustrate the proposed framework by means of an aircraft wheel brake system (WBS) controller example extracted from ARP 4761 [1].

The remainder of the paper is organised in the following five sections. Section 2 reviews related work. Section 3 describes compositional failure analysis through the application of the OOBBN framework. Section 4 describes the safety argumentation required to support the analysis in the form of goal structures. Section 5 presents the WBS example as an illustration of the approach. Finally, Section 6 draws some conclusions and discusses future work.

2 Related Work

The notion of BBN was developed in the AI community to facilitate automated reasoning about real-world problems under uncertainty. A BBN represents a directed acyclic graph (DAG) together with associated conditional probability distributions based upon explicit conditional independence assumptions, thereby saving space for probabilistic computation [17]. In practice, BBN models are often interpreted as causal models [16], in which the directed edges represent knowledge about causal relations. Modular development of large-scale BBN models has also been explored in a form of objects [18]. Several BBN tools such as Hugin [2] are also available to facilitate inductive and deductive reasoning in an automated manner. BBN models have already been applied to solving software engineering problems. Fenton et al [3] developed a number of generic BBN patterns to support software safety case development and risk assessment. Sutcliffe et al proposed a method of constructing generic BBN models to evaluate usability [8] and later developed an automated tool to evaluate reliability and performance through different configurations of BBN models [9]. However, little work exists on leveraging BBN-based reasoning power for the purpose of safety analysis.

Whilst BBN-based reasoning can replace human reasoning by combining subjective estimates and statistical data theoretically, it has been generally accepted that producing an accurate BBN model can be very difficult. One possible solution is to create an effective dialogue between human reasoning and automated reasoning in order to communicate the domain knowledge and explain the reasoning behind the BBN models [5]. There are two classes of the explanation methods in the literature [13]: explanation of the construction of BBN models, and explanation of the BBN inference process on the basis of evidence obtained. The research in automated explanation is still in its early stages, however. We believe that existing argumentation approaches such as GSN may be used to offer such a dialogue.

The benefits of compositional reasoning have been increasingly recognised in the software community. At York, Fenelon et al [6] proposed a compositional failure modelling prototype – Failure Propagation and Transformation Notation (FPTN). Within the FPTN framework, failure behaviours of an architectural component can be classified in terms of failure propagation and generation behaviours specified by Boolean logic. Failure behaviours of a composite system can thus be inferred by its underlying structure as captured by its architecture. The authors have examined the application of Communicating Sequential Processes (CSP) as an implementation of FPTN [19]. Although CSP can capture uncertainty in both a qualitative and

quantitative manner [15], as a behaviour modelling language it is lacking expressive power for reasoning about safety risk. In this paper we have attempted to integrate the OOBBN framework with compositional failure modelling.

3 Compositional Failure Analysis

A specification describes the desired behaviours of an architectural component situated in a specific environment [11]. The behaviours are inherently causal, as the provided service of that component is dependent upon its required service as stipulated in the specification. In practice, the behaviours can be effectively interpreted in terms of stimulus and response [4]. Obviously, a stimulus can be treated as the cause of its designated response. The stimulus can be generated by either the environment or the component itself. The stimulus-response formulation can be applied recursively, where the response of a component becomes the stimulus of connected components in the architecture. As a result, system properties can be inferred in a compositional manner.

Yet the stimulus-response specification is only desired and perceived by the stakeholders, as the regularity between stimulus and response may not be true due to the potential existence of defects embedded in either that component or its situated environment. Deviation arises when the causal link between a stimulus and response is disrupted or intervened (using Pearl's term [16]). The problem then lies in the robustness of the causation as specified under all circumstances. Put another way, the main concern will be the exhaustiveness and credibility of the potential disruptions of the epistemic causation, as well as their possible safety consequences once they do occur, since deterministic causation may not be realistic. Deviations can be generalised in terms of failure modes. Previous work at York has developed a comprehensive set of failure modes for software systems as part of the SHARD (Software Hazard Analysis and Resolution in Design) method [14]. We interpret the SHARD failure modes with respect to the stimulus-response specification as follows:

- *Omission*. Response part does not hold while the stimulus and environment parts hold.
- *Commission*. Stimulus or environment part does not hold while the response part holds.
- *Timing*. Timing constraint specified in the response part is violated while the other parts hold.
- *Value*. Accuracy constraint specified in the response part is violated while the other parts hold.

If we translate the specification of an architectural component into pairs of stimulus and response, we will have a DAG model (i.e. a BBN structure)

$$C = (V, E)$$

where V is defined as a set of Boolean variables, each corresponding to a distinct element of all identified stimuli and responses, and E is defined as a set of directed edges among V, each corresponding to a distinct element of all identified causal relations between stimuli and responses. As a result, each architectural component can

be mapped onto an individual BBN model. Architectural components of the same component type (i.e. sharing the same specification) will be mapped onto the same BBN model. Formally, the mapping can be defined as a total, surjective function where the domain is defined as the number of architectural components and range is defined as a set of BBN models. Following the OOBBN terminology, the stimulus and response nodes in *V* can be represented by input and output nodes, respectively. The corresponding BBN models are known as OOBBN models and their instances (representing architectural components) can be composed together within a composite OOBBN model, in which these OOBBN instances are represented as instance nodes (or subnets) and connected by linking the output nodes of an instance node to the input nodes of another, as specified in an architectural view.

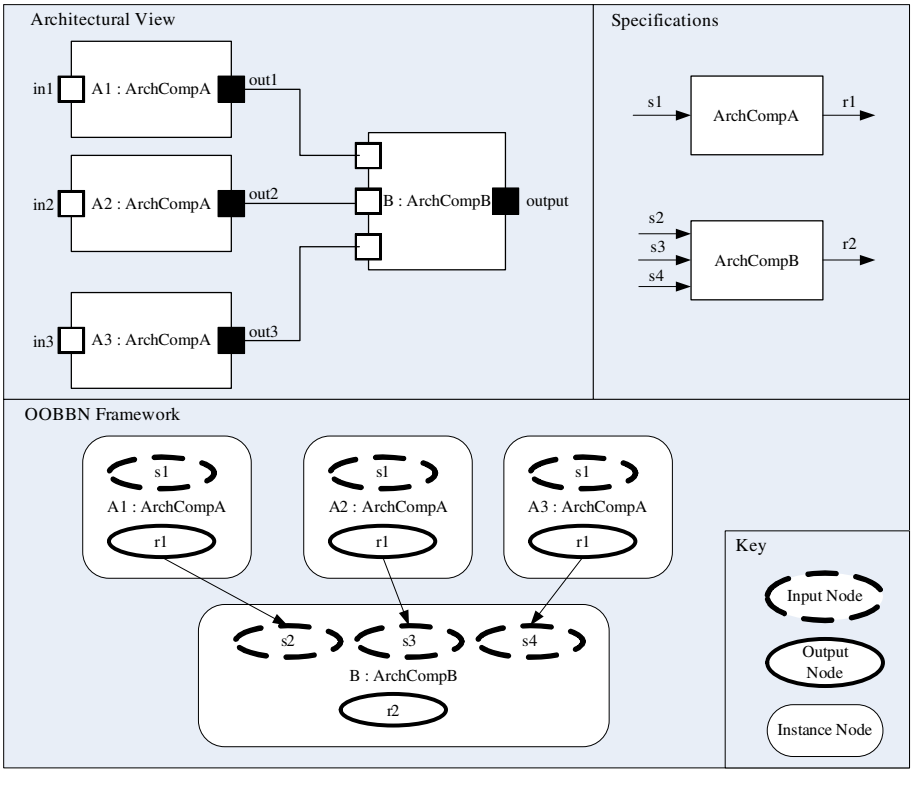


Fig. 1. A schematic model of compositional failure modeling

Figure 1 illustrates a schematic model of a compositional failure modelling process. We assume that a run-time view of an architecture is available, which introduces three parallel-running architectural components of the same type and an architectural component responsible for arbitrating the outputs of the three. Since there are two types of architectural components (i.e. two architectural specifications) in the run-time view, there will be two corresponding OOBBN models, in which the

input and output nodes are specified according to their specifications. Finally, we will have a composite OOBBN model that represents the composition of the four architectural components, as described in the run-time view.

The aim of compositional failure modelling is to identify the possible component deviations and reason about their safety implications at the system level. It is equally important to reason about the credibility of each deviation identified, as some deviations are certainly more likely than others. From the perspective of evolutionary design, credibility estimation can also help the architect to prioritise safety concerns in terms of risks, since addressing all safety concerns in a one-off manner is simply unrealistic. To estimate the credibility of a deviation (e.g., the credibility of omission failure P(*Not-Response* | *Stimulus*)), we need to distinguish the types of architectural components: hardware (e.g., processor or communication link), software, human (e.g., operator), or environmental components (e.g., weather and runway conditions). For a hardware or environmental component, historical data collected will often be adequate to justify the existence of a specific deviation. For example, a sensor can fail due to electromechanical wearout, moisture intrusion, or vibration. For software or human agents, the architect should either identify the relevant causal factors given his/her knowledge and judgement, or choose appropriate BBN patterns to add them into the BBN subnet relating to the architectural component. Figure 2 illustrates a simple BBN model to estimate the credibility of omission failure of *ArchCompA*.

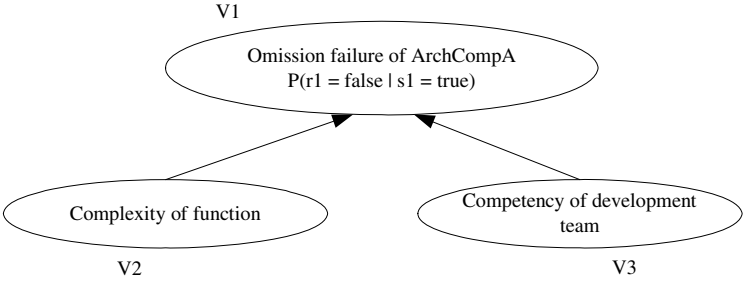


Fig. 2. A simple BBN model for estimating credibility of omission failure of ArchCompA

As shown in Figure 2, we simply assume that the credibility of omission failure is subject to the product-related factor (i.e. complexity of the behaviour of that architectural component) and the process-related factor (i.e. competency of the allocated development team). The resultant BBN model thus consists of three nodes: *V1* (a Boolean variable), *V2* (a discrete node with three states) and *V3* (a discrete node with three states). The derivation of the conditional probabilities such as P(*V1*=true | *V2* = medium, *V3* = low) within this example BBN model can be based upon the subjective judgement or statistical functions (e.g., the Beta function [3]), whilst the prior probabilities are simply undefined. Once the architect has inputted the relevant process and product findings (e.g., *V2* = high, *V3* = medium), the credibility of omission output of the component type *ArchCompA*, for example, can be obtained automatically through the BBN inference engine. Similarly, we can estimate the credibility of omission failure of the architectural component type *ArchCompB*

through using the same BBN model with different findings entered (e.g., *V2*= low, *V3* = medium). For different failure behaviours of the same architectural component, it is possible to adopt different BBN models to estimate their credibility.

Consequently, for each OOBBN model representing an elementary architectural component, we should be able to derive all the conditional probabilities for the output nodes through credibility estimation, whilst leaving the conditional probabilities of the input nodes undefined. Having defined the conditional probabilities of elementary OOBBN models, we can compile the composite OOBBN model by means of an OOBBN tool and conduct inductive reasoning (i.e. 'what if' analysis) by setting specific values of input nodes and determine whether the credibility of system failures (i.e., omission of system outputs when system inputs arrive) is acceptable with respect to specific risk acceptance criteria. Obviously, further mitigation mechanisms will be required if system failures are unacceptably credible.

The procedure of BBN-based credibility estimation is inevitably subjective: given the same deviation, different architects may have different BBN models in terms of the DAG models (i.e., a set of causal factors) and conditional probabilities (i.e. the relative strength of these casual factors). In order to produce an accurate BBN model one solution is to exploit machine learning [10] over the past project data. The effectiveness of learning would however be subject to the availability of past projects with sufficient commonality to the current project. Another solution is to exploit human reasoning capabilities to review a BBN model produced. This is labour-intensive, however. We therefore propose to use GSN to help explain and justify BBN models, as discussed in the next section.

4 Safety Argumentation for Architectures

Once the BBN-based reasoning produces a "safe enough" prediction (i.e., the credibility of all safety-related system failures is acceptably low), an architect should produce preliminary safety arguments to justify the BBN models produced and the underlying inferences. The purpose of safety argumentation at architectural level is twofold:

- *Consolidation*. By seeking safety arguments behind the BBN-based safety analysis, an architect can obtain better insight into the BBN models in terms of a model's assumptions, context and implications.
- *Communication*. By presenting safety arguments behind BBN-based reasoning and communicating them to safety assessors (who may have little knowledge about BBN), there can be an improvement on the overall agreement with respect to safety acceptability prior to system construction.

At York we have developed GSN to provide an effective medium for structuring and communicating safety arguments in terms of goal structures. Figure 3 shows the principal symbols of GSN. An away goal is a goal that is not defined within the module where it is presented but instead in another module. Within GSN, safety arguments can be generalised in terms of meta-arguments, from which domain-specific arguments can be directly instantiated. These meta-arguments can be captured in a form of GSN patterns using the multiplicity, optionality and entity

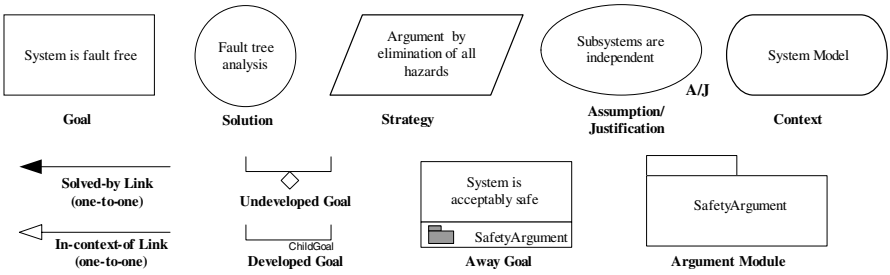


Fig. 3. The principal symbols of GSN

abstraction extensions [12]. The remainder of this section describes a four-step procedure of producing preliminary safety arguments to justify BBN-based failure analysis and architectural decisions made.

Step 1: Identify the context and boundaries of architectural safety arguments
Failure analysis is only part of safety assessment, and architecture itself cannot guarantee the achievement of system safety. It is therefore crucial that an architect be able to distinguish safety arguments derived from architectural considerations and non-architectural ones, and identify their relationships to the top-level system safety objectives. In GSN, this can be achieved by defining argument modules, declaring their interface and linking the declared modules through GSN relationships. Obviously, only argument modules related to architectural parts will be further developed by the architect.

Step 2: Create the primary safety argument
A primary argument consists of explaining the safety properties of an architecture with respect to the overall safety objectives specified, and how the desired properties can be enforced by the proposed system structures and constraints. We distinguish two classes of safety properties at an architectural level:

- *Normal behaviours of an architectural component*. The normal behaviours of an architectural component include both functional and non-functional requirements such as timing constraints. Apparently, a key safety concern here is if these requirements have been met satisfactorily through the proposed architecture. Equally important for safety is to ensure that all the safety-related normal behaviours are correct with respect to what are intended from the safety viewpoint. For example, a deadline requirement of an aero-engine controller may be implicitly derived from historical values related to similar products. The relevancy of this timing requirement to this particular project must be validated through extensive engine simulation and trials. For another, functional requirements may be derived from the application of safety-related design decisions (e.g., failure detection). Correctness arguments for these derived requirements must be established.
- *Failure behaviours of an architectural component*. Deviation analysis concerns the possible ways of violating the normal behaviours of an architectural component and their contributions to the known system hazards. Establishing an argument of

exhaustive identification of failure behaviours and corresponding mitigation alternatives is recognised as the key to the robustness of dependability design. The creation of the exhaustiveness arguments should be straightforward. For example, the exhaustive identification of failure behaviours of an architectural component can be relied upon the use of predefined failure/deviation modes of that component. We assume two basic forms of failure behaviours within boundaries of a composite system: failure propagation and failure generation. Component failures can only be propagated through dependencies between architectural components identified in an architectural view. Under the feasibility of failure propagation is the assumed claim that architectural components are independent given that there exists no connection between them in an architectural view. This independence assumption may be conditional however. For example, two parallel-running software tasks in a run-time view may share the same processor in the deployment view. It is thus important for an architect to establish the independence argument. The soundness of such an argument needs to be further validated through thorough review of all related architectural documents by a safety assessor.

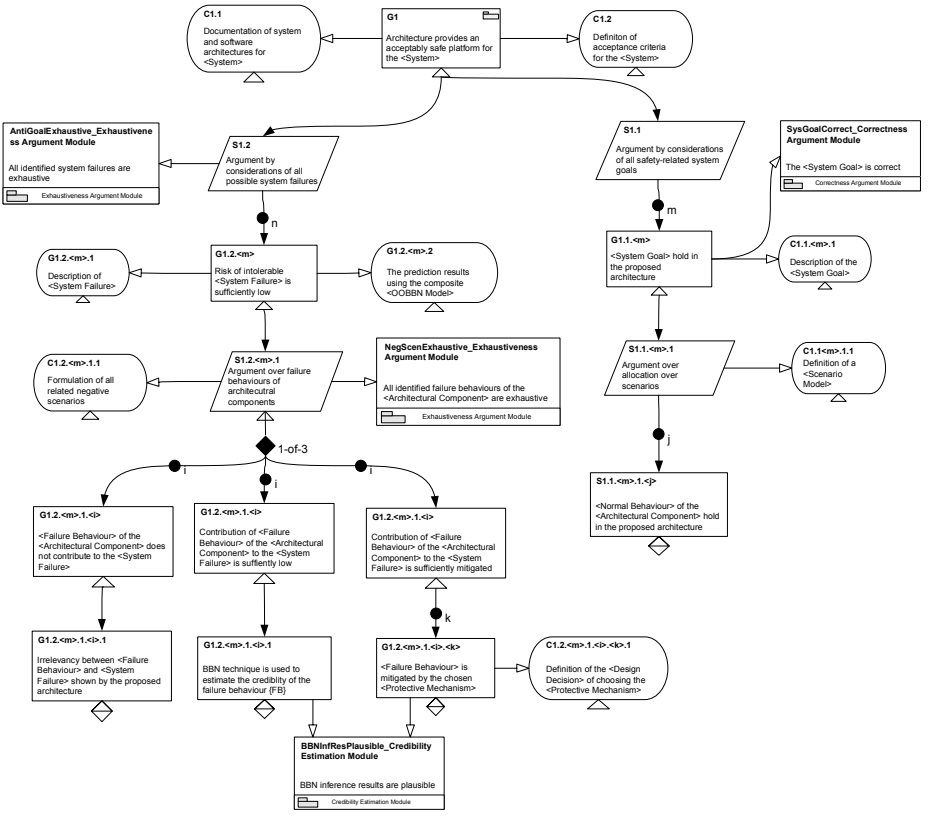


Fig. 4. Meta-argument for the primary argument

Figure 4 shows the meta-argument for top-level primary safety argumentation. The argumentation process starts by supporting the top-level claim of acceptable safety through defining the acceptability criteria for a system's architecture. The meta-argument is then divided into two parts: a qualitative and quantitative part. The quantitative part argues that the risk of failure behaviours of the architecture is acceptably low. The qualitative part addresses how the normal behaviours of the architecture meet the necessary safety-related goals. Three backing arguments are also referred as the contextual parts within the primary argument structure to address two particular concerns regarding the refinement: the correctness of the identified system goals, and exhaustiveness of the identified system failures, and plausibility of the BBN inference results. We will discuss the development of backing arguments in the next step.

Step 3: Create the backing arguments

Backing arguments offer rationale behind the primary argument in terms of the following aspects: exhaustiveness, independence, correctness, and credibility estimation. The creation of exhaustiveness, independence and correctness arguments should be straightforward. Here we will focus upon the creation of credibility estimation arguments that justify the results generated by compiled BBN models. A compiled BBN model consists of the following three elements:

- Causal structure. A causal structure encodes information about the causal factors involved and the explicit assumption of conditional independence. For example, an architect simply assumes that complexity of the software function, quality of requirements specification and quality of development process are the dominant factors contributing failures of a software component.
- Conditional probabilities. The conditional probability table (CPT) for a child node expresses the relative strength of causal influence towards that node. The CPTs for parentless nodes are deliberately undefined, as we will enter findings once the BBN model is compiled. For the same BBN model example in Figure 2, the more complex the software function is, the more likely it will fail; the better-quality the development process is, the less likely the software component will fail. Between the two factors, the complexity may be assumed to be the most influential factor; the quality of the development process comes second. The definition of the conditional probabilities should be consistent with this implicit ranking assumption and should be validated through evidence weighing or sensitivity analysis [13].
- Findings entered. A finding on a BBN node means one of the states of that node is observed to be true. In our application of BBN, findings are entered for parentless nodes only. The findings are not necessarily the evidence in the real world. For the previous example, we may simply claim that the quality of requirements specification will be good by assuming a rigorous the review procedure. Whether the claim is satisfied depends upon the result of requirements specification review that will be available in the subsequent development stages.

Justifying the BBN prediction results thus involves the steps of justifying the definition of the causal structure, conditional probabilities of child nodes, and findings entered for the parentless nodes. Figure 5 shows the meta-argument for credibility estimation using BBN models. The meta-argument consists of three parts in order to

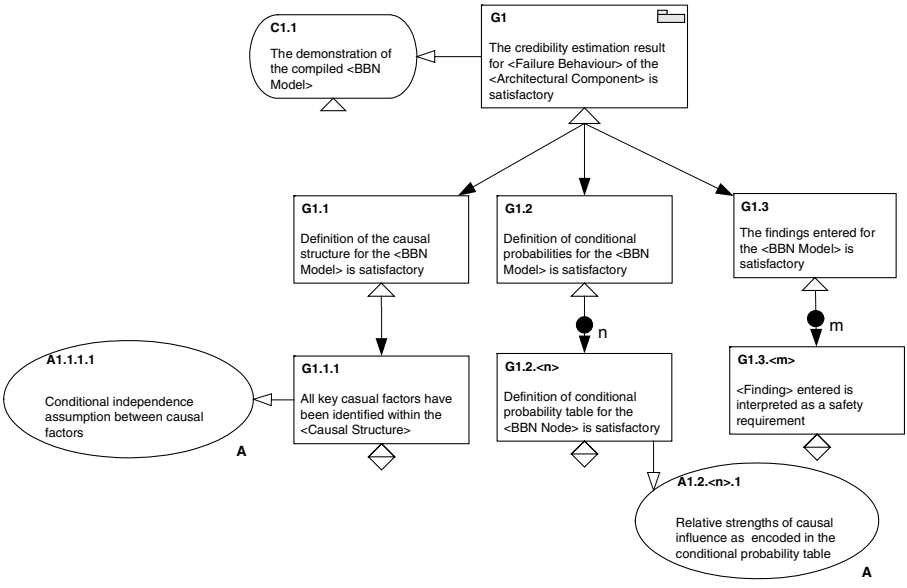


Fig. 5. Meta-argument for backing argument of credibility estimation

support the claims of plausibility of the definition of causal structure and conditional probabilities, and the inputs of findings, respectively. The three parts of argument are refined through examining the corresponding elements within the BBN model (such as items of causal factors, CPTs, and findings entered). The assumptions of conditional independence and relative causal strengths must be explicitly specified, as shown in Figure 5.

Step 4: Review the architectural safety arguments and generate feedback

The architectural safety arguments developed must be reviewed by an independent review team. The aim of the review is to verify the safety arguments developed and identify new safety problems as input for the subsequent architectural design stages. The reviewers should include both safety assessors and related system stakeholders. The review procedure starts by presenting the primary safety argument and explaining the safety goals/concerns identified, safety-related architectural strategies chosen, and relevant context behind the primary argument. By now, the reviewers will have a good idea of current progress of architecting. They will also study the supporting documents (e.g., architectural views) that the argument refers to. The reviewers then start to ask probing questions regarding to the validity of the goal structure presented. Issues identified will be fed back to the subsequent design process.

Once the primary argument is agreed, the reviewer will examine all backing arguments. Review of arguments for correctness, exhaustiveness and independence should be straightforward. Review of argument for credibility should be accompanied by the demonstration using a BBN tool (e.g., walking through the BBN nodes, conditional probability tables and inference results). Explanation of the context of the BBN model produced is also required. It must be stressed that the main use of BBN

models is to reason about the probability of systematic failures in the early stages of system development so that negative scenarios can be prioritised. Therefore, the reviewers should compare the BBN prediction results with their subjective judgement. If any inconsistencies found, further clarification of BBN models is required.

The output of the review procedure is a set of architectural feedback in a form of safety issues and derived requirements, as well as the underdeveloped goals in the argument structures, which will be addressed in the next iterations.

5 Example

Our example concerns the design of WBS in which a software system (residing in the Brake System Control Unit – BSCU) is required to generate braking commands according to the braking request from a pilot (via pedal) and the current status of the WBS. The system consists of two hydraulic supplies: the GREEN and BLUE hydraulic supplies, which are selected automatically through the selector valve. A system architecture of WBS is illustrated in [1]. We assume that the specifications of each system component have been identified in a stimulus-response form. We then map each system component into individual OOBBN model in terms of input and output nodes. Finally, we compose instances of these OOBBN models into a composite one to represent the whole system architecture. Figure 6 illustrates the composite OOBBN model captured by Hugin tool.

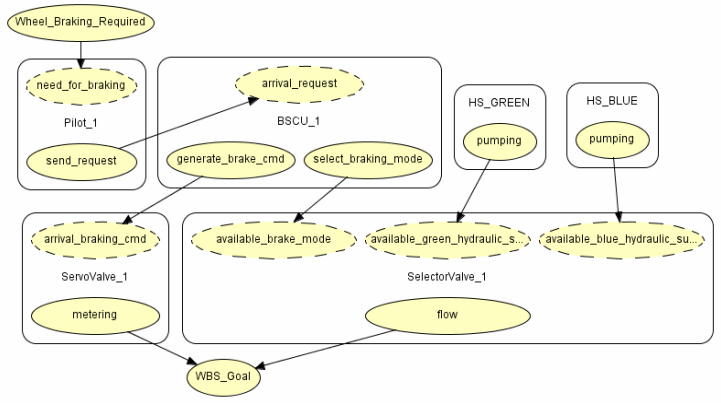


Fig. 6. A composite OOBBN model formed on the basis of WBS system architecture

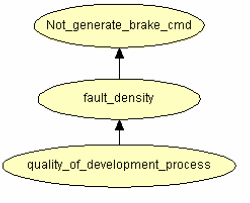
As shown in Figure 6, specifications of some architectural components such as hydraulic supply or power system do not have the stimulus part. On the other hand, the specification of BSCU can have multiple response parts. Two ordinary BBN nodes (i.e. Boolean variables) are also added in the OOBBN model to aid the architectural reasoning process: the WBS goal and system context (i.e., when wheel braking feature is required during specific aircraft flight phases). An obvious system hazard is the case when the context is true but the WBS goal is false (i.e. loss of

wheel braking). An alternative system hazard is when the WBS goal holds within the wrong context (i.e. inadvertent wheel braking). The conditional probabilities between the WBS goal and subnets (e.g., ServoValve_1) are defined as a Boolean function, whilst the prior probability of the system context variable is deliberatively undefined, as the architect can freely perform 'what if' analysis by setting different values.

In order to predict the credibility of both system hazards identified, we need to define the credibility of failure behaviours of each system component in terms of conditional probabilities. For hardware components such as ServoValve_1, we simply assume they can only generate omission failures due to the underlying random faults with the credibility P (*Not-Response* | *Stimulus*) = 1E-5 and SelectorValve_1 and ServoValve_1 can also propagate any failure from BSCU_1 with certainty P(*Not-Response* | *Not-Stimulus*) = 1. For software/human components, we need to identify the relevant causal factors available in the current state of system development. For simplicity, we here assume that Pilot_1 is fault-free by claiming that both omission and commission of braking request sent by pilot are impossible (i.e., P(*send_request* = true | *need_for_braking* = true) = 1). For BSCU_1, we may assume that no product-related metrics are available at this level, and thus we adopt process-based prediction. For the purpose of demonstration, we use the simplest process metric available (i.e. quality of development process) to predict the number of latent software faults during system operations. Figure 7 shows the BBN model derived from empirical data relating to CMM levels [7] in order to estimate the credibility of omission failures of BSCU software such as P(*generate_braking_cmd* = false | *arrival_request* = true).

Not_generate_brake_cmd

fault_density	0 - 0.5	0.5 - 1	1 - 2	2 - 4	4 - 6	6 - 10	10 - 20
false	1	1	0.9995	0.995	0.95	0.9	0.4
true	1.0E-6	1.0E-5	0.0005	0.005	0.05	0.1	0.6



fault_density

quality_of_...	1	2	3	4	5
0 - 0.5	0	0	0	0.1	0.75
0.5 - 1	0	0	0.1	0.75	0.2
1 - 2	0	0.1	0.75	0.1	0.05
2 - 4	0.1	0.75	0.1	0.05	0
4 - 6	0.75	0.1	0.05	0	0
6 - 10	0.1	0.05	0	0	0
10 - 20	0.05	0	0	0	0

Fig. 7. The BBN model created for estimating credibility of omission failure of BSCU software

If we decide to decompose the BSCU into two individual channels on the basis of the use of Command/Monitor pattern [20] (i.e., a COM channel responsible for braking calculation, and a MON channel responsible for detecting faults from COM channel given knowledge about the reasonableness of the braking output), the subnet BSCU_1 is thus decomposed into two subnets: COM_1 and MON_1. The internal structure of COM_1 is the same as the initial subnet BSCU_1 except that the response

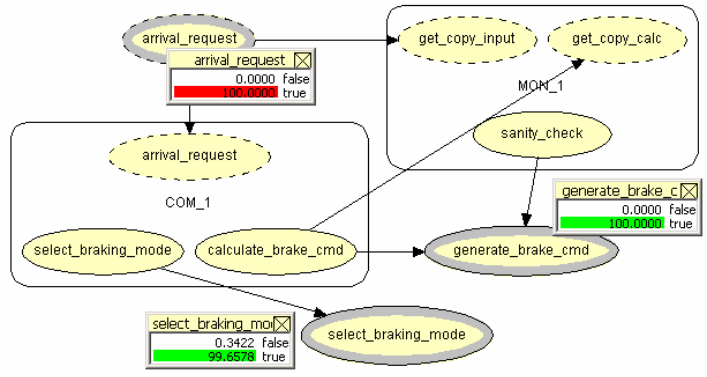


Fig. 8. The compiled OOBBN model for BSCU designed with Command/Monitor pattern

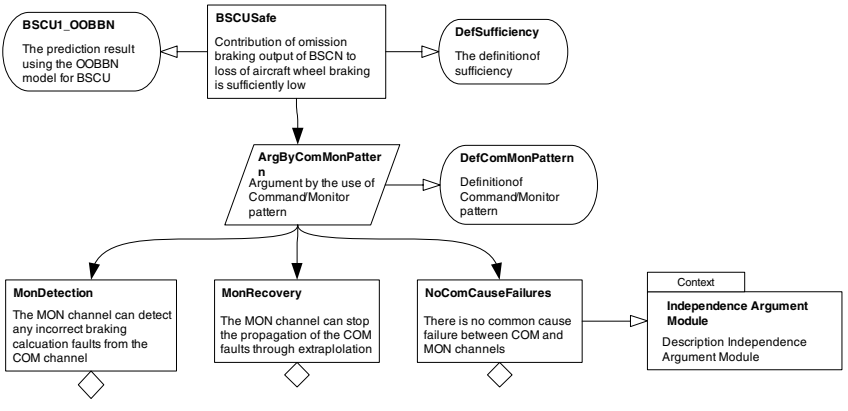


Fig. 9. Part of the primary safety argument for the WBS example

part is changed to become the calculation of braking commands. The MON_1 subnet is defined by assigning the Boolean function between the response part sanity check and the stimuli parts from inputs and COM output. Figure 8 shows the compiled BBN model for the refined subnet BSCU_1 in order to predict the credibility of omission output of BSCU. The probability of generating braking commands has changed to 100% due to the additional safety check by the MON channel, whilst the reliability of selecting the braking mode is still unchanged (i.e., 99.97%). If the prediction results are acceptable, we then create the preliminary safety argument which can be directly instantiated from the predefined meta-argument. The primary argument simply replays the progress of the WBS system design: i.e., how the WBS system goal is achieved in the proposed architecture and the two system hazards have been addressed sufficiently. Figure 9 shows part of the primary argument, which shows that failure behaviours of BSCU have been sufficiently mitigated.

Having reviewed these arguments by the safety assessor, it becomes obvious that some value failures of the COM channel are subtle and thus undetectable. Therefore

the zero credibility of omission braking output predicted by OOBBN models is not realistic. Consequently, we may need to refine the COM channel in order to address the undetectable failures. For example, we may improve the reliability of the COM channel to ensure that the credibility of undetectable COM failures is sufficiently low.

6 Conclusions and Future Work

We have presented a novel approach to architectural reasoning about safety in a compositional and semi-automated manner. The approach is based upon BBN-based deviation analysis and goal-based argumentation to justify the deviation analysis results. Although it may be attractive to realise the ultimate goal of automated reasoning about safety in the future, human reasoning is still the vital part of architectural reasoning. The proposed method for safety argumentation is aimed to improve human reasoning by means of clearly-structured dialogues, thereby offering complementary means of architectural reasoning. The proposed approach relies heavily upon the development of BBN models derived from architectural knowledge about the system domain and current status of the development process. We expect that an architectural reasoning language that sits between existing architectural description languages and OOBBN inference engine will thus be desired. Our future work includes the precise definition of an architectural reasoning language, and integrating BBN inference engine and automatic generation of GSN-based safety arguments into existing architectural design environments.

References

1. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, Society of Automotive Engineers, Inc. (1996)
2. Hugin Researcher package. Hugin Expert (2007)
3. The SERENE Method Manual SafEty and Risk Evaluation using bayesian NEts: SERENE, ERA Technology Ltd. (1999)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison Wesley, Reading, MA, USA (2003)
5. Druzdzel, M.J.: Qualitative Verbal Explanations in Bayesian Belief Networks. Artificial Intelligence and Simulation of Behaviour Quaterly 94 (Special Issue on Bayesian Belief Networks), 43–54
6. Fenelon, P., McDermid, J., Nicholson, M., Pumfrey, D.: Towards Integrated Safety Analysis and Design. ACM Computing Reviews 2(1), 21–32
7. Fenton, N.E., Neil, M.A.: Critique of Software Defect Prediction Models. IEEE Trans. on Software Engineering 25(5), 675–689
8. Galliers, J., Sutcliffe, A., Minocha, S.: An impact analysis method for safety-critical user interface design. ACM Transactions on Computer-Human Interaction (TOCHI) 6(4), 341–369
9. Gregoriades, A., Sutcliffe, A.: Scenario-Based Assessment of Nonfunctional Requirements. IEEE Trans. on Software Engineering 31(5), 392–409
10. Heckerman, D.: A Tutorial on Learning with Bayesian Networks. Microsoft Researcgh (1995)

11. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: Proceedings of the 17th international conference on Software engineering (ICSE'95), Seattle, US, pp. 15–24. ACM Press, New York (1995)
12. Kelly, T.P., A Arguing Safety Systematic Approach to Safety Case Management D.Phil Thesis, Department of Computer Science, University of York, York (1999)
13. Lacave, C., Diez, F.J.: A review of explanation methods for Bayesian networks. The Knowledge Engineering Review 17, 107–127
14. McDermid, J.A., Nicholson, M., Pumfrey, D., Fenelon, P.: Experience with the application of HAZOP to computer-based systems. In: Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95), pp. 37–48 (1995)
15. Morgan, C.: Of Probabilistic Wp and SP-and Compositionality. In: Symposium on the Occasion of 25 Years of CSP, London, pp. 220–241. Springer, Heidelberg (2004)
16. Pearl, J.: Causality: models, reasoning, and inference. Cambridge University Press, Cambridge, U.K., New York (2000)
17. Pearl, J.: Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann Publishers, San Mateo, Calif. (1988)
18. Pfeffer, A., Koller, D.: Object-Oriented Bayesian Networks. In: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97), pp. 302–313. Morgan Kaufmann, San Francisco (1997)
19. Wu, W., Kelly, T.: Failure Modelling in Software Architecture Design for Safety. SIGSOFT Softw. Eng. Notes 30(4), 1–7
20. Wu, W., Kelly, T.: Safety Tactics for Software Architecture Design. In: Proceedings of the 28th International Computer Software and Applications Conference, IEEE Computer Society, Los Alamitos (2004)

Application of Interactive Cause and Effect Diagrams to Safety-Related PES in Industrial Automation

Hans Russo and Andreas Turk

Infoteam Software GmbH
Hans.Russo@infoteam.de

Safety requirements have a high impact on current industrial applications. Companies are liable by law for injuries to health and environmental hazards. Today international standards exist to prove for hazard avoidance. A decisive part of safe industrial applications is the software running a Programmable Electronic System. Programmable systems cannot be certified in general, so a time-consuming certification process has to be re-initiated during each commissioning. Hence, there is a strong need for easy-to-use tools, which not only simplify the application development, but do also support the certification process by modelling and presenting the system's behaviour in an easily accessible way. We present a methodically diverse approach combining both, safety-related and standard requirements, within a single application. We apply the documentation technique of "Cause & Effect Diagrams" to a software tool. This allows developing efficiently safety-related applications up to Safety SIL 3 [2].

1 Introduction

OpenPCS is the only IEC 61131-3 workbench certified for "Portability Level" by PLCopen. Infoteam Software GmbH integrated a new concept called OpenPCS/SIL. SIL means the Safety Integrity Level [2]. OpenPCS/SIL is designed for automation engineers enabling them to address within one engineering tool both the safety and the standard requirements placed on an automation solution.

In the field of automation safety-related aspects are always directly related to a standard application to be created. The safety aspect of OpenPCS/SIL is based on the proven in use concept of Cause & Effect Charts and refines this method to a level, which allows developing safety applications effectively and efficiently up to SIL3 in combination with standard functionality. With that, OpenPCS/SIL reaches a high level of competitiveness due to its efficiency within the safety-related development process. Another advantage of the close collaboration between the safety and the standard part of an application by using Cause & Effect is to save time and money during safety specific activities, verification and validation. Cause & Effect Charts are a well known method in process control and related industries for documenting the wiring of plants and facilities. So the idea was born and realized to use Cause & Effect as a method of developing safety-related applications.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 187–196, 2007.

The originality of the approach to use the Cause & Effect to develop a safety-related application is similar to the approach of use of object-oriented programming languages like Small Talk or C++ to develop OO-Software. Surely it is possible to develop object-oriented software with C or Assembler but it is not forced to do it right. The analogy says: Cause & Effect forces the development of a safety-related application in comparison to other programming languages e.g. FBD. The evidence for that is that the Cause & Effect programming approach mirrors many requirements of the standard IEC 61508 [2]. The IEC 61508 standard sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic components (electrical/electronic/programmable electronic systems (E/E/PESs)) that are used to perform safety functions. Some main requirements that are touched from the Cause & Effect approach are found in the chapters 7.4.2, 7.4.3, 7.4.4 IEC 61508-3:2001 [3].

This will be demonstrated in chapter 7.4.2.4 [3] and chapter 7.4.4.3 [3]

The standard requires in chapter 7.4.2.4 [3]:

"The design method chosen shall possess features that facilitate software modification. Such features include modularity, information hiding and encapsulation"

This Requirement is realized by the use of Cause & Effect due to the finding the design of the application this means finding the causes and find out what the reaction has to be is equal to the implementing phase. Furthermore it is planed to generate the documentation automatically from these informations.

The standard requires chapter 7.4.4.3 [3]:

"To the extent required by the safety integrity level, the programming language selected shall:

a) have a translator/compiler which has either a certificate of validation to a recognized national or international standard, or it shall be assessed to establish its fitness for purpose;
b) be completely and unambiguously defined or restricted to unambiguously defined features;
c) match the characteristics of the application;
d) contain features that facilitate the detection of programming mistakes; and
e) support features that match the design method."

This Requirement is realized by the use of Cause & Effect due to:

a) It is planed to fully certify OpenPCS/SIL.
b) The restrictions are strong and clear.
c) It is the application itself
d) Automatically error detection
e) It is the design itself

2 Cause and Effect Methodology

The Cause & Effect methodology consists of the Cause & Effect matrix and input and output values. The Cause & Effect matrix contains four quadrants. Quadrant 1 has no

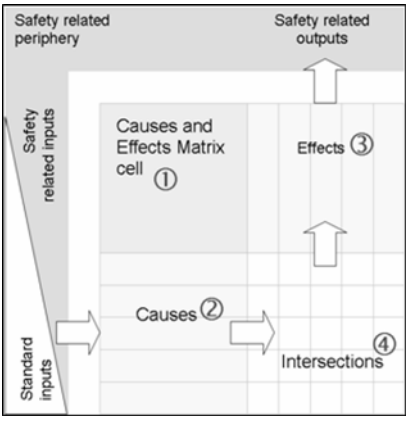


Fig. 1. Cause & Effect Methodology

functionality; it is used to hold some general information. Quadrant 2 lists the causes from top to down. A cause is an effect producer, e.g. to push a stop button should stop the device. So pushing the stop button is the cause. Stopping the device is the effect.

Quadrant 3 lists the effects from left to right. The quadrant 4 lists all logical interconnections called intersections between Causes and Effects.

In our case all Causes are represented through an electrical input Signal. The special requirements to that signal is that it has to be a SAFEBOOL type which is BOOL with additional information like PL [2], SIL [2], PDF [2], PFH [2]. It is planned to use that additional information to calculate the Safety Integrity of the System that is taken into account from the Cause & Effect Matrix.

3 OpenPCS/SIL Component Overview

OpenPCS/SIL consists of three components:

1. C&E Editor
2. C&E Viewer
3. C&E Runtime System (RTS)

The component C&E Editor enables the automation engineer to develop the application. The C&E Editor will be explained later in more detailed.

The C&E Viewer is a control view that controls automatically that there is no falsification during compilation. The Viewer shows the compiled and de-compiled input. The user is also able to view his inputs after compilation and de-compilation.

The safety relevant criterion is that the de-compiled view of the source code (C&E Matrix) has to be identical to the editor view, where the user enters his code.

The C&E Safety Runtime System runs on the controller as a separate task. There are some interactions between standard and safety runtime system, but nevertheless the systems are independent from each other as far as possible.

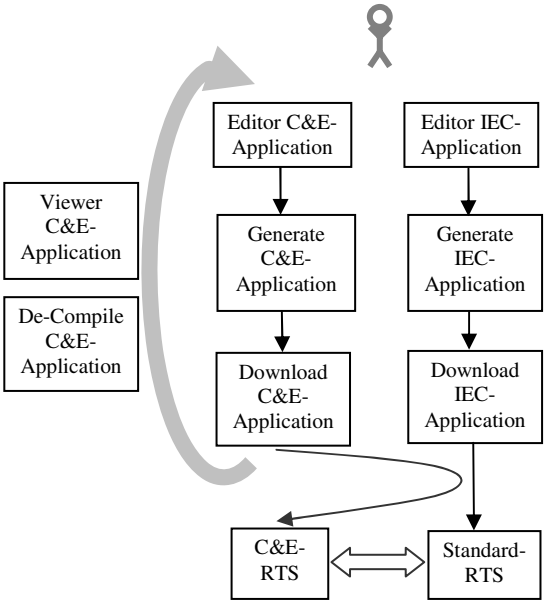


Fig. 2. Component Overview

As well as the standard RTS the C&E RTS is processed cyclically. At the beginning of each cycle the safety-related input signals of the periphery are read into the safety-related process image. During the cycle the input variables are translated into output variables by the Cause & Effect matrix. At the end of each cycle the safety-related process image is written into the periphery.

4 Cause and Effect Matrix in OpenPCS/SIL

The Cause & Effect (C&E) matrix contains all causes and effects and their interconnections with each other. The C&E matrix is build similar to the C&E matrix of C&E methodology.

4.1 Quadrant I

Quadrant is the matrix cell, which is used to display certain attributes of the matrix such as location within the name of the application, version, date of creation, date of modification, date of certification, Safety Integrity Level, project tree, dimension, etc.

4.2 Quadrant II

In quadrant II all causes are listed from top to down. The alignment of a cause line visualizes the entering of the safety-related input signal into the matrix on the left

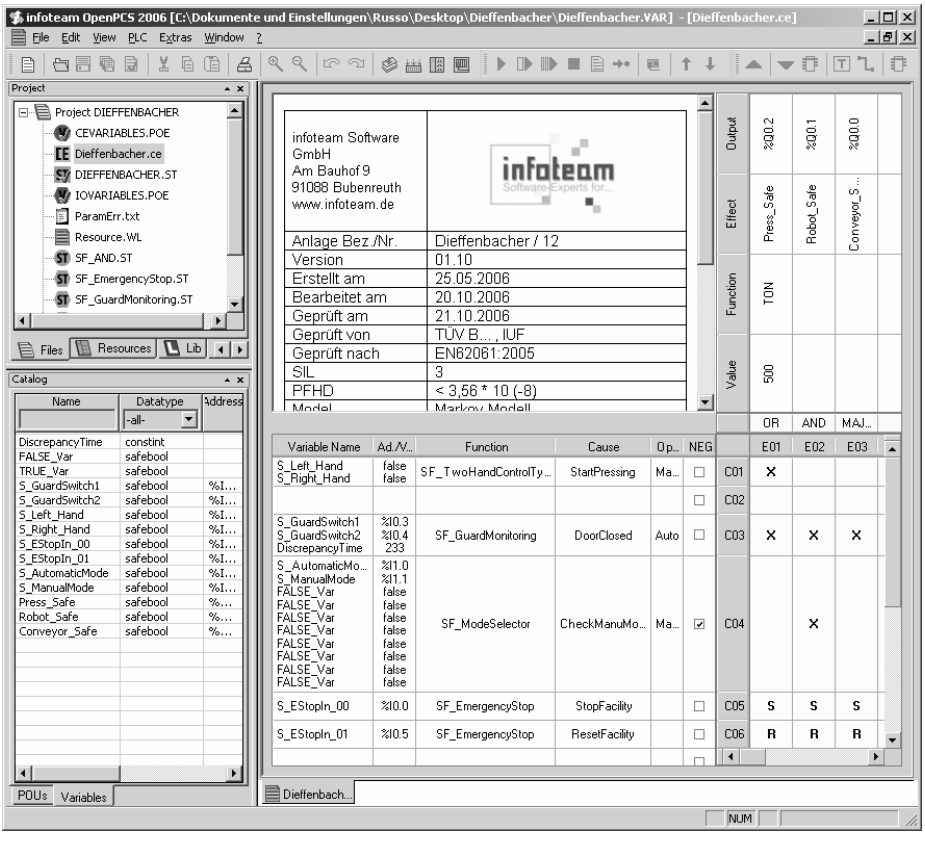


Fig. 3. C&E Editor integrated in OpenPCS

side. A cause line is defined through the name and the safety function of a cause. The safety function determines how many input variables have to be assigned and how many output variables have to be written: e.g. an emergency stop has one input variable; a SF_TwoHandControl_TypIII has two input variables [4], both have one output. The input signals are processed according to the safety function. The safety function is selected through a combo box in the cell.

If pre-certified safety functions are used, the matrix allows creating a safety application which is easily certifiable. The integration of a pre-certified function in OpenPCS/SIL is organized via a template library. It is planned to offer a template network library within OpenPCS/SIL, that integrates at least all safety function blocks defined in the Technical Specification of PLCOpen [4]; see bellow for some safety function blocks.

4.3 Quadrant III

In quadrant III all effects are listed from left to right. An effect line is one row of the grid. This alignment of the effects within the matrix visualizes the exit of the

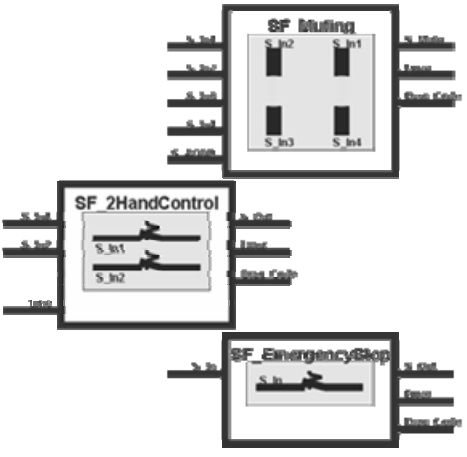


Fig. 4. Three examples from Safety Software Technical Specification [4]. All variables that are owner of a prefix S_ are SAFEBOOL variables.

safety-related signals from the top of the matrix into the periphery. An effect is defined by its output variable. An effect is always the answer to the question "Is something safe". The input of an effect line is the result value of all causes, which are interconnected with that effect line. The semantic how the interconnections are merged together is explained in next chapter "Quadrant IV". The effect simply maps its input to its output variable according to a TON or TOF function block that can be used to adapt the time behavior of the effect. An effect provides a special attribute to configure the default mapping of an assigned cause AND, OR, MAJORITY.

4.4 Quadrant IV

The intersections are the heart of the implementation of any Cause & Effect application. Therefore most of the programming rules apply to the intersections' quadrant IV.

Quadrant IV contains all interconnections between the causes and the effects. It is possible to map one output of a cause to each effect that has to respect that cause. This is a "1 to n" relationship. On the other hand it is possible to map every cause to one effect line. All causes in one intersection column below an effect line are connected to that single effect. The results of all causes have to be merged according to the semantic rules to one single SAFEBOOL value which is the input for the effect.

Causes can be connected to an effect by way of

Assignment (X) manifold (more than one assignment per column)**.** In case there is more than one cause connected to an effect by an assignment, the effect's attribute interconnection determines the exact computation of the assignments. Also one cause can be connected with many effects by an assignment (more than one assignment per row). At the top of an intersection column there is a specifier that determines the rules how the different intersections assignments in a column (one assignment belongs to one Cause) are combined with each other. There are three different specifiers: AND,

OR and MAJORITY. AND means that the signal result is TRUE, if all interconnected causes are TRUE. OR means that the signal result is TRUE, if at least one interconnected cause is TRUE. MAJORITY means that the combined calculation result is determined from the majority of causes. If the Signal from two causes of three causes is TRUE the combination is also TRUE. If the Signal from two causes of four causes is TRUE the combination is FALSE, due to there is no MAJORITY.

Set (S) manifold (more than one set per column)**.** In case there is more than one cause connected to the effect by a set, the corresponding causes are interconnected like an AND (see above). Also one cause can be connected with many effects by a set (more than one set per row). Set intersections have priority to assignments and reset intersections. The set intersections recognize a falling edge of the cause's output value defined in this row and pulls the effect's input values of this column down until a reset is recognized for this effect.

Reset (R) only once (only one reset per column)**.** Only one cause can be interconnected with many effects by a reset (more than one reset per row). Reset intersections have priority to assignments but not to set intersections. The reset intersections switch the computation of the column's effect back to the assigned causes on a rising edge of the corresponding cause unless other Set intersections hold.

4.5 Sophisticated Error Detection

During the whole development process there is a compilation and de-compilation loop working in background. If there are detected syntactical or compilation errors, these errors are reported immediately to the editor component. This leads to a changed text font from black normal to a red bold italic text at the object that encapsulates the error. Two examples of errors are: an assignment on an intersection column without a defined effect/cause column/row or a cause is made with a safety function, which input variables are not connected to a variable.

5 Example

5.1 Application Requirements

A safety-related application within a hazardous area for providing a molding press with material shall be realized. The hazardous area can be entered through a safety door. There are two options for the provision process.

Manual mode: a worker lifts the material manually into the press and triggers pressing. In this case the press must operate only if a two-hand-control is operated properly.

Automatic mode: the material provision cell is equipped with a handling robot. The robot lifts the material into the press, controlled by a process control system. The same system triggers the press. In this case the press must operate only if the safety door is closed and the control system requests pressing.

For both modes there has to be an emergency stop available. Furthermore there exists an operation mode selector and a reset button.

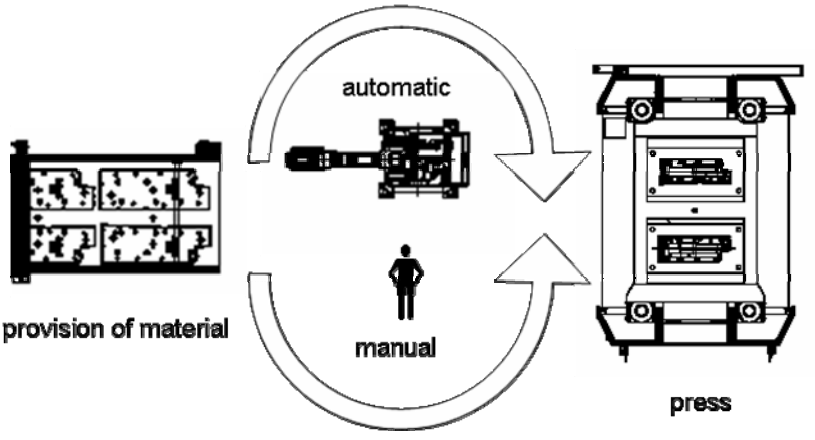


Fig. 5. Molding Press: The hazardous area is surrounded with a fence which has a safety door to access the area

5.2 Safety Application Developing

First we have to find all effects. We are able to find the effects with the following questions:

1) Is the conveyor safe?
2) Is the robot safe?
3) Is the press safe?

For each effect we need an output variable of type SAFEBOOL which represents the answer to the questions above. Each variable defines an effect.

Next we need to known what safety devices are available:

1) Emergency Stop
2) Two hand control for manual operation mode
3) Safety Door for automatic operation mode
4) Operation mode selector

For each safety device we have a pre-certified safety function block in our template library. Now, the input variables to represent the input devices have to be defined.

Now we are able to create the program and documentation of the safety application:

1) Put in all effects by drag and drop the variable into a row of quadrant III.
2) Put in all causes by naming the cause and select the safety function block and drag and drop the input variable from the variable window of OpenPCS (see variable window at the left bottom of Fig:3) onto the corresponding variable of the safety function block and select the operation mode.
3) Interconnect all causes with the effects that have to observe the causes.
    a. The two hand control has to be considered for the press only in manual mode

b. The Safety Door is has to be considered for all devices in automatic mode
c. The mode selector can be used to answer the question if the robot is safe in manual operation mode. In manual operation mode the robot is always unsafe.
d. The Stop Facility button has to stop all devices. So all effects are connected to that cause with a set intersection.
e. The Reset Facility is the reset button for all devices

After 30 minutes the safety application is up and running and documented. The result is shown in figure 3.

Due to the pre-certified "safety function blocks" and the pre-certified OpenPCS/SIL programming component the certification process is less time-consuming then the standard programming technique for a safety application. Additionally it is planed to use the safety matrix for calculation of safety relevant values like SIL [2], PFH [2]… which are an essential part of the certification process.

6 Benefits

Implementing safety-related applications with Cause & Effect closes the gap between hazard analysis, implementation and documentation in the process of safety-related software development.

Implementation of safety-related applications with function block diagram based programming languages, required manual verification and validation steps of analysis to implementation and again of implementation to documentation. The Cause & Effect implementation supersedes these manual steps because analysis, implementation and documentation are done with the same method based on the same notation.

Hence OpenPCS/SIL reduces verification and validation effort for safety-related software significantly compared to software written with tools based on conventional programming languages. This saves cost of the software related certification process.

Furthermore Cause & Effect is realized as a subsystem of standard OpenPCS, but not as a standalone tool. With that, safety and standard part of an application are well covered by one tool throughout all phases of development. Hence OpenPCS/SIL meets the requirements to address both safety and standard aspects of today's automation solutions.

Moreover, programming with Cause & Effect is simple and straightforward. This approach does not only make Cause & Effect predestined for the safety sector, but it allows efficient tool support during software development. With that, the overall engineering time is reduced. Many possible coding errors are avoided by the simplicity of the Cause & Effect methodology or recognized by the error detection. For the Safety development of safety-related applications OpenPCS/SIL reduces complexity and enforces best practices.

Additionally the safety concept of OpenPCS/SIL is TÜV approved complying with IEC 61508.

References

1. EN 61131-3:1993 Speicherprogrammierbare Steuerungen
2. IEC 61508 Functional safety of electrical, electronic, programmable electronic safety-related systems, IEC 61508. IEC, Geneva (1999)
3. Feltens P., Marko L.: A New Diversity Approach to Safety-Related PES 7th International Symposium Mai 4-5, 2006 Cologne – Germany, TÜV Nord (2006)
4. Safety Software Technical Specification Part 1 Concepts and Functions Blocks Version 1.0 – Official Release, PLCopen – Technical Committee 5
5. 61508-3:2001 Functional safety of electrical, electronic, programmable electronic safety-related systems, Part 3 IEC 61508. IEC, Geneva

Survival by Deception

Martin Gilje Jaatun[1], Åsmund Ahlmann Nyre[1], and Jan Tore Sørensen[2]

[1] SINTEF ICT, NO-7465 Trondheim, Norway
Martin.G.Jaatun@sintef.no
http://www.sintef.com/ses
[2] Norwegian University of Science and Technology, NO-7491 Trondheim, Norway

Abstract. A system with a high degree of availability and survivability can be created via service duplication on disparate server platforms, where a compromise via a previously unknown attack is detected by a voting mechanism. However, shutting down the compromised component will inform the attacker that the subversion attempt was unsuccessful, and might lead her to explore other avenues of attack. This paper presents a better solution by transforming the compromised component to a state of honeypot; removing it from duty, while providing the attacker with bogus data. This provides the administrator of the target system with extra time to implement adequate security measures while the attacker is busy "exploiting" the honeypot. As long as the majority of components remain uncompromised, the system continues to deliver service to legitimate users.

1 Introduction

The history of the Internet shows that it is not possible to develop a system that is both *impervious to attack* and *useful* (i.e., provides anything more than rudimentary functionality) – no matter how carefully crafted the armor may be, the vandals[1] always seem to be able to find a chink in it.

Attacks on e-commerce installations and general web sites frequently employ platform-specific exploits based on known vulnerabilities. In later years, the "patch window" has been steadily decreasing, to the point where we now face "zero-day exploits" that are being wielded even before a patch for the specific vulnerability is generally available. Clearly, new mechanisms are required to combat this threat. Our contribution to the cause is a system for *Increasing Survivability by dynamic deployment of Honeypots* (ISH). In the following, we will discuss the theoretical backgound and describe our prototype ISH implementation.

2 Background

Protagonists of honeypots have by many been considered the lunatic fringe of the computer security community, but Lance Spitzner [2] and the Honeynet Project

[1] We agree with Marcus J. Ranum [1] that this may be a more descriptive term for what is usually referred to as a "hacker".

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 197–208, 2007.

[3] have contributed to get more mainstream attention (if not general acceptance) for honeypot ideas. Recent publications such as [4] and [5] at recognized conferences, and others listed on the Honeynet homepage [6] lends academic credibility to the honeypot as an information security resource.

To quote [6], the *raison d'etre* for the honeynet project (and thus, a honeypot) is

> *To learn the tools, tactics and motives involved in computer and network attacks, and share the lessons learned.*

Indeed, the idea of "peering over the shoulder" of an active hacker has been pursued by many, and classic papers such as [7] and [8] describe how pioneering defenders in an ad-hoc fashion have thrown together what in reality were the first (after-the-fact) honeypot systems[2].

The idea of dynamically transforming a compromised system to a state of honeypot was introduced in [9], but the authors did not describe in detail how this might be accomplished. Disparity and redundancy are classic tenets of dependability [10] and (by extension) survivability. We have employed the definition of survivability given in [11], but with added emphasis on malicious activity rather than accidental incidents.

2.1 Related Work

- *SITAR* [12] is an architecture that aims to provide a system invulnerable to attack, using replication, software diversity and a voting mechanism.
- *Bait and Switch Honeypot* [13] is an open source project that is in many ways similar to our own. The main difference is that Bait and Switch uses a firewall proxy to direct malicious traffic to a (permanent) honeypot server, and relies solely on an Intrusion Detection System (Snort [14]) to differentiate legitimate from malicious activity.
- *Shadow Honeypots* [15] also employ a proxy-like mechanism to classify traffic, routing suspicious traffic to a special shadow server that makes a final decision.
- *MPITS* [16] is a relatively simple system that employs disparity and redundancy to offer a basis for intrusion tolerance. We have employed MPITS as an important component in the prototype implementation of ISH; MPITS is described further in section 2.2. Note that ISH depends on service replication on disparate software platforms, but not directly on MPTIS.

2.2 Minimal Proxy for Intrusion Tolerant Systems

MPITS was developed by Broen [16] to provide a less complex basis for intrusion tolerant systems, that additionally would serve as a reference system when comparing existing, more complex systems. Although existing intrusion tolerance

[2] That these early efforts did not develop further, can partly be ascribed to the fact that this kind of activity quickly proved too time-consuming – a point we will return to later.

systems have a relatively high level of complexity, they are still vulnerable to *single point of failure*. Acknowledging that the connection point to the external network always will be a single point of failure, MPITS seeks to minimize the likelihood of compromising the unit by limiting the functionality and complexity of the system. The low complexity yields better understanding and enables a more thorough inspection of the source code to eliminate vulnerabilities.

MPITS utilizes replication of services on disparate software platforms to achieve survivability. The system consists of two types of components; a number of application servers and a proxy server (see figure 1). The application servers are the servers containing the actual service the system is providing, while the proxy server works as the connection point to the outside world and manages all inbound and outbound traffic.

The proxy will forward all incoming requests to the application servers, and process the responses. In theory, all well-formed requests should generate the same resonse if the various application servers are functionally equivalent. In practice, there may be minor differences, which is why MPITS groups replies in *equivalence classes*, based on a configurable notion of what is "close enough". To determine whether two responses belong to the same equivalence class, the responses are compared byte for byte, and all discrepancies counted. If the error ratio is below a configurable threshould, the responses are considered equivalent. Special characters may be weighted to indicate their increased or decreased relative importance, such that numbers may be labeled more crucial than letters in a banking transaction. The equivalence algorithm of Broen is rather simplistic, and requires further development. Once the responses in the different equivalence classes have been tallied, MPITS performs a voting process to determine which is the majority response. For the configuration depicted in Fig. 1, the following possibilities exist:

– All three responses are put in the same equivalence class; the request is considered benign, and the response is forwarded to the external client. (Voting: 3-0)

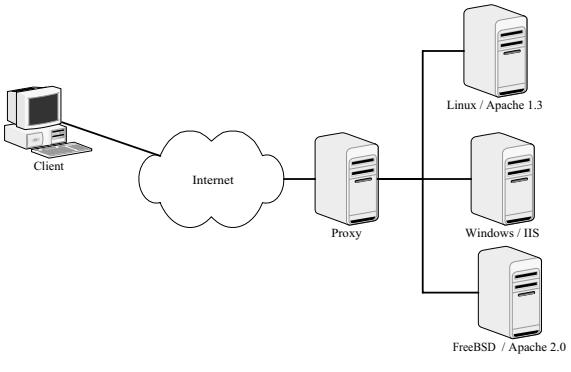


Fig. 1. An overview of the MPITS architecture

– Two responses are put in one equivalence class, and the last in another; the odd man out is considered compromised, and the response from the majority equivalence class is forwarded to the external client. (Voting: 2-1)
– All three responses are put in different equivalence classes; no determination can be made regarding which response is valid, and the system cannot generate a response. (Voting: 1-1-1; "Hung jury")

MPITS thus only provides a means for determining that something is amiss, but makes no attempt do do anything about the situation. It is therefore considered a basis or framwork for intrusion tolerance, rather than an intrusion tolerant system.

3 System Idea

The main goals of the ISH system are to deliver critical services to legitimate users even when under attack, detect and detain the attacker without alerting same, and provide bogus data to the attacker. This is illustrated in Fig. 2.

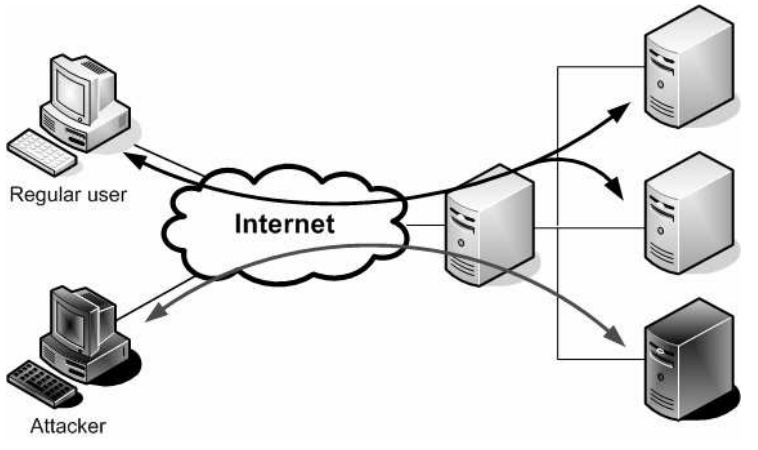


Fig. 2. A compromised component stringing an attacker along

The idea is that if a server unit is exposed to an exploit specific to that particular platform, the response will be different than the one generated by the two other units. All well-formed requests, on the other hand, should result in the same response or output. Once it is determined which unit is the odd man out, this unit can be isolated and removed from further voting.

4 System Overview

A logical description of the ISH system is given in Fig. 3. The fundamental components are as follows:

– Voter
– Router/Switch
– Logging unit

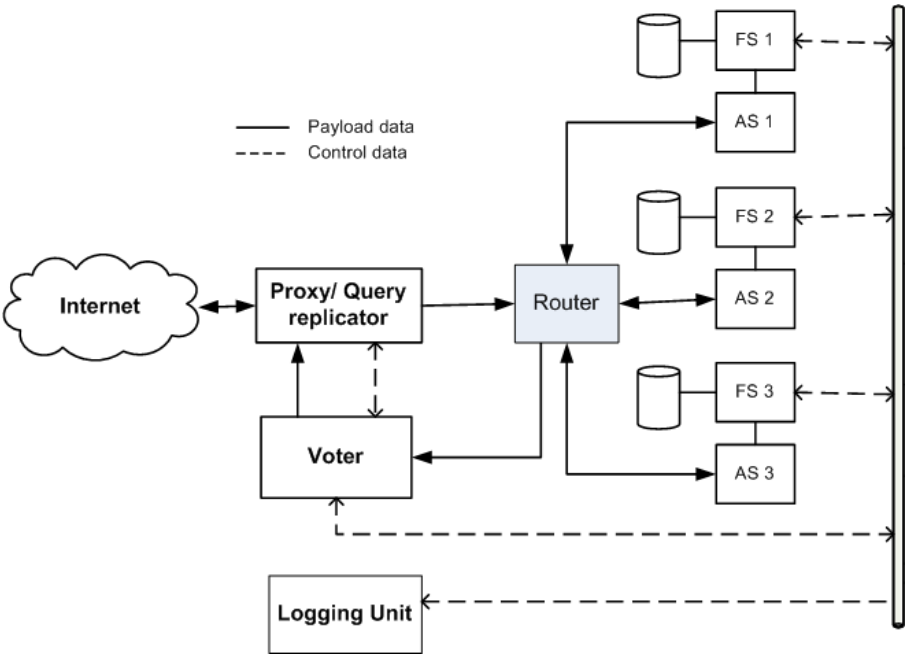


Fig. 3. Conceptual system overview

– Proxy
– Server units

In the following, we briefly describe each component.

Voter: The voter component is taken from MPITS, as described earlier. The voter is responsible for detecting attacks based on response discrepancies, and taking appropriate response.

Router/Switch: For connectivity, and also hiding internal network structure. This is a standard COTS component.

Logging Unit: A separate write-only loghost, for logging attacker activity. The logging is only activated once an attack has been detected.

Proxy: The proxy forwards requests and responses between client and server(s).

Server Unit: In these days of rootkits [17], it is difficult to trust any system that does not consist of pure hardware. Such systems, however, no longer exist. To up the ante against the attackers, we have designed a prototype server unit that allows us to verify the operating system files of an application server while it is still running.

The server unit is composed of two computers: A file server and the actual application server, as depicted in Fig. 4. The latter has no writeable file system of its own, but mounts an exported file system from the former.

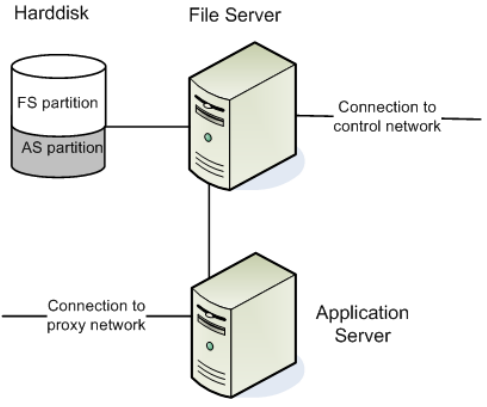


Fig. 4. Server unit

In addition to providing the application server with a file system, the file server also performs integrity checking of important files [18] to detect a compromise, and will report any anomalies to the voter.

Once a compromise is detected (either via the voter, or via the file integrity check), the file server will replace all sensitve data on the unit with obfuscated data prepared in advance. This is accomplished by maintaining a shadow volume that is periodically updated to keep it roughly equivalent to the real partition; the file server can then simply switch partitions – in all but the most unfortunate circumstances[3] this would be possible to achieve without the attacker noticing.

The dedicated file server could in theory be replaced by a Storage Area Network solution where an independent mechanism could verify the integrity of the files.

5 Implementation

For practical reasons we had to scale down our ambitions for the initial prototype implementation; while the original plan had called for three separate hardware/software platforms, e.g. Solaris on Sun, Windows on Intel and Linux on PowerPC, we had to settle for identical Intel PCs running FreeBSD and Linux, and two different versions of the Apache web server. The prototype ISH implementation is illustrated in Fig. 5.

[3] In the case of a web server with dynamic content, such an unfortunate circumstance could be that some information has significantly changed or been added just prior to the attack; the attacker might then notice that the information is different or missing on the "compromised" unit. This only applies to information that would be available to all clients, and not to information that first requires a breach of the access control mechanisms of the service.

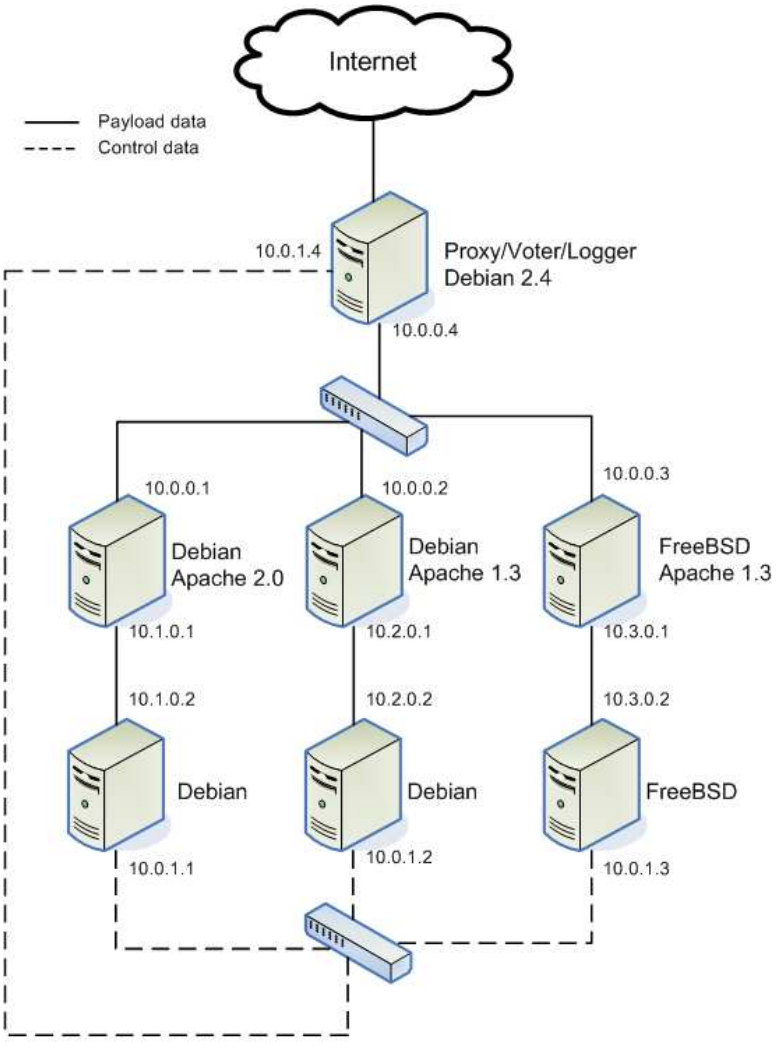


Fig. 5. Prototype System

The proxy unit is based on MPITS, extended with an attacker handling feature. Most importantly, when the voter detects a 2-1 voting anomaly, we no longer forward the majority response to the client; in this situation the client is assumed to be an attacker, and thus the minority response (from the compromised unit) is returned.

5.1 Identifying and Detaining the Attacker

Based on our description above, detecting that an attack has taken place based either on a voting discrepancy or on file integrity violation detection is fairly manageable (but see also section 5.2). However, new challenges arise when faced with the problem of identifying the attacker.

If we operated a single-user system, it would have been trivial: The current user would have been the culprit. With several concurrent users, it is more difficult – we have settled for labeling the first request causing a discrepancy in the voter as an attack, and using the originating IP address of this request as the address of the attacker. We acknowledge that this approach has its obvious downsides due to the common usage of VPNs, NATs and the fact that attackers may control several IP-addresses through bot-nets or the like. If the server utilizes a login feature, a user profile may be utilized, but for servers with open access such as web servers, this approach is not feasible.

Once the attacker has been identified, traffic originating from this source is no longer distributed to all the units; only to the compromised unit (now acting as honeypot). Responses from the honeypot is naturally not voted on, but passed on directly. In this sense, the user (i.e. attacker) interacting with the honeypot actually experiences improved efficiency with respect to an uncompromised system; this "spare capacity" may be used e.g. for extended logging of attacker activity.

5.2 False Positives and Negatives

The success of the voter relies on the assumption that well-formed input will result in the same output, regardless of which platform the service is implemented on. Unfortunately, practical tests have shown that this is not necessarily true, in rare cases causing the voting unit to detect a discrepancy based on legitimate input. This implies that careful configuration is required on each platform, and special modifications (e.g. suppresion of platform-specific staus messages) may be necessary to prevent spurious discrepancies. This of course also has implications for the specification of equivalence classes, as mentioned earlier.

Furthermore, it is possible that a given exploit would generate innocuous output, but have side-effects that causes a compromise further down the line. Since the voter is based on the generated output, such an attack would not be detected.

6 Discussion

Strictly speaking, dynamically transforming a system into a honeypot only makes sense as long as you are dealing with a human attacker; i.e. if you are concerned with detecting "dumb", indiscriminate network worms, a few ordinary honeypots sprinkeled around your domain would probably have served nicely. However, even in such cases the service duplication of our system will ensure that the adverse effects of the worm will be mitigated.

Another objection to ISH might be that it represents "security through obscurity" – one could argue that once it is known that a given installation is implemented by ISH, its value is nil. However, if ISH were to be deployed in the spirit of "Defense in depth" we claim that this does represent at least two additional lines of defense; no single-platform exploits would be applicable to our system, and any attack that does succeed, but changes one of the files in our "integrity check set" will still be flagged and cause the unit to be removed from service.

To focus on our idea of dynamic heneypot deployment, we have in our presentation intentionally omitted discussing other network intrusion detection mechanisms (e.g. [19], [20]), but we acknowledge that to the extent that ISH is an intrusion detection tool, it may be augmented by employing additional (traditional) IDS mechanisms, either before the traffic reaches ISH, or as part of ISH.

6.1 Determining the Optimal Number of Units

The use of three server units in our prototype means that once a single unit is transformed into a honeypot, a subsequent compromise of one of the two other units can be detected but not localised. This could be remedied by increasing the number of units[4] , but there will none the less be a point at which additional intrusions will mean that the entire system must be taken down. However, as long as systems are diligently updated with the latest patches (and otherwise protected against known attacks), such an occurence will be rare – zero-day exploits aren't *that* prolific. Also note that as long as the server units are on disparate platforms, repeat infections from automatic "bots" will be avoided, as only the unit that already is compromised will be vulnerable to a given exploit[5].

Also note that in contrast to a traditional honeypot, the goals of ISH do *not* include being penetrated by known exploits – we assume that in a production system, other measures will be in place that will be able to block known attacks. Thus, only new, otherwise undetected, and *successful*[6] attacks will cause ISH to transform a unit to honeypot state.

6.2 Application Areas

ISH would be applicable to business-critical services with high availability and security requirements. However, we realize that network administrators already have their hands full with managing the current crop of firewalls and intrusion detection systems. Furthermore, the ISH system also represents added cost for

[4] Or by using virutal machines.

[5] To be fair, this would ultimately require all the software to be devoloped according to "true" n-version programming [21].

[6] Attacks that are unsuccessful, either due to blocking by other mechanisms, or because the underlying system is not vulnerable to the particular attack, will not trigger honeypot deployment.

hardware (in the worst case multiplying the initial procurement costs by seven), software development (at least three-fold) and maintenance.

Thus, we presume that ISH would be of greatest interest to *managed security providers* for customers who offer such business-critical services to *their* customers. The managed security provider could deploy ISH to provide the service in question, but would additionally use it as a complement to their existing efforts in detecting new attacks/exploits. This would be of benefit not only to the current customer, but also to other customers of the managed service provider and to the community in general.

For the managed security provider, ISH would represent an improvement over a conventional honeypot or honeynet, in that the deployed ISH system would be a "real" system until it is successfully attacked.

7 Further Work

Our prototype ISH implementation is a very simple web server; a logical extension would be to implement a system for a generic service. There are also ample opportunities for less trivial extensions.

7.1 Code Integrity

Even though we practice defense in depth to detect intruders who do not trigger a voting anomaly, there still remains the challenge of detecting intrusions that do not alter the file system of the affected unit.

In a very interesting approach presented by Wang and Dasgupta [22], it is possible to verify the correctness of all static parts of a Linux kernel by employing a special autonomus "co-computer" (a single-board computer with access to the system bus). We believe this solution could be adapted to ISH to increase protection against arracks that leave the file system intact.

7.2 Attacker Separation

It would have been preferable to identify the attacker based solely on the session generating the exploit traffic, and used dynamically configured switches to route traffic from/to the attacker along a separate path. This would also allow us to separate an attacker from legitimate traffic originating from the attacker's IP address.

7.3 Darkhost Voter

Although we have strived to keep the proxy/voting unit simple, it still has an uncomfortable level of complexity when considering that it represents a single point of failure with respect to security.

If we can put the voting mechanism on an "invisible" (i.e. dark) host without an IP address, we leave only a very simple proxy and query replicator on the

publicly available host, while at the same time ensuring that the vital (and much more complex) voting unit cannot be adressed directly from the internet. The darkhost would have to craft packets with the proxy unit as originating address; this would require some fancy footwork with respect to ensuring that TCP sequence numbers etc. are properly maintained.

8 Conclusion

We have presented ISH, a prototype system that through duplication of server units detects new platform-specific attacks, and enhances the survivability of the system as a whole by transforming the compromised unit to a state of honeypot, while the uncompromised units continue to deliver service to legitimate users.

Acknowledgements

The research for this paper was partly carried out as part of Mr. Nyre's MSc thesis at NTNU, with subsequent additional testing by Mr. Sørensen. Further information is available at http://www.sislab.no/survive.html.

We are very grateful to Torgeir Broen for being allowed to use the MPITS code base as a starting point for our own efforts.

References

1. Ranum, M.J.: Thinking about firewalls. In: Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II) (April 1994)
2. Spitzner, L.: Honeypots – Tracking Hackers. Addison-Wesley, Reading (2003)
3. The Honeynet Project, Know Your Enemy – Revealing the Security Tools, Tactics and Motives of the Blachat Community, Addison-Wesley, Reading (2002)
4. Baecher, P., Koetter, M., Holz, T., Dornseif, M., Freiling, F.: The nepenthes platform: An efficient approach to collect malware. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 165–184. Springer, Heidelberg (2006)
5. Pouget, F., Holz, T.: A pointillist approach for comparing honeypots. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 51–68. Springer, Heidelberg (2005)
6. The HoneyNet Project. [Online]. Available: http://www.honeynet.org
7. Cheswick, B.: An evening with Berferd in which a cracker is lured, endured, and studied. In: USENIX Conference Proceedings, pp. 163–174. USENIX (1992)
8. Stoll, C.: Stalking the wily hacker. Communications of the ACM 31(5), 484–497 (1988)
9. Jaatun, M.G., Hallingstad, G.: Techniques for increasing survivability in NATO CIS. In: proceedings of the 1st European Survivability Workshop, February 2002, Köln-Wahn, Germany (2002)
10. Laprie, J.-C.: Dependable computing and fault-tolerance: Concepts and terminology. In: proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15), pp. 2–11 (1985)

11. Ellison, R.J., Fisher, D.A., Linger, R.C., Lipson, H.F., Longstaff, T., Mead, N.R.: Survivable network systems: An emerging discipline. Sofware Engineering Institute (SEI), Carnegie Mellon University, Tech. Rep. CMU/SEI-97-TR-013 (1997-1999)
12. Wang, F., Gong, F., Sargor, C., Goseva-Popstojana, Trivedi, K., Jou, F.: SITAR - a Scalable Intrusion-Tolerant Architecture for Distributed Services. In: proceedings of the 2001 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY, June 2001, pp. 38–45. IEEE Computer Society Press, Los Alamitos (2001)
13. Bait and Switch Honeypot.[Online]. Available: http://baitnswitch.sourceforge.net/
14. Snort - a network intrusion detection system. [Online]. Available: http://www.snort.org
15. Anagnostakis, K., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., Keromytis, A.: Detecting targeted attacks using shadow honeypots. ICS-FORTH, Crete, Greece, Tech. Rep. TR-348 (January 2005)
16. Broen, T.: Innbruddstolerante systemer: En eksperimentell utprøving og vurdering. Master's thesis, University of Oslo, Norway (May 2005)
17. Hoglund, G., Butler, J.: Rootkits, 1st edn. Addison-Wesley, Reading (2006)
18. Labs,S.: The Samhain file integrity system user manual, available from http://la-samhna.de/samhain/manual
19. Bace, R.G.: Intrusion Detection. Macmillian Technical Publishing (2000)
20. Northcutt, S., Novak, J.: Network Intrusion Detection, 3rd edn. New Riders Publishing (2002)
21. Ashrafi, N., Berman, O., Cutler, M.: Optimal-design of large software-systems using n-version programming. IEEE Transactions on Reliability 43(2), 344–350 (1994)
22. Wang, L., Dasgupta, P.: Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System. In: Proceedings of the Third IEEE International Symposium on Security in Networks and Distributed Systems, IEEE Computer Society Press, Los Alamitos (to appear, 2007)

How to Secure Bluetooth-Based Pico Networks

Dennis K. Nilsson[1], Phillip A. Porras[2], and Erland Jonsson[1]

[1] Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
[2] Computer Science Laboratory, SRI International
333 Ravenswood Ave., Menlo Park, CA 94025
{dennis.nilsson,erland.jonsson}@ce.chalmers.se, phillip.porras@sri.com

Abstract. We have examined Bluetooth-based Pico-network (Piconet) applications in wireless computing and cellular devices and found an extensive number of "unexpected abuses", where the security expectations of the device owner can be violated. We have studied the underlying causes of such problems and found that many products lack the controls to administer these devices securely. We also observed cases where explicit security claims from the Bluetooth protocol are not satisfied. We classify a number of abuses and security violations as Bluetooth protocol design flaws, application-layer implementation errors or simply pitfalls in the security management. Using this classification we define a core set of requirements that would improve security significantly.

Keywords: Bluetooth, pico networks, security controls, design flaws, implementation flaws.

1 Introduction

Wireless personal area networks are emerging with explosive growth as more devices and peripherals are becoming untethered and users continue to evolve toward the expectation of mobility in computing. Among the necessities for wireless computing has been the need to efficiently link small numbers of devices across short distance, and to this end one highly successful protocol that has emerged is Bluetooth.

Among its appeals, Bluetooth incorporates a low-cost, low-power, radio frequency management scheme for quickly forming ad hoc networks.

Unfortunately, there has been an extensive and continually growing list of "unexpected abuses" of Bluetooth-enabled devices that have plagued users and led to protocol revisions, as well as revisions to address the vulnerabilities in applications that overlay Bluetooth Pico Networks (Piconets). We have investigated and suggested countermeasures to more than a dozen such attacks.

This paper is a shortened version of a more extensive work [1]. The reminder of this paper is as follows. Section 2 discusses previous work related to identifying Bluetooth vulnerabilities. Section 3 provides a brief overview of Bluetooth network concepts. In Section 4, we categorize vulnerabilities based on where we

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 209–223, 2007.

believe the vulnerability was introduced. In Section 5, we define a core set of security requirements. Lastly, Section 6 summarizes our findings, and the paper is concluded in Section 7.

2 Related Work

In recent years, Bluetooth security has received significant attention. In particular, Bluetooth's authentication and data encryption services have been examined. A broad summary of vulnerabilities in the Bluetooth protocol was given in [2]. These vulnerabilities were categorized into weaknesses in the security concept, man-in-the-middle attacks, problems with encryption, uncontrolled propagation of radio waves and other security aspects. Some of the vulnerabilities mentioned in [2] have been taken into consideration in our paper.

Gehrmann and Nyberg [3] propose methods for link key establishment (pairing) through the use of Encrypted Diffie-Hellman key-exchange or Diffie-Hellman key-exchange with check values. These new suggested pairing procedures can replace the weak Bluetooth pairing procedure to improve security while the user's convenience is maintained.

Janssens [4] has listed a plethora of exploits that demonstrate many inherent security risks with Bluetooth devices. This extensive list contains descriptions of vulnerabilities but no solutions. Several additional studies have proposed security improvements to the authentication procedure and data encryption scheme [5,6], while others have suggested overlay components to facilitate better production of security credentials [7].

An analysis on worms spreading in a Bluetooth environment was performed by [8]. However, the analysis does not explain the underlying causes for the vulnerabilities that the worms use. Moreover, it does not describe how to prevent the worms from spreading.

3 Bluetooth Communication Basics and Security Features

Below we briefly summarize the procedures that allow devices to communicate and access services.

- The devices establish radio coordination by synchronizing on the same frequency-hopping pattern.
- The devices authenticate each other using a shared link key[1]. This is called Link Management Protocol (LMP)-authentication.
- If the devices do not possess the shared link key (e.g., the first time two devices communicate), the devices will first generate the shared link key and then perform authentication. This is called LMP-pairing.

[1] The link key is a symmetric key stored in both devices that can be used for future communication. Throughout this paper a link key is a combination key unless otherwise specified. Thus, the link key is specific for a link between two devices.

- The devices establish secure sessions and gain authorization to access network services.

Device Discovery and Connection
A device can be transitioned into a non-discoverable or non-connectable mode to control Piconet participation. Non-discoverable mode allows a device to remain silent to inquiry requests and avoid revealing its presence. Non-connectable mode allows a device to ignore connection requests. A device does not need to be in discoverable mode for it to be connectable (in other words, a device can be in non-discoverable and connectable mode) [9, vol.4, p.189-194].

LMP-Authentication
During LMP-authentication, illustrated in Fig. 1, one device is the *verifier* and the other device is the *claimant*. The verifier generates a random number and sends it to the claimant, which calculates a response using the shared link key and the received number. The response is sent to the verifier, which verifies that the claimant possesses the shared link key.

LMP-Pairing
The pairing procedure is illustrated in Fig. 1. Pairing control can be enforced by putting a device in non-pairable mode, where the device will ignore pairing requests [9, vol.4, p.195].

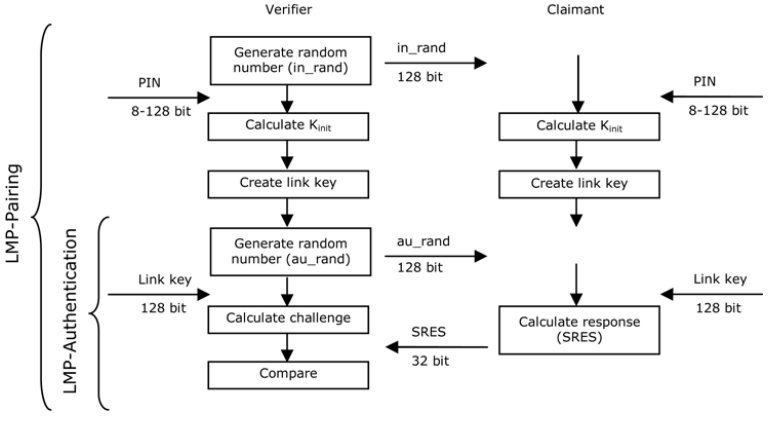


Fig. 1. LMP-Pairing and LMP-Authentication [9, vol.4, p.223-224]

Communications Security
Confidentiality in Bluetooth is achieved by using encryption. Integrity protection of the transmitted data is not part of the Bluetooth standard. However, there exist several integrity checks in the protocol, e.g., the baseband performs an integrity check using the 1-bit sequence number [9, vol.3, p.117].

Security Modes and Service Levels
There exist three security modes: *Mode 1* (nonsecure mode), no security; *Mode 2*, service-level security; and *Mode 3*, link-level security. In mode 2, secure communication may be imposed on a per network service basis. There are three levels of service security: *require authorization* (includes require authentication), *require authentication* and *require encryption* that can be used to implement security policies. For a device in security mode 3 with a security policy that requires encryption, secure communication is applied to the complete link before the devices are allowed to access network services [9, vol.4, p.198-200] [10,11].

4 Piconet Vulnerabilities and Their Relationship to Poor Security Control

We performed thorough analysis of the Bluetooth protocol specification and on the implementation of Bluetooth technology in several devices in all of the mentioned areas in Section 3. We tried known exploits with successful outcome and also discovered new exploits during our analysis. Table 1 summarizes a set of Bluetooth vulnerabilities. For each vulnerability, we identify the applicable area and classify the weakness as a manifestation of a Bluetooth protocol design flaw, an application-layer implementation error or a pitfall in the security management of noncustomizable Piconet devices. A *design flaw* is a specification flaw in the Bluetooth protocol, which requires revision of the protocol. The implementation errors we are interested in are caused for two reasons. The first obvious reason is an implementation flaw in the application itself caused by poor programming. The second reason is the lack of security controls necessary to disable or change the security policy for the vulnerable application. By implementation *error* we mean error in the implementation of Bluetooth functionality (the second reason). A *pitfall* is defined as a weakness that can be exploited by a third party but does not violate any security specifications. This vulnerability classification, based on terms from [12], is used in Table 1.

The following attacks are described by first explaining the *vulnerability* that an attacker could exploit using a specific *method*. The *cause* identifies the underlying problem causing the vulnerability. Lastly, a *countermeasure* against the exploit is suggested.

4.1 Device Discovery and Connection

A device owner may wish to hide a device from discovery and prevent participation in Piconets. Without proper device discovery and connection controls this is not possible.

Discovering Non-discoverable Devices

Vulnerability: Devices in non-discoverable mode can be found by a third party.
Method: A third party can check for the presence of a device that is in non-

Table 1. Summary of Bluetooth Device Vulnerabilities

Applicable Area	Vulnerability Classification	Exploit Name
Device Discovery and Connection	Pitfall — in non-connectable mode implementation	Discovering Non-discoverable Devices
LMP-Pairing	Error — in non-pairable mode implementation and PIN management	Peripheral Hijacking Reverse Peripheral Hijacking Online PIN Cracking Forced Re-Pairing
	Error — in non-pairable mode implementation and PIN management Design flaw — in Link Management Protocol	Online PIN Cracking 2 Offline PIN Cracking
LMP-Authentication	Design flaw — in Link Management Protocol	Relay Attack
	Error — in non-pairable mode implementation and PIN management Design flaw — in Link Management Protocol	Forced Re-pairing 2
Communications Security	Design flaw — in Bluetooth packet definition	Traffic Forging and Replay
	Pitfall and design flaw — dependence on PIN for secure key production is often limited by static or guessable PINs	Encryption Key Attack
Security Modes and Service Levels	Error — in require authorization, require authentication and secure mode 3 implementation	Bluejacking Bluesnarfing Bluebugging

discoverable and connectable mode to which it knows the Bluetooth device address (BD_ADDR) by submitting a connection request, to which the device would reply. Even when the device address is unknown, a device in non-discoverable and connectable mode can be discovered through bruteforce methods. Redfang is a tool developed to detect non-discoverable devices in this manner [13].
Cause: The device owner cannot transition the device into non-connectable mode, which potentially leads to the device being discovered in non-discoverable mode through bruteforce methods, which we consider a pitfall.
Countermeasure: Provide controls for device owners to transition a device into non-connectable mode and non-discoverable mode (see Section 5.1).

4.2 LMP-Pairing

We also find that Bluetooth implementations often fail to provide the device owner adequate control to transition between pairable and non-pairable mode. If a device is in pairable mode it is susceptible to several abuses.

Peripheral Hijacking

Vulnerability: For devices without a sufficient man-to-machine interface, pairing control is commonly enforced by physically pressing a button on the device. However, there exist devices that will respond to pairing requests even though the device is not set to this “pairing” mode.
Method: There are headsets that allow pairing even though the headset has not been set to “pairing” mode. An attacker can connect to the headset and pair with it by guessing the PIN.
Cause: The device owner cannot transition the fixed PIN device into non-pairable mode, which we consider an implementation flaw.
Countermeasure: Provide secure LMP-pairing control (see Section 5.2).

Reverse Peripheral Hijacking

Vulnerability: Certain devices engage in automatic Piconet reestablishment. The device owner cannot disable this feature.
Method: An attacker can assume the identity of a previously paired device by spoofing its BD_ADDR. The peripheral will then connect to the attacker believing that it is talking to the previously paired device. The attacker can guess the PIN and pair with the device.
Cause: The device owner lacks the ability to block automatic Piconet reestablishment and to transition the device into non-pairable mode. We consider this an implementation flaw.
Countermeasure: Same as Peripheral Hijacking.

Online PIN Cracking

Vulnerability: Devices that use a fixed PIN (usually four decimal digits) are susceptible to online PIN cracking. An attacker can simply try arbitrary PINs. In response to this threat, the Bluetooth protocol employs a delay scheme that increases the response time exponentially upon successive failed attempts from the same claimant, until a maximum delay value is reached [9, vol.3, p.799]. However, the attacker can spoof different BD_ADDR (i.e., assume the identity of different claimants) for every authentication attempt to circumvent this security mechanism.
Method: The attacker calculates a link key based on a guessed PIN. The fixed PIN device uses its fixed PIN to calculate the link key. The fixed PIN device (verifier) then verifies if the attacker (claimant) has calculated the correct link key. If the response does not match the challenge it means that the wrong PIN was guessed. The attacker starts over with another PIN and a different BD_ADDR. If the response matches, the attacker and the fixed PIN device are paired [4].
Cause: Same as Peripheral Hijacking.
Countermeasure: Same as Peripheral Hijacking.

Online PIN Cracking 2
We propose an improved version, illustrated in Fig. 2, to the previously described online PIN cracking method. Here, the attacker is the verifier. In the first version, the fixed PIN device was the verifier.

Vulnerability: Same as Online PIN Cracking.
Method: The attacker (A) calculates C_K_{init} based on a guessed PIN. The fixed PIN device (B) calculates K_{init} using the fixed PIN. Authentication on K_{init} is initiated by the attacker by sending the random number au_rand. Since B knows the correct PIN the response $SRES$ is correctly calculated. From this point, the attacker can compare the received $SRES$ to the calculated challenge. If they match the correct PIN is found. Otherwise, another PIN is tried until the correct PIN is found. By spoofing different BD_ADDRs (verifiers) for every authentication attempt the repeated attempt security mechanism is bypassed.

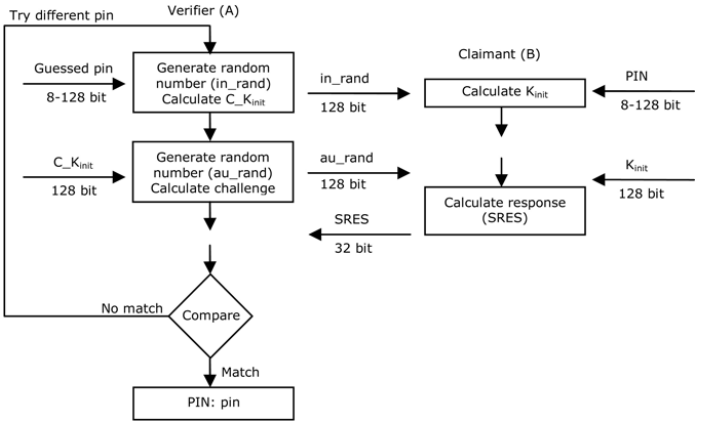


Fig. 2. Online PIN Cracking Technique

Cause: Information leakage in the challenge-response scheme is used to bruteforce the PIN. We consider this a design flaw. In addition, the device owner lacks the ability to alter the easily guessable fixed PIN and to transition the device to non-pairable mode, which we consider an implementation flaw.
Countermeasure: Prevent information leakage (revise the protocol), provide secure LMP-pairing control (see Section 5.2).

Offline PIN Cracking

Vulnerability: The messages exchanged during the pairing procedure can be used for offline PIN cracking.
Method: The messages needed to perform the offline PIN cracking can be acquired in two ways. The first way is to use one round of online PIN cracking, where the attacker sends the messages in_rand, au_rand and receives the message $SRES$.

The second way is to capture the messages exchanged between two devices during the pairing procedure by using, e.g., a Bluetooth protocol analyzer and packet sniffer [14].

Using these three messages (in_rand, au_rand and $SRES$), the PIN can be bruteforced offline. The initialization key (C_K_{init}) for a guessed PIN is calculated using the same in_rand, and then the verification algorithm is run on the calculated C_K_{init} using the same au_rand. If the calculated challenge does not match the captured $SRES$, an incorrect PIN was guessed, and the procedure starts over with another PIN. This procedure is repeated until the calculated challenge matches the $SRES$ value, in which case the correct PIN was guessed [15,16].
Cause: Same as Online PIN Cracking 2.
Countermeasure: Same as Online PIN Cracking 2.

Forced Re-pairing

Vulnerability: An attacker can take advantage of the limited link key store to force two devices to re-pair. By initializing a forced re-pair the attacker can choose an appropriate time and place for eavesdropping, which substantially simplifies the task of capturing the messages being exchanged during the pairing procedure. Possession of these messages allows offline PIN cracking.
Method: The attacker pairs with the fixed PIN device by successfully guessing the PIN. This creates a new link key that overwrites one of the stored link keys [17]. This procedure is repeated using different BD_ADDR for each pairing attempt until all stored link keys are overwritten. Alternatively, the attacker spoofs the BD_ADDR of a previously paired device if known, and then pairs with the fixed PIN device to just overwrite that link key. The result is that the previously paired device is forced to re-pair the next time it connects to the fixed PIN device since its corresponding link key in the fixed PIN device has been overwritten.
Cause: We consider this an implementation flaw since device owners lack the ability to transition the device into non-pairable mode and to alter the usually short and trivial ("0000") fixed PIN.
Countermeasure: Same as Peripheral Hijacking.

4.3 LMP-Authentication

The following attacks on the authentication procedure are due to design flaws in the Bluetooth protocol specification. These attacks assume that the devices involved have already established a relationship and share a link key.

Forced Re-pairing 2

Vulnerability: An attacker can inject messages during the authentication procedure to force two devices to re-pair.
Method: When two devices perform LMP-authentication, a random number, `LMP_au_rand PDU` (Protocol Data Unit), is sent from the verifier to the claimant. An attacker can inject an `LMP_not_accepted PDU` to the verifier claiming to be from the claimant before the real claimant replies with the calculated response `LMP_sres PDU`. If the verifier receives an `LMP_not_accepted PDU`, it believes that the claimant has lost its link key, and the two devices must pair again to create a new link key. Another method of forcing two devices to re-pair is if an attacker injects an `LMP_in_rand PDU` to the claimant claiming to be from the verifier before the verifier sends the `LMP_au_rand PDU` to the claimant. This causes the claimant to believe that the verifier has lost its link key, and pairing is started [16].
Cause: The authentication messages can be spoofed since there is no proper way to establish identity, which we consider a design flaw.
Countermeasure: Provide secure LMP-authentication (see Section 5.3).

Relay Attack

Vulnerability: An attacker can relay authentication messages and get authenticated without possessing the link key.
Method: An attacker can use two devices (E_A and E_B), connected to each other through some medium, to mount a man-in-the-middle attack against two pre-paired devices A and B. The attacker does this by establishing two Piconets: $A - E_B$ and $E_A - B$. A believes it is talking to B when in fact it is talking to E_B, and B believes it is talking to A when in fact it is talking to E_A. Messages are relayed through E_A and E_B.

There exist two methods to exploit this vulnerability depending on which device is initiating the authentication. In the first method the target device A (verifier) initiates authentication by sending a random number to E_B (claimant), which relays the number to E_A. E_A (verifier) passes the number onto B (claimant), which in turn calculates the correct response using its link key. This response is sent back to A via E_A and E_B. A verifies that this response is correct, and E_B is now authenticated [5]. This is not a proper authentication since E_B does not possess the link key. The second method is similar, but the attacker is initiating the authentication.
Cause: The authentication messages can be relayed since there is no proper way to establish identity. We consider this a design flaw.
Countermeasure: Same as Forced Re-pairing 2.

4.4 Communications Security

There exist weaknesses in how the encryption key is calculated, which can lead to unintended information leakage. Furthermore, the integrity aspect for communications security is bluntly missing [11].

Traffic Forging and Replay

Vulnerability: Bluetooth is subject to forging, modification and replay threats even if the link key is not known [4]. Moreover, when a link key is shared among several devices in a Piconet, as is the case with unit keys[2] and master keys[3], there exists no intra-Piconet secure communication.
Method: An attacker can capture packets and resend them into the Piconet with minimal interference from the Bluetooth sequence number, which is a single bit in length [4] [9, vol.3, p.117]. Messages can be forged and sent to devices in a Piconet to force re-pairing. Furthermore, an attacker can relay and replay authentication messages to get authenticated. Moreover, a slave device can read or inject traffic between the master and another slave when the link key is shared among the devices in the Piconet [18].

[2] A unit key is generated in a device the first time the device is started. This key is sent to devices with which it wishes to communicate. Thus, the same key is used for communicating with several devices [9, vol.3, p.780-781].
[3] A master key is generated by the master and shared among several slaves in the same Piconet [9, vol.3, p.783-784].

Cause: Messages can be forged since there is no proper way to establish identity. We consider this a design flaw in the Bluetooth packet definition since an attacker can successfully inject messages into the Piconet. We also consider it a design flaw to share symmetric keys that are used for authentication among several devices.
Countermeasure: Provide secure Piconet communication (see Section 5.4).

Encryption Key Attack

Vulnerability: An attacker can calculate the encryption key, if the link key is compromised, by capturing the random value sent initially between the two devices enforcing encryption. This means that an external entity can read confidential communications between Piconet participants. In Piconets using unit keys any slave can capture the random value sent initially between the two devices enforcing encryption and calculate the encryption key [18].
Method: After calculating the encryption key, the attacker can generate the same ciphering stream used to encrypt the data and can thus decrypt all messages in the encrypted communication [4].
Cause: The encryption key depends on the link key, and the link key in turn depends on the PIN. It is a pitfall since manufacturers often make PINs weak or trivially guessable. Encrypted data is not confidential between two devices when the encryption key is built on a shared link key available in other devices than those two devices, which we consider a design flaw.
Countermeasure: Provide secure Piconet end-point communication (see Section 5.4).

4.5 Security Modes and Service Levels

Bluetooth-enabled devices often provide a simplified interface for controlling the Bluetooth features in the device. As a result, controls for security modes and service levels are often missing.

Bluejacking

Vulnerability: The Object Push Profile (OPP) is used for pushing and pulling data to and from a device. For example, phonebook entries can easily be exchanged using the OPP. An implementation flaw in the OPP allows data to be pushed without authentication. The device owner cannot disable or enforce security on the vulnerable service.
Method: An attacker can connect to the OPP on a device and push data without authentication [19].
Cause: We consider this an implementation error since the device owner lacks controls to require authorization or authentication for the vulnerable service. In addition, the device owner lacks controls to transition the device into security mode 3, where link-level authentication is done.
Countermeasure: Provide security mode and secure service control (see Section 5.5).

Bluesnarfing

Vulnerability: An implementation flaw in the OPP allows data to be pulled without authentication. The device owner cannot disable or enforce security on the vulnerable service.
Method: An attacker can connect to the OPP on a device and pull private data without authentication [20].
Cause: Same as Bluejacking.
Countermeasure: Same as Bluejacking.

Bluebugging

Vulnerability: Certain devices provide a service for serial communications and support the AT command set[4]. An implementation flaw allows an attacker to establish a serial connection without authentication. The device owner cannot disable or enforce security on the vulnerable service.
Method: An attacker can gain full access to the AT command set without authentication. With these AT commands the device (in particular a cell phone) can be controlled to read/create/delete phone book entries, read/send SMS and make phone calls [20].
Cause: Same as Bluejacking.
Countermeasure: Same as Bluejacking.

5 Requirements for a Secure Piconet

In analyzing present Bluetooth attacks, we find that devices often lack the ability for users to control the security features of their devices. Furthermore, the security features in the Bluetooth protocol provides incomplete protection against the threats that they are expected to counter (Section 4).

In this section, we suggest an additional set of security requirements to address these issues. In Fig. 3, we illustrate the logical insertion points where our proposed security requirements fit in the link management and L2CAP[5] network service protocol stacks.

5.1 Device Discovery and Connection

We propose a combination of two security controls (1 and 2) to provide device owners an ability to fully isolate a device from communication.

– Requirement 1: A device should provide secure device discovery control. This is fulfilled if the device owner, and only the device owner, can transition the device between discoverable mode and non-discoverable mode.

[4] The AT command set is a language developed originally for controlling modem communications. It contains commands to dial, hangup, answer incoming calls etc.
[5] The logical link control and adaptation protocol (L2CAP) is responsible for higher-level protocol multiplexing and packet segmentation [9, vol.1, p.50].

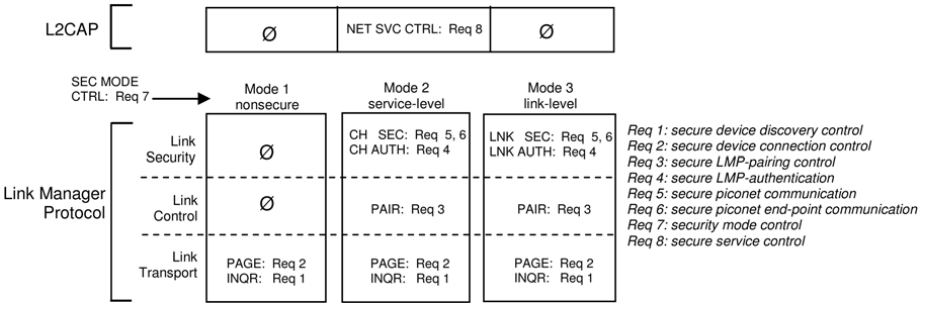


Fig. 3. Logical Insertion Points for Security Requirements for Bluetooth Devices

– Requirement 2: A device should provide secure device connection control. This is fulfilled if the device owner, and only the device owner, can transition the device between connectable mode and non-connectable mode.

5.2 LMP-Pairing

Device owners should have full control over when devices should engage in pairing. The credentials used should also be controlled by the device owner.

– Requirement 3: A device should provide secure LMP-pairing control. This is fulfilled if the device owner, and only the device owner, can
1) transition the device between pairable mode and non-pairable mode,
2) prevent the device from initiating pairing with unpaired devices,
3) prevent the device from participating in automated reestablishment of Piconets with previously paired devices,
4) alter the PIN.

5.3 LMP-Authentication

We express requirements for secure LMP-authentication to prevent the relay and forced re-pairing attacks.

– Requirement 4: A device should provide secure LMP-authentication. This is fulfilled if the device will only successfully LMP-authenticate with another device with which it shares a link key, and only the verifier or the claimant can reinitiate the LMP-pairing procedure.

5.4 Communications Security

We express confidentiality and integrity requirements for secure communication.

– Requirement 5: A Piconet should provide secure Piconet communication. This if fulfilled if no external entity can read communications between

Piconet participants, and no external entity can inject traffic that will be accepted by Piconet participants.

– Requirement 6: A Piconet should provide secure Piconet end-point communication. This is fulfilled if the Piconet provides secure Piconet communication, and no Piconet slave can read communications other than those between itself and the master, and no Piconet slave can inject traffic between the master and another slave.

5.5 Security Modes and Service Levels

Device owners should have controls to handle security modes and service levels. A device in security mode 3 requires authentication before any access to a service is granted.

– Requirement 7: A device should provide security mode control. This is fulfilled if the device owner, and only the device owner, can
  1) transition the device into security mode 3,
  2) require encryption for security mode 3,
  3) prevent the device from transition to security mode 1 (nonsecure mode).

– Requirement 8: A device should provide secure service control in security mode 2. This is fulfilled if the device owner, and only the device owner, can
  1) require authentication for all hosted services,
  2) require authorization for all hosted services,
  3) require encryption for all hosted services.

6 Areas of Concern

We have analyzed the vulnerabilities and identified three main areas of concern: *specification weaknesses*, *administration and management problems* and *device manufacturing issues*.

Specification Weaknesses
There exist several design flaws in the Bluetooth authentication procedures and encryption and integrity services. Bluetooth allows the relaying of authentication messages so that devices not possessing the shared key can successfully authenticate themselves. Moreover, an attacker can force Piconet participants into the pairing procedure, which can lead to the various link key compromises. Bluetooth offers no protection against replay attacks and offers no message integrity checks. To sum up, the current Bluetooth standard does not provide a robust protection for Piconet devices.

Administration and Management Problems
In Section 5, we define eight security requirements, whereof five are security control requirements that allow device owners to fully control the connection

and network service accessibility of their devices. Bluetooth devices offer very little control over the potential security management capabilities that devices in high-sensitivity environments need to make accessible. These include the ability for the device owner to control non-discoverable mode, non-connectable mode, non-pairable mode, service authorization control, service authentication control, secure mode transition and encryption control. In addition, Section 4 illustrates the importance of PIN management. Short PINs can lead to link key compromise or forcing a device to pair. Also, devices should avoid using fixed PINs. When fixed PINs are necessary, those PINs should be long and random (*not* "0000").

Device Manufacturing Issues
One problematic aspect of Bluetooth device management is that of providing significant configurability of the Bluetooth protection services, as well as the management of security credentials, when the device has a minimal man-to-machine interface. The Bluetooth protocol needs to consider techniques that allow greater security in credentials and greater flexibility in security management in environments where the user cannot provide significant input beyond a synchronization button or on-off switch.

7 Conclusion

We examined the causes of security abuses that can be mounted against Bluetooth devices, and suggested that many of these abuses can be prevented by using the security features already included in Bluetooth, if only the device owner had the ability to properly administer these features. We also observed cases where even explicit security claims from Bluetooth are not satisfied due to design flaws. We express our understanding of the needed security controls to securely manage Bluetooth devices by defining a set of security requirements, based on how abuses violate the current Bluetooth protocol and implementations.

We also consider the question of whether the current protocol can be used to develop Piconet applications that satisfy a higher degree of security than what is typically accepted in contemporary products. To fully address this question we believe that one must both remove the inherent flaws in the Bluetooth protocol design and equally important, incorporate greater owner control of the Bluetooth security features according to the proposed security requirements.

References

1. Nilsson, D.K., Porras, P.A., Jonsson, E.: Analyzing and Securing Bluetooth-based Pico Networks. Technical report, Chalmers University of Technology (2007)
2. BSI. Bluetooth, Threats and Security Measures. Technical report, BSI (2003)
3. Gehrmann, C., Nyberg, K.: Enhancements to Bluetooth Baseband Security. In: 6th Nordic Workshop on Secure IT-systems (NordSec) (2001)
4. Janssens, S.: Preliminary Study: Bluetooth Security. Technical report, Vrije Universiteit Brussel (2005)

5. Levi, A., Cetintas, E., Aydos, M., et al.: Relay Attacks on Bluetooth Authentication and Solutions. In: Aykanat, C., Dayar, T., Körpeoğlu, İ. (eds.) ISCIS 2004. LNCS, vol. 3280, Springer, Heidelberg (2004)
6. Ritvanen, K., Nyberg, K.: Upgrade of Bluetooth Encryption and Key Replay Attack. Technical report, Helsinki University of Technology (2004)
7. Rousseau, L., Arnoux, C., Cardonnel, C.: A Trusted Device to Secure a Bluetooth Piconet. In: Gemplus Developer Conference (2001)
8. Su, J., Chan, K.K.W., Miklas, A.G., et al.: A preliminary investigation of worm infections in a bluetooth environment. In: 4th ACM workshop on Recurring malcode, ACM Press, New York (2006)
9. Bluetooth SIG.: Bluetooth Specification Version 2.0 + EDR (2004)
10. Muller, T.: Bluetooth Security Architecture (1999)
11. Gehrmann, C., Persson, J., Smeets, B.: Bluetooth Security. Artech House, Inc. (2004)
12. Howard, J.D., Longstaff, T.A.: A Common Language for Computer Security Incidents (1998)
13. Whitehouse, O.: Bluetooth. In: CanSecWest (2004)
14. Frontline.: FTS4BT Bluetooth Protocol Analyzer & Packet Sniffer (2005)
15. Jakobsson, M., Wetzel, S.: Security Weaknesses in Bluetooth. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 176–191. Springer, Heidelberg (2001)
16. Shaked, Y., Wool, A.: Cracking the Bluetooth PIN. In: 3rd USENIX/ACM Conf. Mobile Systems, Applications, and Services (MobiSys), ACM Press, New York (2005)
17. Motorola: HS820 Wireless Headset with Bluetooth Technology (2005)
18. Bluetooth SIG. Bluetooth Security White Paper (2002)
19. Bluejackq: Bluejacking (August 2005), bluejackq.com
20. Laurie, A., Holtmann, M., Herfurt, M.: Hacking Bluetooth enabled mobile phones and beyond - Full Disclosure. In: Blackhat Security Briefings (2005)

Learning from Your Elders: A Shortcut to Information Security Management Success

Finn Olav Sveen, Jose Manuel Torres, and Jose Maria Sarriegi

Tecnun (University of Navarra)
Manuel de Lardizábal 13, 20018 San Sebastián, Spain
{fosveen,jmtorres,jmsarriegi}@tecnun.es

Abstract. Knowledge Management (KM), Quality Management (QM) and Safety Management (SM) are mature fields that have evolved and improved over time. Information security management (ISM) has aspects of these fields. E.g. tougher customer demands require continuous quality improvement, while new threats create a need for constantly improved security. Information technology brings new opportunities, but also challenges for KM, as it does for security. Organizations must comply with increasingly stricter safety laws, analogous to ISM requirements given by e.g. the Sarbanes-Oxley act. Research and practical experiences in KM, QM and SM have generated valuable insights that the younger, immature field of ISM can learn from. We present ten lessons and apply them to ISM. Key insights include the emphasis of good implementation over selection of model, the necessity of multi disciplinary teams, long term thinking, measurement, visualizing security costs, benchmarking, continuous improvement, collaboration, going beyond compliance and security as a competitive advantage.

Keywords: Information Security Management, Knowledge Management, Quality Improvement, Safety Management.

1 Introduction

The trend over the last years has been one of increasing amounts of security vulnerabilities. The CERT Coordination Center reported 1090 vulnerabilities in the year 2000. By 2006 the figure had multiplied eightfold to 8064 vulnerabilities[1]. Although some of this increase may be attributed to better vulnerability detection tools, it is quite clear that we are still facing serious challenges in Information Security Management (ISM).

We believe that it is necessary with a large, systematic effort to improve information security. The complexity of the information security problem requires commitment from all levels of an organization. Furthermore, no magic formulas will instantly fix the problem. However, we do not need to start entirely from scratch. Pioneering work has already been done in the information security community. Some

[1] http://www.cert.org/stats/

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 224–237, 2007.

examples are [1-3]. Furthermore, the process of improving ISM may be enhanced by looking towards other fields where similar problems have already been solved.

Fundamentally, achieving better information security means becoming better at learning. Not just on the individual level, but on an organizational and even inter-organizational level. We are continuously finding new and novel ways to use communications technology. It follows that new and novel methods of attacking the systems also will be found. In such a situation old knowledge quickly becomes obsolete. An example is computer security before and after the Internet. Daily anti-virus updates were unnecessary as the speed with which viruses spread were limited. When the Internet came that changed dramatically. The Internet Worm, ILoveYou and Melissa showed how fast malware can spread in a networked world. To keep up with and pre-empt advances in attack techniques, defenders must learn faster than attackers.

Effective continuous learning is necessary to achieve satisfactory information security in today's world. This makes ISM dependent upon effective Knowledge Management (KM) and Quality Management strategies. Furthermore, the challenges facing ISM have in many cases been encountered previously in these fields. E.g. tougher customer demands require continuous quality improvement, while new threats create a need for constantly improved security. Information technology brings new opportunities, but also challenges for KM, as it does for security.

An example of a field which has successfully leveraged both KM and QM is Safety Management (SM). Like ISM, SM is characterized by complexity. Multiple factors interact, creating dangerous situations and presenting considerable challenges for managers. These challenges go beyond purely technical issues, e.g., organizations must comply with increasingly stricter safety laws, analogous to ISM requirements given by e.g. the Sarbanes-Oxley act.

KM, QM and SM are all mature fields compared to ISM. There is an abundance of theories and studies, which is sorely lacking in ISM. By critically examining the experiences of KM, QM and SM, we can speed up the transition of ISM to a more mature state. Below we present ten lessons that we believe managers should learn.

2 Not Which But How

At the present there are more than 80 information security management models and standards [4]. The most widely known are ISO-17799 (BS-7799), COBIT , CERT-CC Security Improvement Modules, the IT Infrastructure Library, ADDME Approach and ISPME collection of standards. As a manager, which one should you choose? The obvious answer is that you should choose the one that fits best with your requirements and your situation. However, figuring out which one that is, is seldom straightforward.

The situation for KM is very similar. A few examples of KM models are Nonaka and Takeuchi's Five-phase model of knowledge creation [5], Davenport and Prusak's Knowledge Market model [6] and Probst, Raub and Romhardt's building blocks of knowledge management [7]. As the preceding shows there is wide variety of theoretical models.

QM standards are abundant. For example, there are standards for complaints handling (ISO 10002:2004), for software quality (ISO/IEC 25001:2007), for medical devices (ISO 13485:2003) as well as more generic standards like ISO 9000:2005, European Foundation for Quality Management (EFQM), The Malcolm Baldridge Award and the Capability Maturity Model. These standards build on more generic paradigms like the Plan-Do-Check-Act cycle, Total Quality Management (TQM) and Six Sigma [8], which encompasses a wide range of activities in an organization.

So, which methodology or standard do you choose? On the whole it does not matter. The theories and models are similar and overlap to some extent. Furthermore, there have been successful applications of a wide variety of KM and QM programs. Lee-Mortimer [9] describes an implementation of Six Sigma at a UK based manufacturing company. "Six Sigma has enabled the company to eliminate a wide range of long standing process variations." [9] Prajogo and Sohal [10] describe an implementation of a TQM program at an Australian automotive manufacturer. "…the success enjoyed by the company for many years is closely tied to its long history of implementing a sound quality management system." [10] Actually, you can find case studies about successful implementations of almost any model or methodology. All of them can be successful if correctly implemented. Within SM this can be seen through near-miss reporting systems. Near-misses refer to "almost accidents" or "close calls". While many such systems have been successful in improving safety, others have been less successful or complete failures. An example is a study of the introduction of near-miss reporting systems at two Danish factories [11]. The study found that only one of the two factories was able to improve safety. The authors state: "At Plant B the implementation of the incident reporting scheme failed (i.e., the company did not succeed in getting a system up and running, and thus the employees could not report any NM's [near-misses] and did not increase the reporting of MI's [minor incidents])."

Repenning and Sterman [12] studied process improvement and learning programs through more than a dozen case studies. In their words: "Most importantly, our research suggests that the inability of most organizations to reap the full benefit of these innovations has little to do with the specific tool they select. Instead the problem has its roots in how a new improvement program interacts with the physical, economic, social and psychological structures in which implementation takes place. In other words it's not just a tool problem, any more than it's a human resources problem or a leadership problem. Instead it's a systemic problem, one that is created by the interaction of tools, equipment, workers and managers."

Security is in many ways similar. To implement an effective security program the environment in which the program operates must be taken into account. Security may interfere with business needs, and thus be disregarded by managers and staff [13]. Focusing mainly on technical security leaves an organization open to social engineering attacks, as effectively demonstrated by Winkler [14] and Mitnick [15] who give real world examples. On the other hand, focusing solely on human factors leave computer networks open to technical attacks. Another example is exclusive focus on external threats which leaves the organization open to insider attacks, of which there are considerable amounts. In the FBI Computer Crime Survey 44% of responding organizations had experienced intrusions from within their own organization [16].

Security, like improvement and learning programs, is a systemic problem. It is not a matter of only installing the latest firewall, rather, "security is a process" [17], a process that must be carefully managed to ensure successful implementation and continuous effectiveness. It must take into account the full range of economic, technical, social, external and internal factors that affect the security of an organization. Currently, all information security models include the key factors necessary for success. More standards are confusing and unnecessary; focus should now be directed towards implementation.

3 Think Wide

The preceding discussion highlights another important need: multidisciplinary efforts. Systemic problems can not be tackled by a single discipline. We use the histories of KM and QM as examples. KM and QM are both complex problems, characterized by complex environments with social and technical factors. These are characteristics they share with ISM. We will examine KM and QM in turn and see that the historic developments of both fields are similar.

McElroy distinguishes between two generations of KM [18]. The first generation focuses on knowledge integration. It was thought that the organizations possessed the right knowledge, but that often it was not available to those who needed it. First generation KM therefore focused on tools to facilitate knowledge sharing. Often these initiatives were technically oriented, focusing on data repositories and the like.

However, this ignores the important element of how organizations create new knowledge. Nonaka and Takeuchi were among the first to recognize this [5]. They developed a theory of knowledge creation which emphasizes that knowledge has both tacit and explicit elements. First generation knowledge management only takes the latter into account. The tacit element, developed mainly through social interaction has a key role in ensuring both knowledge transfer and creation [5]. Second generation KM focuses on accelerating the creation of new knowledge [18]. KM has transitioned from a mainly technical viewpoint of knowledge repositories to encompass both the technical systems needed to store information and communicate over large distances, and the social processes that are necessary for knowledge creation and transfer.

QM methodologies initially focused on statistical tools [19]. Although first developed in the 1920s they first became popular during World War II. Quality was inspected into the product at the end of the assembly line [19]. The focus was on detection and fire fighting, not prevention [19]. Defective products were not detected until after they had gone through most of the production process. Furthermore there was no systematic methodology for preventing such defects in the future. In the 1950s and 60s the focus shifted from quality control to quality assurance, i.e., a shift from detection to prevention [19].

With the advent of TQM, quality became a company wide endeavor. Systematic learning and improvement became a part of quality management paradigms. Total quality means that the whole business process, not just the end product is subject to improvement and control. This was first practiced in Japan, which in the post war years took to heart the lessons of quality gurus such as Deming and Juran [19]. It can be argued that the Japanese quality movement, encompassing the whole of the

enterprise, is the reason for their strong competitive advantage in the 70s and 80s. Unlike western businesses, Japanese businesses saw quality as the responsibility of the whole company [19].

KM and QM have gone from focusing on a limited set of tools and areas to encompassing the whole business. In essence KM and QM are both multifaceted problems. One has to take into account all stages of a process: technical, organizational and social issues. Thus, to solve systemic problems systemic solutions are necessary.

Another example comes from SM. A brief review of the safety literature reviews that achieving high safety standards requires, among others, the active participation of top management, dedicated safety personnel, investigators who are skilled at finding accident/incident causes, engineers, ergonomics specialists, health care personnel, operators at the sharp end, suppliers and contractors [20-24]. None of these groups can achieve high safety standards on their own, as modern accidents are often the cause of systemic, organizational factors [25].

Traditionally, information security has been the domain of IT administrators. However, information security relies equally on non-technical disciplines. Detecting insiders may require psychological profiling [26], while legal action requires the support of lawyers, especially because computer crime often crosses international borders. Single vector solutions are no longer sufficient to counter today's diverse threats [27]. We have reached a time where multidisciplinary teams (engineers, economist, lawyers, and policymakers) must try to forge common approaches in the name of information security management improvements [2].

4 Think Long

The issue of short-term gain versus long-term success regularly arises in the business media. The tendency to favor short-term gain is often much lamented. The work of Repenning and Sterman [12] indicate that short-term strategies are not so much chosen, but rather forced. Improvement programs require time and resources to implement successfully, and organizations seldom have unlimited resources. When a crisis hits, e.g., missing a production target, resources may be routed away from improvement processes to amend the problem, giving short term relief. However, removing resources from improvement processes reduces future benefit from them. Thus in time, the lack of improvement causes more crises, leading to yet more resources being taken away from improvement, again, causing yet more crises. And so the organization reaches a vicious circle which is hard to exit. A similar effect has been found in safety. The tendency to correct production shortfalls by bypassing safety, leads, in the long run, to poorer safety and also poorer production. The topic of bypassing safety will be revisited later.

The natural tendency to fall into a short-term reactive mode is further strengthened by the long time that is often required to implement improvement and learning programs [12]. For example, developing TQM programs in marketing departments may take seven to eight years [28]. When the benefit of the improvement work is far away, it becomes easier to fall into a short-term reactive mode. In Dörner's [29, p. 198] words: "We human beings are creatures of the present."

The same problem exists within the realm of information security as well. Wiik et al. [30] describes how a Computer Security Incident Response Team (CSIRT) can fall into a "capability trap". Working harder to alleviate short term pressures takes resources away from long term improvement efforts. The team falls into the same vicious circle as described by Repenning and Sterman [12]. Symptomatic of both Repenning's [12] and Wiik's [30] work is that the solution involves a worse-before-better scenario, making any exit from the circle of crises difficult.

A study by Royal Dutch Shell indicate that the longest living companies are those who are most sensitive to their environment [31]. They react to new threats and changes in the marketplace with foresight, before it is too late [31]. Skipping out on security today, may mean disaster tomorrow.

5 Always Improve

This lesson is closely connected with our previous one. As has been previously mentioned, the post-war success of the Japanese economy can be attributed to their focus on quality management. However, something more fundamental is at work. Fueling the continuous advance in quality was their ability to innovate. Not once or twice, but continuously [5]. The ability to innovate is closely connected with the creation of new knowledge [5]. The only sustainable business advantage is the ability to learn faster than your competitors [18]. Everything else can in principle be copied. This is especially true in today's business climate where the focus is on knowledge and less on resources, capital and labor. Thus organizations must become "learning organizations" [32]. And in our opinion, in the area of information security they must learn fast.

Today organizations are continuously facing new information security challenges. The sustained development of new attack methods must be met by a sustained effort to create counter methods. And at heart of this is the creation of new knowledge faster than competitors, or in this case, attackers, especially as offence is much easier than defense. The attacker can attack anywhere while the defender has to defend everywhere [2]. Attack-tool developers, hackers, crackers and insiders currently dictate the speed of information systems' insecurity. Continuous improvement programs for information security will help organizations to keep up with increasing threat sophistication. Furthermore, just keeping up is not enough. The lead must be taken back and continuous improvement programs may hold the key to this.

6 Together Is Better

Sharing information on risk factors, realized or not, enables learning from the mistakes and solutions of others. This is a potentially powerful lever, as reinventing the wheel is both time-consuming and expensive. A good example is NASA's Aviation Safety Reporting system where thousands of incident reports are logged every year. Each report is analyzed and the information is fed back to airlines, regulators, airplane producers and airports, enabling them to improve routines and equipment [21]. In this way future accidents are avoided or mitigated. It is true that

airplane accidents have not disappeared, but countries which have well working aviation safety reporting systems perform better than those who do not. For example the accident rate in Taiwan is higher than in the US and western Europe, something which has been attributed to a poorly functioning national reporting system [33].

As previously mentioned, Senge [32] stresses that organizations must become "learning organizations". Those organizations that do not take advantage of all sources of knowledge will fall behind those who do. Knowledge is not just created inside an organization, but an organization interacts with its environment, absorbing and creating new knowledge [34]. A crucial insight of Nonaka and Takeuchi is that knowledge is created through interaction [5].

Davenport and Prusak [6] speak of knowledge markets. According to them a firm has an internal knowledge market consisting of sellers, buyers and brokers. A seller's willingness to share knowledge is dependent on three conditions: 1) Reciprocity, i.e. expectations of future advantage by sharing knowledge now. 2) Repute, i.e. becoming known as a knowledgeable person and the status that entails. 3) Altruism, i.e. sharing because one likes to help. If we extend Davenport and Prusak's concepts to an industry sector or the society as a whole, we can infer that companies will not share their knowledge if there is no advantage for them to do so. Furthermore, the advantage must be mutual. In our opinion it is unlikely that commercial organizations will share their knowledge for reasons of repute or altruism.

It follows that the knowledge an organization can absorb from the outside is limited unless there is genuine reciprocity. Furthermore, by not sharing knowledge with others, knowledge creation is also reduced, as the crucial social interaction that is necessary for knowledge creation becomes restricted to inside the organization. Large multinational organizations may have the resources necessary to keep information security internal, but the majority of businesses are too small to operate entirely independently. Organizations can not afford to shut themselves off from the outside.

The benefit of sharing information has been recognized by government and the security community for some time. The federal US government has encouraged the establishment of Information Sharing and Analysis Centers (ISACS) [35]. However distorted incentives [2] and the tendency to free ride [36] are detriments to successful information sharing. We believe that these challenges must be overcome for security to improve on a larger scale. Although an extreme case, the terrorist attack of 9/11 is a revealing example. The 9-11 Commission Report highlights how the lack of information-sharing between the FBI, CIA and other important parties impeded the detection of the attackers' intentions and thus the actual attack [37].

7 The Boiled Frog

Measuring information security is hard. Currently there is no reliable, absolute measure for security level. Information security exists in the context of a social, dynamic system of which only a few of the elements are directly visible. For example, actual incident rates may be masked by underreporting [38]. Indicators tend to be heavily biased, indirect and subject to a wide range of different interpretations.

However, it is still necessary to measure. It is a widely accepted principle that an activity cannot be managed properly if it is not measured. Setting goals without the

ability to measure whether the goal have been reached is meaningless. Furthermore, as Senge [32] eloquently explains, changes that take place over long time periods tend to go unnoticed. Senge recounts an experiment where a frog is placed into a bowl of water. If the frog is placed in a bowl of boiling water, it jumps out as it instantly recognizes the heat. However, if the frog is placed in a bowl of cold water which is then slowly heated, the frog will happily stay in, even if the water starts to boil. The frog's sensory system is geared towards detecting sudden changes in the environment, whereas slow, long term changes are not noticed.

In quality management this need to measure can be seen in the rise of Six Sigma. Compared to TQM, Six Sigma is not a radically new paradigm. It is the natural evolution towards "data based management".

To avoid "being boiled", that is not noticing degradation of security or an increased threat, adequate measures for information security must be developed. Experiences from KM show us that it is possible to develop a wide variety of indicators to measure intangible resources. Sveiby [39], Brooking [40] and Edvinsson [41] have developed methods for measuring the value of intellectual capital.

8 Show Me the Money

Numerous case studies and textbooks in KM, QM and SM points to the importance of top management support. Without top-management support new initiatives are not followed up or prioritized. To gain top-management support it is necessary to evaluate the economic impact. Top-management is unlikely to sanction initiatives which have no clear impact on the bottom line, long-term or short-term. Thus, sooner or later it becomes necessary to justify information security expenses.

This has also been recognized in QM where there has been a long tradition of measuring quality costs. For example Six Sigma makes an explicit connection between the bottom line and the improvement effort. Pande et al. urges Six Sigma teams to measure the "Cost of Poor Quality", as it "speaks a language that almost anyone understands: money" [8, p. 231].

In security the direct expenses such as wages, the cost of security-equipment and -software is easy to measure, whereas the costs of security breaches are more difficult ascertain in advance. Nevertheless, the cost of poor security should be measured. Waiting until it is actually realized, i.e., when a serious incident strikes, leaves the organization unprepared as security then is an invisible part of the agenda. Although it is difficult to measure the implicit costs of incidents, methodology for measuring such costs exist [42]. If the cost of poor security is not known it is also difficult to know how much should be spent on security.

9 Benchmark

Benchmarking is an accepted and effective way of assessing an organization's performance compared to leading organizations. Information security is seldom considered a primary activity. Thus, being leading innovators in ISM is perhaps beyond most organizations. Benchmarking provides these organizations with

important information on best practice. However, in information security it is difficult to know what best practice is. Deep probing surveys and case studies are sorely lacking.

The lack of such research has its natural explanation in the very nature of information security data. Such studies would expose an organization's vulnerabilities, which might be used by potential attackers, including competitors.

Contrasting the situation for information security with that of KM, QM and SM, we find numerous surveys and case studies, ranging from more superficial studies of whole industry sectors to deep probing studies of single companies. For example Prajogo and Sohal [10] follow a company's QM practices closely for twenty years. Nonaka and Takeuchi [5] give in-depth examples from individual projects in many different companies. With a battery of case studies the Knowledge Management Casebook lays bare the KM practices of Siemens [43]. The Best Practices in Knowledge Management & Organizational Learning Handbook has case studies of companies such as Ernst & Young, AT&T and Microsoft [44]. Within SM there is a tradition of sharing. Examples include sharing of accident classification and reporting routines and studies of why accidents happen, such as Vaughan's study of the Challenger disaster [45] and Tsuchiya et al's study of the Tokaimura nuclear criticality accident [46]. The existence of such widely available studies enable comparison with industry leaders, and thus also a measure of where to improve your own organization.

Without knowing where you are it is impossible to know where to go. Thus, ways must be found to overcome the information security "data barrier". Our own team has been working on this problem for some time [47-49].

10 Secure Advantage

Traditionally, security has been seen as an expense, a necessary evil. This is much the same view as that which has been taken on safety. However, research on safety show that a high level of safety is not just a necessary evil, but a business enabler and to some extent, even a competitive advantage.

Cooke describes the Westray mine disaster [50]. Prior to the disaster accidents caused production delays. To recoup the lost time, managers, and thus also workers, started to cut corners, disregarding safety procedures. This lead to more accidents, more lost time, and thus even more corner cutting. Westray was in a vicious circle, a problem that constantly amplified itself. Ultimately, the mine suffered a disaster. The preceding supports the notion of safety as a necessary evil.

However, Cooke did not just describe the historic development of the disaster. He also created a computerized simulation model of Westray mine. The results of which is interesting. By shifting focus from recouping time to prioritizing safety the organization could have been able to turn the vicious circle into a virtuous circle. More focus on safety leads to less lost time, which leads to more production targets being met. This again reinforces the pressure on safety, as the urgency of meeting the production schedule is lessened, causing yet fewer accidents, less lost time and again a more efficient production. In this latter case, safety is not just a necessary evil, but a

real business enabler. It can even be argued that it is a competitive advantage, as long as the competition is not doing the same.

We argue that it is likely that the same kind of dynamics can be found in information security. Organizations that are constantly plagued by computer problems loose precious time, and like in safety, the time loss may make managers and employees take shortcuts, which may lead to more problems. In a sense this kind of behavior is counter intuitive. One would expect that accidents or security incidents would cause people to be more cautious. However, strong economic pressures to produce may overshadow such learning.

Organizations may gain an economic advantage by the reduction of information security incidents. However, there is also an indirect advantage to good information security. For an explanation we look to QM. The various certifications and awards that exist within quality management have given organizations a means with which to advertise their excellence within the area. When a company has been certified in ISO-9000 or have received a quality award, customers know that the company meets a certain standard.

With the increasing use of information technology to transfer information, manage bank accounts and buy goods online, the need for security is not going to go away. On the contrary it will become even more important. Thus being certified in a security standard not only ensures that the company meets a high standard, but it also constitutes an advertisement tool for customer confidence.

11 Compliance Is Not Enough

Organizations looking to improve their security have, as previously mentioned, several different standards available. These standards provide a good starting point. However, there is an inherent danger in relying on standards and regulations. Relying on them may shift the focus away from continuous improvement towards conformity with requirements, a tendency which have been noted in among others, the French healthcare system [51].

A survey of medical device makers indicate that top performers have an edge on the competition because they focus on proactive quality topics [52]. Numerous organizations have moved to get ISO-9000 registration; however, many do so not out of an actual wish to improve quality. Chelsom states that: “Many organizations have obtained ISO[-9000] registration simply because major clients require them to do so... While in response, many suppliers have obtained ISO-9000 approval through a genuine quality improvement effort, others have paid lip-service to their customers’ demands, ...” [53] In the words of Karapetrovic: “Although it may seem illogical at the first glance, ISO-9000 is not about compliance at all. It is about quality systems and effective and efficient ways to assure and improve quality” [54].

In the context of ISM, the concern over regulatory biases have been mentioned by Caralli and Wilson [55]. The experiences from QM show that forgetting improvement when complying with standards is not just a possibility, but a real concern. [ISO-9000] is the lowest common denominator of any successful quality system.” [54] The same applies to information security standards. Organizations have to go past

compliance towards continuous security improvement. Complying with a standard is not a pillow that you can rest your head on, compliance is only a starting point.

12 Conclusion

ISM is fundamentally about learning and improving quality. Therefore we have highlighted some of the crucial lessons of KM, QM and SM, and we have related them to ISM. If information security managers do not realize these lessons, they risk a costly re-invention of the wheel.

First, we have highlighted that choosing a method is not so important, but that process implementation is paramount. Second, we have stressed the importance of thinking holistically. ISM must move from a technically oriented discipline towards one that encompasses both technical and organizational issues. Third, long term sustainability must take precedence over short term gains. Organizations must evaluate the security threat before implementing new business systems. Fourth, the process nature of ISM makes it necessary to continuously improve to meet new threats that arise from changing conditions. Fifth, communication with external actors is crucial for accelerating knowledge absorption and generation. Few companies have the resources available to go at ISM entirely alone. Sixth, assessing the organization's performance is necessary. To stop the ship from taking on water one must first know where the holes are. Seventh, top-management support is crucial. To gain it one must speak their language, namely money. Eighth, organizations should compare their performance with the top performers. Ninth, compliance with standards is a good start, but only a start. Tenth, security should be leveraged as a business enabler.

Information Security Management is currently preoccupied with technical measures and short sightedness. Committing to the ten lessons above will help move ISM towards a more mature state that will lead to an improved security situation. Like KM, QM and SM, ISM must become something that you simply do. It must become a natural part of any business that relies on information technology, which today are most businesses.

The subtitle of this paper is "A Short Cut to Information Security Management Success". The only short cut lies in realizing the ten lessons above. The hard work of implementing them still remains. And lastly we would like to say that these are probably not all the lessons that an ISM manager should learn, but they are a starting point. The lessons above are all common sense, but in our research we have yet to find any company who actually do all of the above.

References

1. Andersen, D.F., et al.: Preliminary System Dynamics Maps of the Insider Cyber-Threat Problem. In: 22nd International Conference of the System Dynamics Society, The System Dynamics Society, Oxford, England (2004)
2. Anderson, R.: Why information security is hard - an economic perspective. In: 17th Annual Computer Security Applications Conference (2001)

3. Gonzalez, J.J. (ed.): From Modeling to Managing Security: A System Dynamics Approach. Research Series, vol. 35. Norwegian Academic Press, Kristiansand, Norway (2003)
4. Putnam, A.: Information Security Management References. In: U.S.H.o. Representatives (eds.) Mapping of Existing Work on Infosec "Best Practices" Subgroup (2004)
5. Nonaka, I., Takeuchi, H.: The Knowledge-Creating Company. Oxford University Press, New York & Oxford (1995)
6. Davenport, T.H., Prusak, L.: Working Knowledge: how organizations know what they know. Harvard Business School Press, Boston Massachusetts (1998)
7. Probst, G., Raub, S., Romhardt, K.: Managing Knowledge: building blocks for success. John Wiley & Sons, Chichester (2000)
8. Pande, P.S., Neuman, R.P., Cavanagh, R.R.: The Six Sigma Way: how GE, Motorola, and other top companies are honing their performance. McGraw-Hill, New York (2000)
9. Lee-Mortimer, A.: Six Sigma: Effective Handling of deep rooted quality problems. Assembly Automation 26(3), 200–204 (2006)
10. Prajogo, D.I., Sohal, A.S.: The Sustainability and Evolution of Quality Improvement Programmes - An Australian Case Study. Total Quality Management 15(2), 205–220 (2004)
11. Nielsen, K.J., Carstensen, O., Rasmussen, K.: The Prevention of Occupational Injuries in Two Industrial Plants Using an Incident Reporting Scheme. Journal of Safety Research 37(5), 479–486 (2006)
12. Repenning, N.P., Sterman, J.D.: Nobody ever gets credit for fixing problems that never happened. California Management Review, 43(4) (2001)
13. Schultz, E.: The human Factor in Security. Computer&Security 24, 425–426 (2005)
14. Winkler, I.: Spies Among Us: How To Stop The Spies, Terrorists, Hackers, And Criminals You Don't Even Know You Encounter Every Day. Wiley, Indianapolis (2005)
15. Mitnick, K.: The Art of Deception. Wiley, Chichester (2002)
16. Abagnale, F., et al.: FBI 2005 Computer Crime Survey. Federal Bureau of Investigation (2005)
17. Schneier, B.: Secrets & Lies: Digital Security in a Networked World. Wiley, Chichester (2000)
18. McElroy, M.W.: The New Knowledge Management. Butterworth Heinemann, Amsterdam (2003)
19. Yong, J., Wilkinson, A.: The long and winding road: The evolution of quality management. Total Quality Management 13(1), 101–121 (2002)
20. Collinson, D.L.: Surviving the rigs: Safety and Surveilance on North Sea Oil Platforms. Organization Studies 20(4), 579–600 (1999)
21. Johnson, C.: Failure in Safety Critical Systems: A Handbook of Incident and Accident Reporting. Glasgow University Press (2003)
22. Kjellén, U.: Prevention of Accidents Through Experience Feedback, p. 450. Taylor & Francis, London and New York (2000)
23. Morag, I.: Intel's Incident-free Culture: A Case Study. Applied Ergonomics 2006(38), 201–211
24. Phimister, J.R., et al.: Near-Miss Incident Management in the Chemical Process Industry. Risk Analysis 23(3), 445–459 (2003)
25. Reason, J.: Safety in the operating theatre - Part 2: Human error and organizational failure. Quality and Safety in Health Care 14, 56–61 (2005)
26. Shaw, E.D.: The role of behavioral research and profiling in malicious cyber insider investigations. Digital Investigation 3, 20–31 (2006)

27. Campbell, S.: How to Think About Security Failures. Communications of the ACM 49(1), 37–39 (2006)
28. Fram, E.H.: Not so strange bedfellows: marketing and total quality management. Managing Service Quality 5(1), 50–56 (1995)
29. Dörner, D.: The Logic of Failure. Perseus Books, Cambridge, Massachusets (1996)
30. Wiik, J., Gonzalez, J.J., Kossakowski, K.-P.: Limits to Effectiveness in Computer Security Incident Response Teams. In: 23rd International Conference of the System Dynamics Society, Oxford (2004)
31. Geus, A.d.: The Living Company. Harvard Business School Press, Boston Massachusetts (1997)
32. Senge, P.: The Fifth Discipline. Bantam Doubleday Dell Publishing Group, London (1990)
33. Lee, P.I., Weitzel, T.R.: Air Carrier Safety and Culture: An Investigation of Taiwan's Adaptation to Western Incident Reporting Programs. Journal of Air Transportation 10(1) (2005)
34. Sveiby, K.-E.: A Knowledge-based theory of the firm to guide strategy formulation. Journal of Intellectual Capital 2(4) (2001)
35. Gal-Or, E., Ghose, A.: The Economic Incentives for Sharing Security Information. Information Systems Research 16(2), 186–208 (2005)
36. Gordon, L.A., Loeb, M., Lucyshyn, W.: Sharing information on computer systems security: An economic analysis. Journal of Accounting Public Policy 22(6), 461–485 (2003)
37. The 9-11 Commision Report (2002)
38. Rich, E., Sveen, F.O., Jager, M.: Overcoming Organizational Challenges to Secure Knowledge Management. In: Secure Knowledge Management Workshop, New York, US (2006)
39. Sveiby, K.-E.: The new organizational wealth. In: Managing & measuring knowledge-Based assets, Berret-Koehler Publishers Inc., San Francisco (1997)
40. Brooking, A.: Intellectual capital: Core asset for the third millennium enterprise. Itp-Intern.Thomson Publishing, London (1997)
41. Edvinsson, L.: Intellectual capital. Harper Collins Publishers, New York (1997)
42. Gordon, L.A., Loeb, M.P.: Managing Cyber Security Resources: A cost-benefit analysis. McGraw-Hill, New York (2006)
43. Davenport, T.H., Probst, G.: Knowledge Management Case Book: Siemens Best Practices, 2nd edn. Publicis Corporate Publishing and John Wiley & Sons, Erlangen (2002)
44. Harkins, P., Carter, L.L., Timmins, A.J.: Linkage Inc.'s Best Practices in Knowledge Management and Organizational Learning Handbook. Linkage Press, Lexington, Massachusetts (2000)
45. Vaughan, D.: Autonomy, Interdependence and Social Control: NASA and the Space Shuttle Challenger. Administrative Science Quarterly 35(2), 225–257 (1990)
46. Tsuchiya, S., et al.: An Analysis of Tokaimura Nuclear Criticality Accident: A Systems Approach. In: The 19th International Conference of the System Dynamics Society, System Dynamics Society, Atlanta, Georgia (2001)
47. Torres, J.M., et al.: Managing Information Systems Security: Critical Success Factors and Indicators to Measure Effectiveness. In: Katsikas, S.K., Lopez, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC 2006. LNCS, vol. 4176, pp. 530–545. Springer, Heidelberg (2006)
48. Gonzalez, J.J., et al.: Helping Prevent Information Security Risks in the Transition to Integrated Operations. Telektronikk 101(1), 29–37 (2005)

49. Sveen, F.O., et al.: A Dynamic Approach to Vulnerability and Risk Analysis of the Transition to eOperations. In: 24th International System Dynamics Conference, Nijmegen (2006)
50. Cooke, D.L.: A system dynamics analysis of the Westray mine disaster. System Dynamics Review 19(2), 139–166 (2003)
51. Pomey, M.-P., et al.: Paradoxes of French Accreditation. Quality and Safety in Health Care 14, 51–55 (2005)
52. Parkhurst, J., Shaw, B.: Compliance is Not Enough: The Benefits of Advanced Quality Systems Practices. Medical Device & Diagnostic Industry (2004)
53. Chelsom, J.V.: Performance-driven quality. Logistics Information Management 10(6), 253–258 (1997)
54. Karapetrovic, S.: ISO 9000: the system emerging from the vicious circle of compliance. The TQM Magazine 11(2), 111–120 (1999)
55. Caralli, R.A., Wilson, W.R.: The Challenges of Security Management. Networked Systems Survivability Program, SEI. [cited 2007 12th March] (2004)

Intrusion Attack Tactics for the Model Checking of e-Commerce Security Guarantees

Stylianos Basagiannis, Panagiotis Katsaros, and Andrew Pombortsis

Department of Informatics, Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
{basags,katsaros,apombo}@csd.auth.gr

Abstract. In existing security model-checkers the intruder's behavior is defined as a message deducibility rule base governing use of eavesdropped information, with the aim to find out a message that is meant to be secret or to generate messages that impersonate some protocol participant(s). The advent of complex protocols like those used in e-commerce brings to the foreground intrusion attacks that are not always attributed to failures of secrecy or authentication. We introduce an intruder model that provides an open-ended base for the integration of multiple attack tactics. In our model checking approach, protocol correctness is checked by appropriate user-supplied assertions or reachability of invalid end states. Thus, the analyst can express e-commerce security guarantees that are not restricted to the absence of secrecy and the absence of authentication failures. The described intruder model was implemented within the SPIN model-checker and revealed an integrity violation attack on the PayWord micro payment protocol.

Keywords: intrusion attacks, e-commerce protocols, model checking, SPIN.

1 Introduction

Model-checking of cryptographic protocols takes place on a model of a small system running the protocol of interest together with an intruder model that interacts with the protocol. Security flaws are found by an appropriate state exploration approach that discovers if the system can enter an insecure state, that is, whether there is an attack upon the protocol.

The basic assumptions are summarized as follows: (i) *The encryption method used is unbreakable*, (ii) *The intruder can prevent any message from reaching its destination* and (iii) *The intruder can create messages of his own*. As a consequence of the foresaid assumptions, model-checking analyses treat any message sent by a honest user as a message sent to the intruder and any message received by a honest user as a message sent by the intruder. This setting refers to a system that becomes a machine, which is used by the intruder to generate words (messages). The intruder's behavior is defined as a message deducibility rule base governing composition and decomposition of messages, encryption and decryption with known keys, as well as memorization and use of eavesdropped information.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 238–251, 2007.

In section 2 we provide an overview of the most influential model checking approaches. All of them use the general *Dolev and Yao intruder model* [1], but the intruder's goal is restricted in finding out a message that is meant to be secret or in generating messages that impersonate some protocol participant. *Failures of secrecy* or *authentication* reveal a previously unknown attack on the analyzed protocol.

However, *security guarantees cannot always be expressed as absence of secrecy or authentication failure*. A typical case is the well-known family of *replay attacks*, where the intruder aims to playback previously recorded messages in an attempt to sabotage an ongoing protocol session: in [2] the authors show that failures of information exchange timeliness that enable message replays do not always manifest themselves as secrecy failures. Hence, replay attacks are basically analyzed [3] with special-purpose modal logics, like the BAN logic (named after its inventors called Burrows, Abadi and Needham [4]). Another complication is that recent studies ([5]) concluded in that *authentication is a protocol dependent notion and there is not a unique definition of authentication that all secure protocols satisfy*.

Sections 3 and 4 introduce a philosophically different approach in designing the intruder model. *We also adopt the assumptions of the Dolev - Yao intruder model*, but instead of specifying its behavior with a set of rules governing deducibility of messages, *we attempt to combine multiple attack tactics based on a careful analysis of how they proceed*. Attack tactics are formalized and are then combined into a single Dolev - Yao intruder model within the SPIN model-checking environment ([6], [7]). Four types of attack tactics have been implemented so far, namely: (i) Replay and integrity violation attacks, (ii) Type-flaw attacks, (iii) Impersonation attacks and (iv) Parallel session attacks.

Although we cannot claim that our approach covers all possible attack tactics, *we do not exclude known attacks that are not reflected as failures of secrecy or authentication*. The developed Dolev - Yao intruder model constitutes a supplemental model-checking mean, used as *an open-ended base for implementing more specialized attack tactics*. This enables revealing attacks, which cannot be detected by existing security model checkers, like for example attacks that subvert non-repudiation [8], fairness, accountability, abuse-freeness [9] or other *e-commerce security guarantees*.

An interesting aspect is the comparatively smaller state spaces enabling *analyses* that *are not restricted to small systems running the protocol of interest*. This allows application of the proposed intruder model to larger and more complex systems, thus opening new potentialities in revealing for example multi-protocol attacks [10] on cryptographic protocols that are executed in the same environment.

With the described approach we discovered an integrity violation attack on the PayWord micro payment protocol [11]. The obtained results are shown in section 6.

2 Related Work

One of the first systems that used the Dolev - Yao intruder model and the secrecy failure approach was the Interrogator tool [12]. Given a final state in which the intruder knows some word, which should be secret, the Interrogator tries all possible ways of constructing a path by which that state can be reached. If it finds a path, then it has identified a security flaw.

Finite state analysis of cryptographic protocols has been developed in a range of published works, which implement the secrecy or authentication failure approach within specialized security analysis tools like BRUTUS [13] or within general purpose model checkers like Murφ [14] and FDR (Failures Divergence Refinement) [15].

The most detailed description of a Dolev - Yao intruder model is given for the so-called "Lazy Spy" [16]. The "Lazy Spy" was initially expressed [17] in the traces model of Communicating Sequential Processes (CSP)/FDR and was later integrated into Casper [18], a front-end for semi-automated CSP description of security protocols. Casper works based on a custom-made set of rules governing deducibility of messages through encryption and uses a lazy exploration strategy, which examines the subset of intruder states reachable by the protocol rules. NRL Protocol Analyzer [19] is another well-known tool with a similar Dolev - Yao intruder model.

In [20] the authors provide a thorough review of the most important state space analysis contributions until 1999. A more recent contribution for the model checking of secrecy and authentication is the so-called "lazy intruder" [21] for the on-the-fly model checker of the AVISPA security toolset [22]. The "lazy intruder" avoids an explicit enumeration of the possible messages the intruder model can generate, by storing and manipulating constraints about what must be generated. The resulted symbolic representation is evaluated in a demand-driven way and this approach reduces the search tree *without excluding any attacks*.

3 The Intruder Model

We adopt the pessimistic assumption that the intruder has absolute control over the used communication network, as well as the basic Dolev - Yao assumptions mentioned in section 1 regarding his abilities. More precisely, the intruder *eavesdrops* or *intercepts* messages and analyzes them if he possesses the keys required for decryption. Also, the intruder *can generate messages* from his knowledge and can send them to any protocol participant. The new messages are created from already known messages by applying one or more of four (4) basic operations: *encryption*, *decryption*, *concatenation* and *projection*.

Any attempt to enumerate all meaningful messages that the intruder can send will inevitably lead to an enormous branching of the resulting state space. The model checking approaches of section 2 attempt to preserve the generality of the intruder model while applying specialized techniques to overcome the foresaid problem. However, they are only applicable to a small system running the protocol of interest. If no attack is found, there is still an open possibility for an attack upon some larger system (a principle known as *the absence of model-checking completeness* [23]).

We aim in a less general but complementary approach for the generation of new messages based on an open-ended base of predefined attack tactics. The structure and the number of all possible fake messages are restricted by the patterns and the number of initial messages of the available attack tactics. The intruder model can be thought as two concurrent processes, where the first aims to eavesdrop/intercept exchanged messages and the second performs a non-deterministically selected attack tactic against the ongoing protocol session(s) (Figure 1).

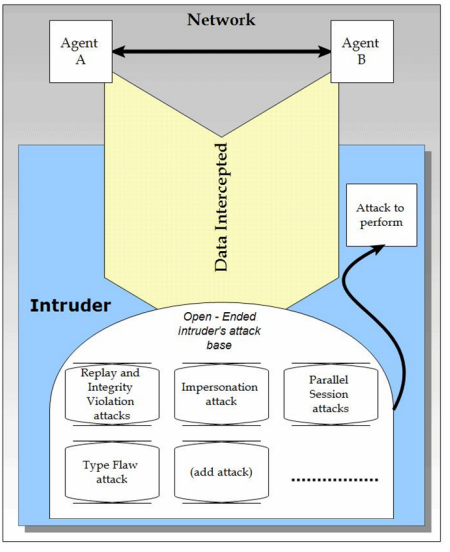


Fig. 1. The intruder process

Upon reception of a *fake message*, by some *victim*, the performed attack step succeeds and the subsequent execution trace is possible to reach an invalid end state or a correctness assertion violation. If the "victim" does not accept the sent fake message, falls into a *fail-stop state*, where he does not continue with the ongoing protocol execution. Protocol correctness, whether it is expressed as reachability of an invalid end state or an assertion check is thus not restricted to secrecy or authentication guarantees.

An *atomic message* may come from one of the sets:

- *Keys*, with members that represent the keys used to encrypt messages, such that every key $k \in Keys$ has an inverse $k^{-1} \in Keys$. For symmetric cryptography the decryption key is the same as the encryption key, i.e. $k = k^{-1}$.
- *Agents*, with members that represent the names of the *honest protocol participants*.
- *Nonces*, which is an infinite set of randomly generated numbers. Members of *Nonces* are used as timestamps that is, any message containing one of them can be assumed having been generated after the nonce itself was generated.
- *Data*, with members that represent the plaintext strings exchanged between the protocol's participants.

We denote by I the *intruder* ($I \notin Agents$). Also, we define the binary relation,

$$is\_key\_of = \{(k, id): k \in Keys, id \in Agents \cup \{I\}, \text{"key } k \text{ is used by the participant } id\text{"}\}$$

such that $|is\_key\_of\ (k)| = 1$ in the case of public key cryptography or $|is\_key\_of\ (k)| = 2$ in the case of symmetric cryptography.

The set *Msgs* of *exchanged messages* is defined inductively over the *disjoint union*

$$AMsgs = Keys \cup Agents \cup \{I\} \cup Nonces \cup Data$$

that represents the set of atomic messages ($Set_i \cap Set_j = \varnothing$ for any two Set_i, Set_j of the unified sets). More precisely:

- If $\alpha \in AMsgs$ then $\alpha \in Msgs$.
- If $msg_x \in Msgs$ and $msg_y \in Msgs$ then $msg_x \cdot msg_y \in Msgs$, where $\cdot$ represents message concatenation.
- If $msg \in Msgs$ and $k \in Keys$ then $\{msg\}_k \in Msgs$.

Each $\mathtt{ag} \in Agents$ may attempt to execute the protocol for a bounded number of times say $\#Ses_{\mathtt{ag}}$ and each such attempt is a separate *protocol session noSes*, such that $1 \le noSes \le \#Ses_{\mathtt{ag}}$. In a protocol session, ag plays either the role of the *initiator* or the *responder*.

We denote by $sent_n^{\mathtt{ag}_{noSes}}$ the finite-length concatenation sequence of messages sent by $\mathtt{ag} \in Agents$ in the course of session *noSes*:

$$sent_n^{\mathtt{ag}_{noSes}} = (sent_{n-1}^{\mathtt{ag}_{noSes}} \cdot msg_n)$$

with the first term equal to the *null sequence* that is, $sent_0^{\mathtt{ag}_{noSes}} = ()$. The sequence $sent_n^{\mathtt{ag}_{noSes}}$ represents participant's ag *history* for session *noSes*, after having sent msg_n. We denote by $rcvd_n^{\mathtt{ag}_{noSes}}$ the finite-length concatenation sequence of messages received by ag in the course of session *noSes*. In a given time instant the acquired *participant's knowledge* for the ongoing protocol execution is given as

$$\mathtt{ag}_{\text{knowledge}} = \bigcup_{\mathtt{ag}_j} \{rcvd_{\max(i)}^{ag_j}\} \cup \mathtt{ag}_{in\_knowledge},$$

for all $1 \le j \le \#Ses_{\mathtt{ag}}$, where $\mathtt{ag}_{in\_knowledge}$ represents the *initial knowledge base* of ag (keys, agent identities and so on) and $i > 0$ represent the terms of the received message concatenation sequences. A protocol session for a honest participant $\mathtt{ag} \in Agents$ is defined formally as a 5-tuple $\langle \mathtt{ag}, j, \mathtt{ag}_{knowledge}, \mathtt{ag}_{history}^{j}, P \rangle$, where $1 \le j \le \#Ses_{\mathtt{ag}}$ and P is a *process description* given as a sequence of *actions* to be performed. We consider the actions **send** and **receive** for sending and receiving messages to/from other protocol's participants.

The assumptions mentioned in section 1 for the general Dolev - Yao intruder model imply that in a given time instant the acquired *intruder's knowledge* for the ongoing protocol execution is given as

$$I_{knowledge} = \bigcup_{\mathtt{ag}_j} \{sent_{\max(i)}^{ag_j}\} \cup I_{in\_knowledge},$$

for all $1 \le j \le \#Ses_{\mathtt{ag}}$, $\mathtt{ag} \in Agents \cup \{I\}$, where $I_{in\_knowledge}$ represents the initial intruder's knowledge base and $i \ge 1$ represent the terms of the eavesdropped message concatenation sequences.

The *protocol model* is given as the asynchronous composition of the models for each protocol session, including the intruder model, whose behavior depends on the

defined *attack tactics*. Attack tactics are non-deterministically selected and are then executed within a single thread of control. Each possible *execution* of the model corresponds to a finite alternating sequence of *global states* and actions:

$$\tau = s_0\, \alpha_1\, s_1\, \alpha_2 \ldots s_n, \text{ for some } n \in N$$

such that $s_{i-1} \xrightarrow{a_i} s_i$ for $0 < i \leq n$ and for the transition relation $\rightarrow$ defined as

$$\rightarrow \subseteq S \times PS \times A \times Msgs \times S$$

where S is the set of global states, PS is the set of protocol sessions and A is the set of action names.

4 Attack Tactics

We formalized and subsequently implemented a series of basic attack tactics. First, we present the elementary tactics that are also used in forming more complex attack scenarios. The implemented attack tactics are the ones that are most often reported in related bibliography.

4.1 Message INterCePTion (INCPT)

Message interception takes place after the occurrence of some action **send** (`ag`, *v*, *msg*)[1], for some `ag`, $v \in Agents$ and some $msg \in Msgs$, if there is no **receive** (*v*, *u*, *msg*)[2] with $u \in \{\texttt{ag}, I\}$ in the suffix execution trace. When intercepting an encrypted message $\{msg\}_k$ there is no **receive** (*v*, *u*, $\{msg\}_k$) action in the suffix execution trace.

4.2 Replay Attack Tactics

Replay attacks take place when the intruder redirects eavesdropped or altered messages within one or more interleaved protocol session(s). We adopt the replay attack classification of [5] and we formalize the following replay attack tactics.

REFlections (R-REF):
In a *reflection attack* the intruder resends an altered version of a previously sent message back to its sender. *Run-internal reflections* are performed within the same protocol session. *Interleaving reflections* use contemporaneous protocol sessions and *classic reflections* use messages obtained from already finished protocol sessions.

The R-REF attack takes place anytime after the occurrence of some action **send** (*v*, `ag`, *msg*), with *msg* representing any non-encrypted $msg \in Msgs$ or after the occurrence of some action **send** (*v*, `ag`, $\{msg\}_k$) such that $I \notin is\_key\_of\,(k) \wedge k^{-1} \in I_{knowledge}$.

The foresaid actions result in a global state where either

$exists(msg,\ sent^{v_j}_{\max(i)})$[3] = `true` or respectively $exists(\{msg\}_k,\ sent^{v_j}_{\max(i)})$ = `true`

[1] The action whereby `ag` sends *msg* to *v*.
[2] The action whereby *v* receives *msg* from ,.
[3] Boolean predicate *exists*(*msg*, *str*) is `true` if the message $msg \in Msgs$ appears in string *str*.

for some $1 \le j \le \#Ses_v$, with $i \ge 1$ representing the terms of an eavesdropped message concatenation sequence.

In the performed reflection attack the intruder alters *msg* based on $I_{knowledge}$ and uses the altered $msg' \in Msgs$ in an action **send** (I, v, msg') or respectively **send** $(I, v, \{msg'\}_{k'})$ for some $k' \in I_{knowledge}$ such that $v \in is\_key\_of(k')$.

The R-REF attack succeeds only when *v* performs the action **receive** (v, I, msg') or respectively the action **receive** $(v, I, \{msg'\}_{k'})$ with the following potential outcomes:

- run-internal reflection

$exists(msg', rcvd^{v_j}_{\max(i)}) = \texttt{true}$ or $exists(\{msg'\}_{k'}, rcvd^{v_j}_{\max(i)}) = \texttt{true}$

- classic or interleaving reflection

$\exists j' \neq j$: $exists(msg', rcvd^{v_{j'}}_{\max(i)}) = \texttt{true}$ or $exists(\{msg'\}_{k'}, rcvd^{v_{j'}}_{\max(i)}) = \texttt{true}$

DEFlections (R-DEF):
In a *deflection attack* the intruder redirects a possibly altered sent message to some participant that is neither the message's recipient nor the sender. *Run-internal deflections* are performed within the same protocol session. *Interleaving deflections* use contemporaneous protocol sessions and *classic reflections* use messages obtained from already finished protocol sessions.

STraight Replays (R-STR):
In a *straight replay attack* the intruder resends a previously sent message to its intended destination. If the eavesdropped message is replaced by an altered version, this attack is also known as *INTegrity Violation attack* (INTV).

Depending on whether this attack is performed within the same session or contemporaneous or non-interleaved sessions, straight replays are also characterized either as run-internal, interleaving or classic replays.

4.3 Type Flaw Attack Tactics (TFLAWS)

A *type flaw attack* arises when the recipient of a message accepts that message as valid, but imposes a different interpretation on the bit sequence than the protocol participant who created it. Type flaw attacks follow the action sequences of the replay attack tactics and may be optionally combined with a message interception (INCPT), in order to prevent reception of intercepted message by its recipient such as to perform a type flaw based message replay.

I triggers a type flaw attack possibly after having altered an eavesdropped $msg \in Msgs$ based on $I_{knowledge}$, thus resulting in some $msg' \in Msgs$. The subsequent action performed by *I* is either **send** (I, v, msg') or **send** $(I, v, \{msg'\}_{k'})$ for some $k' \in I_{knowledge}$ such that $v \in is\_key\_of(k')$.

This attack tactic succeeds if in the global state after the occurrence of the action **receive** (v, I, msg') or respectively **receive** $(v, I, \{msg'\}_{k'})$ there is some atomic message *amsg*, such that

$$exists(amsg, rcvd^{v_j}_{\max(i)}) = \texttt{true}, 1 \le j \le \#Ses_v$$

with $i \geq 1$ representing the terms of $rcvd_n^{v_j}$ and for two sets Set_i and Set_j from the "disjoint" union *Amsgs*,

$$amsg \in Set_i \cap Set_j$$

The described insecure global state expresses the fact that it is possible for an atomic message that was originally intended to have one type (e.g. nonce) to be interpreted as having another type (e.g. key or data). However, this possibility occurs only when both types are represented as bit sequences of the same length, so that when the intruder positions an atomic message in place of a type flawed one, the recipient is fooled into accepting the used atomic message as the one expected according to the owned process description (*P*).

We note that type flaw attacks [24] may not lead to a direct security compromise, since it is possible that the plaintext bit string of the atomic message used by *I* to be unknown to him (the secrecy is still preserved). However, if for example a nonce is used as a key, this is not a good key, because the main concern in generating nonces is to be unique in a protocol session, as opposed to keys that basically have to be non-predictable. Type flaw attacks may result in failures of security properties beyond the typical secrecy and authentication properties, like for example anonymity and non-repudiation [25].

4.4 Simple IMPersonation Attack (IMP)

An *insecure state* (precondition) for the performance of a simple IMP attack is any state where *I* can read the contents of a protocol message sent by some $\mathtt{ag} \in Agents$, who acts as initiator of a new protocol session:

$$\{\exists\, sent_1^{\mathrm{ag}_{noSes}} \in I_{knowledge},\ \mathtt{ag} \in Agents,\ 1 \leq noSes \leq \#Ses_{\mathtt{ag}} :$$
$$\{\, sent_1^{\mathrm{ag}_{noSes}} = msg \text{ for some non-encrypted } msg \in Msgs\}$$
$$\vee\ \{\, sent_1^{\mathrm{ag}_{noSes}} = \{msg\}_k :\ is\_key\_of\,(k) = I \vee (is\_key\_of\,(k) \neq I \wedge k^{-1} \in I_{knowledge})\}\}$$

The IMP attack tactic takes place when the intruder performs the following three subsequent actions against some victim $v \in Agents$, such that $v \notin is\_key\_of\,(k)$ and $v \neq \mathtt{ag}$:

$$\textbf{send}\ (I, v, msg'),\ \textbf{receive}\ (I, v,\ sent_1^{v_{newSes}}),\ \textbf{send}\ (I, \mathtt{ag},\ sent_1^{v_{newSes}})$$

where $msg' = sent_1^{\mathrm{ag}_{noSes}}$, when the latter is a non-encrypted message or otherwise $msg' = \{msg\}_{k'}$, with $k' \in I_{knowledge}$ and $v \in is\_key\_of\,(k')$. Also, v_{newSes} is a unique session identifier for session *newSes*, in which victim *v acts as responder* and the boolean predicate $exists(v,\ sent_1^{v_{newSes}})$ is `false`. If the last mentioned predicate would be `true`, `ag` would realize that the responder in session $\mathtt{ag}_{noSes}$ is not the one selected and would subsequently abort the corrupted protocol session.

4.5 Parallel Session Attack Tactics (PARSES)

Parallel session attacks take place by subsequent interleaving replays among contemporaneous protocol sessions, in which the intruder manipulates protocol participants in multiple roles (initiator or responder), in order to subvert the protocol's goals.

The intruder can under special conditions use the cryptographic protocol dialogs

- as an *oracle* that is, to foretell the contents of otherwise perfectly encrypted messages (refer to the oracle session attack shown in [26]);
- to impersonate a protocol participant (e.g. the BAN-Yahalom attack in [5]);

or possibly to subvert properties beyond secrecy and authentication.

In a parallel session attack the execution sequence τ includes a series of action cycles that open with some action **send** (`ag`, *v*, *msg*) or **send** (`ag`, *v*, $\{msg\}_k$) and this results in

$$exists(msg,\ sent^{ag_j}_{\max(i)}) = \texttt{true} \text{ or respectively } exists(\{msg\}_k,\ sent^{ag_j}_{\max(i)}) = \texttt{true}$$

I either opens a new protocol session v'_{newSes} or responds to an already opened session say v'_m (with $v' \in Agents$ including `ag` and *v*), for which the last action of the process description *P* is not included in the prefix execution sequence of τ. The attack is performed possibly after having altered the eavesdropped $msg \in Msgs$ (based on $I_{knowledge}$), thus resulting in sending some $msg' \in Msgs$ by **send** (*I*, *v'*, *msg'*) or **send** (*I*, *v'*, $\{msg'\}_{k'}$) for some $k' \in I_{knowledge}$ such that $v' \in is\_key\_of\ (k')$. The interleaving replay succeeds if the action cycle ends with a *receive* action by *v'*, yielding a global state such that

$$exists(msg',\ rcvd^{v'_m}_{\max(i)}) = \texttt{true} \text{ or respectively } exists(\{msg'\}_{k'},\ rcvd^{v'_m}_{\max(i)}) = \texttt{true}$$

with max(*i*) = 1, if *m* represents a new protocol session (*newSes*).

A number of successive interleaving replays may end up in a *fail-stop global state* or in either an invalid end state or violation of a *protocol correctness assertion*. The latter possibilities reveal a previously unknown parallel session attack.

5 The PayWord Micro Payment Protocol

We focus on the analysis of the PayWord micro-payment protocol that was first proposed by Rivest and Shamir in [11]. PayWord is a credit based off-line protocol implemented by the use of *hash chains* that are called *chains of PayWords*. In our work we will assume the use of the MD5 hash function [27] denoted by `w(i)`. Three participants are involved in a protocol session: the *Customer*, the *Broker* and the *Vendor*. The Customer (*C*) establishes an account with the Broker (*B*) who issues a certificate containing customer's information and *B*'s name. This certificate will authorize *C* to construct PayWord chains validating himself to some Vendor (*V*). The basic steps of PayWord micro-payments are shown in Figure 2.

Upon reception of the foresaid certificate (`certC`), *C* computes the PayWord chain *w* in reverse order based on a randomly chosen term. Then, he signs the

so-called commitment (*M*) of the PayWord protocol which consists of the calculated first term of the chain (`w(0)`) along with the required customer information; *M* is sent to *V*. In every single payment, a chain term of type, `P:(w(i),i)` is sent to *V* until the last payment, `P:(w(I),I)`. We consider the (attacked) variable-size payment scenario, where the value of each payment varies between `1` and `n`. *V* verifies the payments `P`, by applying the hash function `w` to the last valid payment *v* times, where *v* is the value of the requested payment (`w(i-v)`). At the end of the day, *V* reports to *B* the last (highest-indexed) payment (`w(I),I`) - where `I=max(i)` - received from *C* within the current day, together with the owned *C*'s commitment.

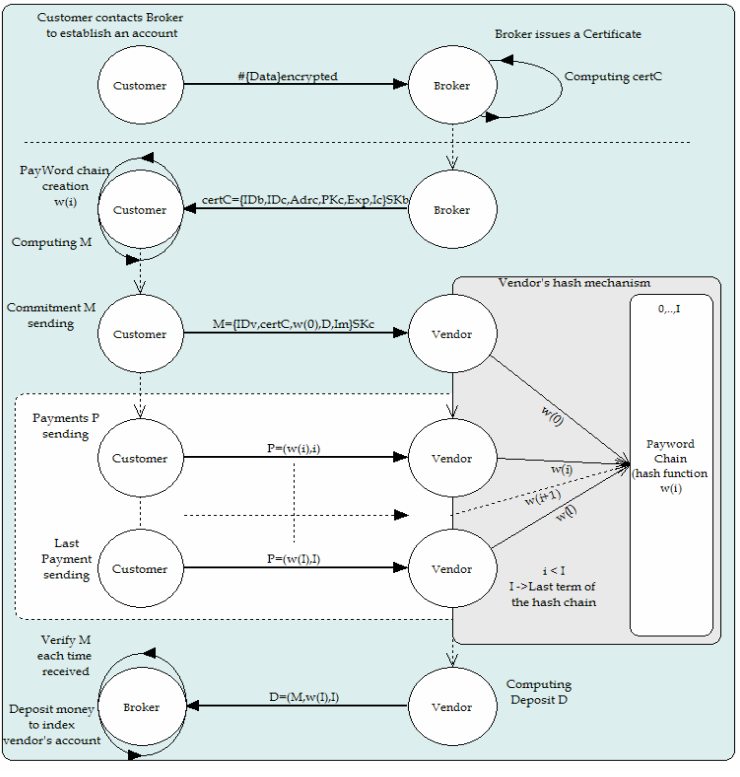


Fig. 2. The PayWord micro payment protocol

Table 1. Glossary of the PayWord protocol notation

IDc	Customer ID	Addrc	Customer address
IDb	Broker ID	certC	Customer certificate
IDv	Vendor ID	Exp	Certificate expiration time stamp
SKb	Broker's key		
PKc	Customer's public key	Ic	Customer's information
SKc	Customer's secret key	Im	Vendor's information
		D	Date

While the use of the hash chain ensures reduced computational requirements for *V*, the attack found on the protocol is based on *V*'s mechanism, when accepting an altered "hashed" message. Provided the intruder's ability to perform hash function calculations by MD5, the detected attack takes place when the intruder intercepts and alters a variable-size payment request.

6 Verification Results

This section provides simulation and verification results obtained within the SPIN model checking environment for the developed PayWord model, when combined with the described intruder model. The simulation output is shown by the automatically generated Message Sequence Chart.

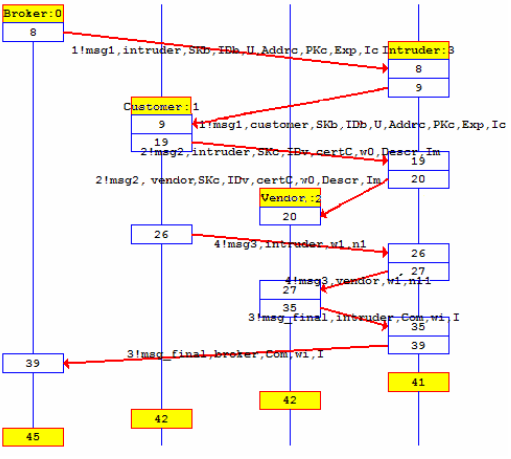


Fig. 3. INTV attack of a variable-size payment (P): *V* accepts an altered message

```
pan: invalid end state (at depth 26)
pan: wrote pan_in.trail
(Spin Version 4.2.6 -- 27 October 2005)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim         - (not selected)
        assertion violations - (disabled by -A flag)
        cycle checks        - (disabled by -DSAFETY)
        invalid end states  +

State-vector 112 byte, depth reached 27, errors: 1
    23 states, stored
     0 states, matched
    23 transitions (= stored+matched)
     0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.003   equivalent memory usage for states (stored*(State-vector + overhead))
0.293   actual memory usage for states (unsuccessful compression: 10603.04%)
        State-vector as stored = 12716 byte + 8 byte overhead
2.097   memory used for hash table (-w19)
0.320   memory used for DFS stack (-m10000)
0.144   other (proc and chan stacks)
0.089   memory lost to fragmentation
2.622   total actual memory usage
```

Fig. 4. Verification output

Figure 3 shows the detected INTV attack. In state 19 *C* sends a commitment (*M*), which is not affected by the intruder and continues with the first variable-size payment attempt (*P*). In state 27 the intruder alters message `(w1,n1)` thus resulting in the fake message `(w1',n1-1)`, which is eventually accepted by *V*. Finally, *V* dispatches message *D* (deposit) and the protocol session ends with a successful INTV attack that is encoded as an invalid end state.

Figure 4 shows the obtained verification output that revealed the described attack scenario. The performed state space search reports an error and generates a counter-example reflecting a feasible path to the defined invalid end state. By the use of the error trail simulation feature of SPIN we roll back the protocol execution and identify the detected flaw.

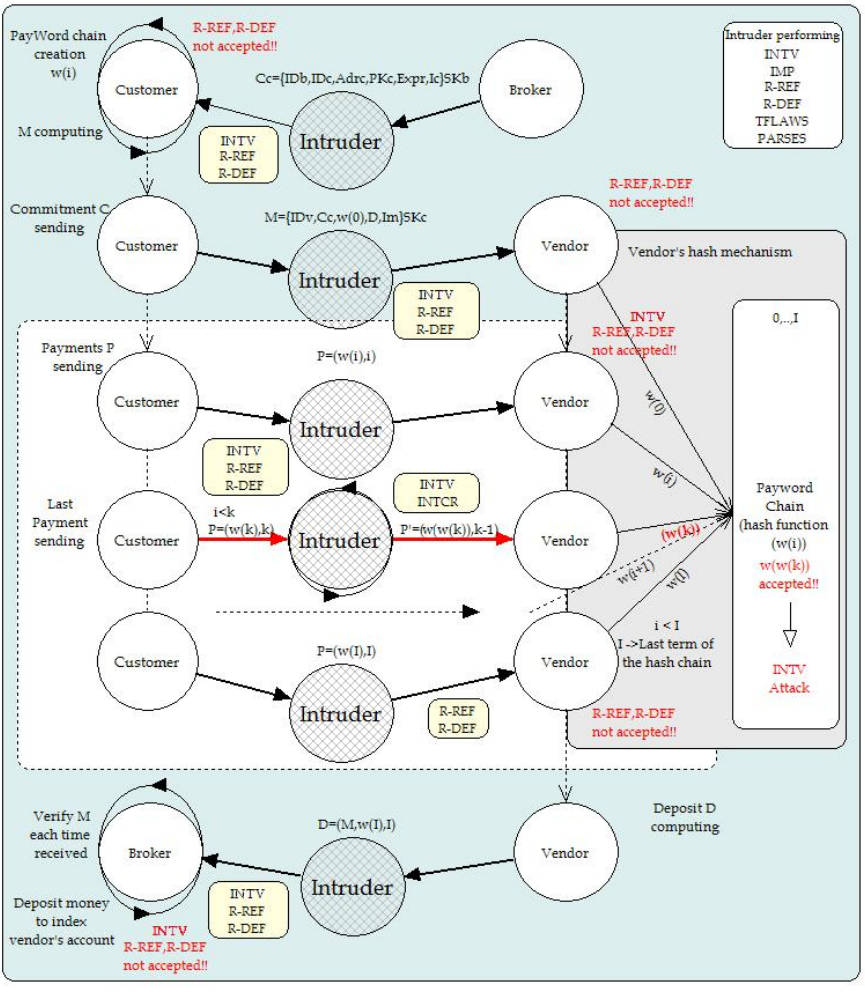


Fig. 5. Attack tactics on the PayWord micro payment protocol

Figure 5 summarizes the attack tactics attempted by the described intruder model and the participants' responses in all protocol steps. Failed attack scenarios are noted as "not accepted!!" and result in fail-stop model states. The detected INTV attack is shown within the frame in the right-hand side that represents *V*'s hash mechanism.

7 Conclusion

This work introduces an open-ended Dolev - Yao intruder model that combines elementary and more complex attack tactics in an attempt to subvert security protocol guarantees. We provided a formalized description of the most often reported attack tactics, which were implemented within the SPIN model-checking environment. The obtained intruder model was applied to a range of electronic payment protocols and revealed an integrity violation attack on the PayWord micro-payment protocol.

Although the proposed model is bound to the absence of model checking completeness - as all published approaches - it constitutes a supplemental model-checking mean, capable to reveal violations of protocol correctness properties, beyond those checked by existing security model checkers.

Finally, the proposed intruder model is open to extensions aiming to integrate more specialized attack tactics that may subvert e-commerce security guarantees like non-repudiation, fairness, accountability, abuse-freeness and so on.

References

1. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Transactions on Information Theory 2/29, 198–208 (1983)
2. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: Proc. of the IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos (1993)
3. Meadows, C.A.: Formal verification of cryptographic protocols: A survey. In: Safavi-Naini, R., Pieprzyk, J.P. (eds.) ASIACRYPT 1994. LNCS, vol. 917, pp. 133–150. Springer, Heidelberg (1995)
4. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Transaction on Computer Systems 8/1, 18–36 (1990)
5. Syverson, P., Cervesato, I.: The logic of authentication protocols. In: Focardi, R., Gorrieri, R. (eds.) Foundations of Security Analysis and Design. LNCS, vol. 2171, pp. 63–137. Springer, Heidelberg (2001)
6. The SPIN model checker official website, available at http://spinroot.com/
7. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1991)
8. Kremer, S., Markowitch, O., Zhou, J.: An intensive survey of fair non-repudiation protocols. Computer Communications 25/17, 1606–1621 (2002)
9. Shmatikov, V., Mitchell, J.C.: Finite-state analysis of two contract signing protocols. Theoretical Computer Science 283, 419–450 (2002)
10. Cremers, C.J.F.: Feasibility of multi-protocol attacks. In: Proc. of the First International Conference on Availability, Reliability and Security, IEEE Computer Society Press, Los Alamitos (2006)

11. Rivest, R.L., Shamir, A.: Payword and Micromint: Two simple micropayment schemes. In: Lomas, M. (ed.) Security Protocols. LNCS, vol. 1189, pp. 69–87. Springer, Heidelberg (1997)
12. Millen, J.K., Clark, S.C., Freedman, S.B.: The Interrogator: Protocol Security Analysis. IEEE Transactions on Software Engineering 13/2 (1987)
13. Clarke, E.M., Jha, S., Marrero, W.: Verifying security protocols with Brutus. ACM Transactions on Software Engineering and Methodology 9/4, 443–487 (2000)
14. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using Murφ. In: Proc. of the IEEE Symposium on Research in Security and Privacy, pp. 141–153. IEEE Computer Society, Los Alamitos (1997)
15. Roscoe, A.W.: Modeling and verifying key-exchange protocols using CSP and FDR. In: Proc. of the 8th IEEE Computer Security Foundations Workshop, pp. 98–107. IEEE Computer Society, Los Alamitos (1995)
16. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall, Englewood Cliffs (1997)
17. Roscoe, A.W., Goldsmith, M.: The perfect spy for model-checking cryptoprotocols. In: Proc. of the 1997 DIMACS Workshop on Design and Formal Verification of Security Protocols (1997)
18. Lowe, G.: Casper: a compiler for the analysis of security protocols. In: Proc. of the IEEE Computer Security Foundations Workshop, pp. 18–30. IEEE Computer Society, Los Alamitos (1997)
19. Meadows, C., Kemmerer, R., Millen, J.: Three systems for cryptographic protocol analysis. Journal of Cryptology 7/2, 79–130 (1994)
20. Gritzalis, S., Spinellis, D., Georgiadis, P.: Security protocols over open networks and distributed systems: formal methods for their analysis, design, and verification. Computer Communications 22, 697–709 (1999)
21. Basin, D., Modersheim, S., Vigano, L.: OFMC: A Symbolic Model-Checker for Security Protocols. International Journal of Information Security (2004)
22. AVISPA: Automated validation of internet security protocols and applications, FET Open Project IST-2001-39252 (2003), http://www.avispa-project.org
23. Lowe, G.: Towards a completeness result for model-checking of Security Protocols. In: Proc. of the 11th Computer Security Foundations Workshop, IEEE Computer Society Press, Los Alamitos (1998)
24. Clark, J., Jacob, J.: A survey of authentication protocol literature: version 1.0, Technical Report, University of York (1997)
25. Heather, J., Lowe, G., Schneider, S.: How to prevent type flaw attacks on security protocols. In: Proc. of the 13th IEEE Computer Security Foundations Workshop, pp. 255–268. IEEE Computer Society, Los Alamitos (2000)
26. Carlsen, U.: Cryptographic protocol flaws – Know your enemy. In: Proc. of the 7th IEEE Computer Security Foundations Workshop, pp. 192–200. IEEE Computer Society, Los Alamitos (1994)
27. Rivest, R.L.: The MD5 Message-Digest Algorithm. In: Internet informational RFC 1321 (1992)

Safety Process Improvement with POSE and Alloy

Derek Mannering[1], Jon G. Hall[2], and Lucia Rapanotti[2]

[1] General Dynamics UK Limited
[2] Centre for Research in Computing, The Open University

Abstract. Safety Standards demand that applications demonstrate they have the required safety integrity, starting with the initial requirements phase. This paper shows how the Problem Oriented Software Engineering (POSE) framework, in conjunction with the Alloy formal method, supports this task through its ability to elaborate, transform and analyse the project requirements. The results of applying this combination to an existing design showed that process improvement can be realised through its ability to detect anomalies early in the life cycle that had previously been detected by much later (and more costly) validation work.

1 Introduction

The first author is a member of a development group which has a successful safety critical design process based on the use of the Z notation [1]. Of critical importance to the process is the proof of conformance of the Z design specification against the formal Z safety properties. Currently, validation occurs well into the design process which, if anomalies with the Z safety properties are uncovered, greatly increases the impact of reworking. One of the reasons for such a late validation is that the current process does not have tool support, but relies on manual proof, which is expensive. In this paper we illustrate, through a case study, how front-end support for safety analyses of early requirements models was added to the process. The candidate front end is based on the combination of the Problem Oriented Software Engineering (POSE) safety pattern proposed in [2] and the Alloy formal method [3]: POSE provides for early requirements models which are amenable to safety analysis, and Alloy for lightweight tool support for animation and proof based on such models and is compatible with Z. The case study was chosen partly because of the existence of known anomalies, ranging from modelling errors through to contradictory requirements, so that the ability of the front end to detect anomalies early could be tested. Indeed, these (known) anomalies were found early in the process; in addition from an analysis of the case study results we conjecture that, given a reasonable process, the front end could have found certain classes of anomaly *ab initio*, and without prior knowledge. Another benefit of the front end is that it is *efficient*, i.e., it uses the same information and models as used for the development task, and so any overhead in validating specific safety analysis models can be avoided.

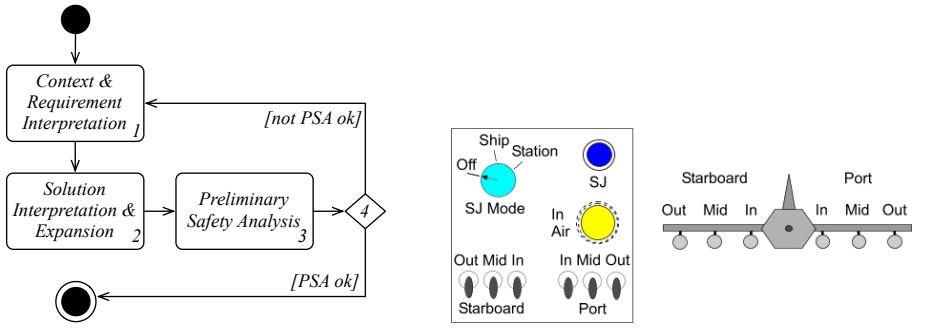
The paper is organised as follows. Brief descriptions of the background to the paper are given in Section 2. Section 3 applies the POSE and Alloy combination

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 252–257, 2007.

on a case study involving the development of a Stores Management System. Section 4 includes related work. Section 5 contains a discussion and conclusions.

2 Problem Oriented Software Engineering

In POSE, software development is viewed as solving a problem, the solution being a machine—that is, a program running in a computer—that will ensure satisfaction of the requirement in the given problem world consisting of real-world domains. POSE is defined as a formal Gentzen-style sequent calculus [4] that allows problems to be transformed into problems that are easier to solve, or that will lead to other problems that are easier to solve. A set of transformation rule schema defined in the calculus capture (atomic) discrete steps in development. Each requires a justification of application in order for the transformation to be solution preserving—simplifying only slightly, this means that a solution to a transformed problem is also a solution to the original problem—although justifications need not be formal. The combination of the justifications is an argument that the solution is adequate as a solution to the original problem. The interested reader is referred to [5,6] for a complete presentation of POSE.



(a) POSE Safety Pattern

(b) Selective Jettison Panel

Fig. 1.

Previous work [2] has identified a re-usable process template or 'pattern' for safety-critical development, shown in Figure 1a as a UML activity diagram. The activities in the figure include the following POSE activities: (a) *Context & Requirement Interpretation*, used to capture increasing knowledge and detail in the context (i.e., the environment) and requirement of the problem; (b) *Solution Interpretation & Expansion*, through which an architecture (logical and/or physical) for the solution is chosen, and used to transform the problem; (c) *Preliminary Safety Analysis (PSA)*, a combination of problem simplification and traditional safety analysis conducted to ensure a feasible solution structure has been chosen. The choice point (labelled 4) uses the outcome of the PSA to determine whether: (a) the current architecture is viable as the basis of a solution;

or (b) whether backtracking and (re-)development of the problem (activity 1) and/or another candidate architecture (activity 2) should be chosen. The process pattern was applied to the case study discussed in this paper in order to support the early detection of anomalies.

3 Case Study

The case study concerns the development of the Stores Management System (SMS) in use on a real aircraft. For space reasons only part of the case study is reported in this paper, namely the design of the selective jettison (SJ) functionality, i.e., the way in which stores are chosen for jettison from the aircraft. Although only one part of the system, this is sufficiently illustrative of the application of the POSE/Alloy combination to the problem of early detection of anomalies. The Selective Jettison Panel (SP) shown in Figure 1b indicates the aircraft has six release stations—corresponding to Outer, Middle and Inner positions on each of the Starboard and Port wings. The pilot (P) controls the store jettison via the SP, using: (a) six selection switches, one for each station; (b) a three position SJ mode selection switch; (c) an 'In Air' indicator lamp; and (d) the SJ button which initiates the jettison. The SJ sequence does not release all the stores at once, as this could result in a collision hazard, rather the step release sequence used is `Out` → `Mid` → `In` (see Figure 1b) with a delay between each step: the SJ is not an atomic action, but rather three atomic actions in sequence. A balance algorithm is applied to most jettison sequences to ensure that the aircraft is not put into an unsafe state. The exception is that a single store jettison is always allowed to give the pilot final control under error conditions.

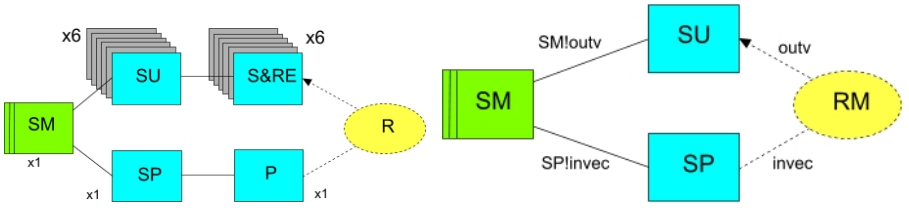
3.1 Applying the POSE Safety Pattern

The first two steps in the POSE safety pattern (Figure 1a) led us to the development problem sketched in Figure 2(a)[1]. Each station (there are six of them) has a Store Unit (SU) and a Suspension and Release Equipment (S&RE) associated with it. The SU provides the power to release or jettison the store held on the S&RE, under the control of the Safety Manager (SM) to be designed. R represents both functional and safety requirements at the system boundary.

The third step of the POSE pattern is to perform a PSA. As described above, this step combines problem simplification and traditional safety analysis. Problem simplification removes the non-direct domains (6x S&RE and P[2]) and transforms requirements R at the system boundary to requirements RM that apply directly to the required solution machine SM. The mechanism is explained in detail in [2], but is not covered in depth here. The idea of reformulating requirements such as RM which apply directly to the required solution machine (SM in this case) is that it helps in the specification of the solution machine. However, before producing

[1] Instead of the formal notation of POSE, which would take too long to explain, here we use a graphical notation derived from that of Problem Frames [7].

[2] The system safety analysis and assumptions associated with the simplification [6] cover the pilot impact on safety.



(a)SMS Architecture and (b) POSE Transformed SMS Architecture

Fig. 2.

such a specification, it is important to establish that the requirements are feasible from a safety perspective; otherwise, the development must backtrack. The Alloy model for the formal requirements specification of the SM was developed directly from the POSE model as depicted in Figure 2(b). The functionality of this Alloy model was based on the POSE model information, the Z specifications, and the derived requirements, RM. There is not space to provide the Alloy specification, but it is based on modifying the trace models of Chapter 6 of [3].

3.2 The PSA Safety Analysis

The formal Alloy model was successfully validated against the requirements RM using a variety of simulation runs with different combinations of inputs covering the range of input values. After this, the PSA safety analysis was undertaken, which followed the pattern used on earlier examples (e.g. [2]), with Functional Failure Analysis (FFA) [8] being applied to identify any system issues. An important FFA issue was 'Multiple station SJ release but balance not applied'. The Alloy model was simulated for a number of jettison package selections involving single and multiple stores. The results should have been that single store jettisons are always allowed, whilst multi-store jettisons must satisfy the balance algorithm. However, a package consisting of the release sequence `p1;p0;p1`, where a single store (`p1`) is released in the `Out` and `In` steps and no release (`p0`) in the `Mid` step, resulted in no balance being applied. Further investigation established that balance was only applied when two stores were being released in the same step (`p2`), and only for that step. The problem occurs in the high level Z specification of the SM, which contains the term $\#releaseLocations > 1 = \theta SM \in balanced$, where # represents the cardinality of the set and θSM represents the bound values of SM. Now *releaseLocations* is the number of jettison pulses being applied (the `p0`, `p1` and `p2`)—this is modelled in Alloy by a function `balance()` which specifies no balance only if there are zero (`p0`) or a single (`p1`) pulse. The problem is that (as noted above) SJ is not an atomic action, but rather three: one each for `Out`, `Mid` and `In`. The functionality given above ensures each atomic action satisfies the balance, but does not ensure the entire SJ sequence does. Hence the pulse sequence `p1;p0;p1` meets the criteria for each of its atomic components, indicating that the model does not require it to be balanced, when it should be. The solution is to base the balance calculation on the SJ station selection and not on the jettison

pulses. When this change was made, the model behaved as required. Other FFA issues were addressed in a similar fashion, and the modelling identified the further known anomaly and a not previously identified one. The latter concerned an inconsistency between two of the Customer supplied requirements documents.

4 Related Work

The case study is based on multi-level safety analyses process typical of many industries; one example is ARP4761 [8] which governs commercial airborne systems. This paper uses Preliminary Safety Analysis, which corresponds to the Preliminary System Safety Assessment (PSSA) phase of ARP4761.

Requirements in POSE follow the fundamental clarification work of Jackson [9] and Parnas [10] which distinguishes between the given domain properties of the environment and the desired behaviour covered by the requirements. This work also distinguishes between requirements that are presented in terms of the stakeholder(s) and the specification of the solution which is formulated in terms of objects manipulated by software [11]. Therefore there is a large semantic gap between the system level requirements and the specification of the machine solution. POSE bridges this gap by transforming the system level requirements so they apply more directly to the solution.

The POSE notion of problem fits well with the Parnas 4-Variable model [10] which is well suited to defining embedded critical applications. The 4-Variable model also forms the basis of the SCR [12] and SpecTRM [13] methods, both of which have toolsets for the development of safety systems. Recent work indicates that POSE will interface to these methods. AMBERS [14], also uses the model (SCR variant) and has similar goals to and is also compatible with POSE, but it does not include the PSA feasibility check.

5 Discussion and Conclusions

The case study results illustrate that application of the POSE safety pattern allows the safety feasibility of a system's requirements to be analysed early in the development life cycle. Earlier papers ([15],[6],[2]) have shown that POSE is flexible enough to work well with a variety of common development approaches. The results of this case study further confirm this finding by showing that the POSE/Alloy combination works well and is suitable for supporting the front end of a safety process. As in the earlier case work [2], the analysis was found to be quick and efficient since there is no need to validate special models produced just for the safety analysis.

The task of using the POSE/Alloy combination to improve the front end of the safety critical process asked two major questions. The first, and easiest to address, was: "Could the POSE/Alloy combination detect the anomalies?" The results demonstrate that both POSE and Alloy can be used in combination to detect the anomalies of interest. The second question is: "Would the POSE/Alloy combination have detected these anomalies by following a reasonably expected process?

This is more difficult, but a plausible answer is that in the case study POSE and Alloy were used as recommended and found the anomalies of interest. This imparts confidence that a process using POSE/Alloy would be able to detect these and similar anomalies at an early phase in the development life cycle. Therefore technically the POSE/Alloy combination could and would be expected to provide the required process improvement for the safety critical process under study. The next step is to evaluate the process improvement on a project using normal, but suitably trained, project engineers rather than those specialising in the POSE and Alloy techniques, to see if similarly encouraging results are obtained.

References

1. Spivey, J.M.: The Z-Notation - A Reference Manual, 2nd edn. Prentice-Hall, Englewood Cliffs (1992)
2. Mannering, D., Hall, J.G., Rapanotti, L.: Towards normal design for safety-critical systems. In: Proceedings of FASE 2007, pp. 398–411 (2007)
3. Jackson, D.: Software Abstractions Logic, Language & Analysis. MIT Press, Cambridge (2006)
4. Kleene, S.: Introduction to Metamathematics. Van Nostrand, Princeton (1964)
5. Hall, J.G., Rapanotti, L., Jackson, M.: Problem oriented software engineering: Formality and the real world. In: Proceedings of IEEE Software Engineering Formal Methods (SEFM 2007), London, UK, IEEE Computer Society Press, Los Alamitos (to appear, 2007)
6. Hall, J.G., Rapanotti, L., Jackson, M.A.: Problem oriented software engineering. Technical Report 2006/10, Open University, Dept. of Computing (2006)
7. Jackson, M.A.: Problem frames: analysing and structuring software development problems. Addison-Wesley, Harlow (2001)
8. SAE: ARP4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. Technical report (1996)
9. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology VI(1), 1–30 (1997)
10. Courtois, P.J., Parnas, D.L.: Documentation for safety critical software. In: 15th International Conference on Software Engineering, Baltimore, USA, pp. 315–323 (1997)
11. van Lamsweerde, A.: Requirements engineering in the year 00: A research perspective. In: 22nd ICSE'00, Limerick (2000)
12. Heitmeyer, C., Jeffords, R.: Applying a formal requirements method to three NASA systems: Lessons learned. In: IEEE Aerospace Conference, Big Sky, MT, IEEE Computer Society Press, Los Alamitos (2007)
13. Leveson, N.G.: Completeness in formal specification language design for process-control systems. In: Proceedings of the third workshop on Formal methods in software practice 2000, Portland, Oregon, ACM Press, New York (2000)
14. da Cruz, M., Raistrick, P.: Ambers: Improving requirements specification through assertive models and scade/doors integration. In: Proceedings of the Safety-critical Systems Symposium, Bristol, UK, Springer, Heidelberg (2007)
15. Rapanotti, L., Hall, J.G., Jackson, M.A.: Problem transformations in solving the package router control problem. Technical Report 2006/07, Computing Department, The Open University (2006)

Defense-in-Depth and Diverse Qualification of Safety-Critical Software

Horst Miedl[1], Jang-Soo Lee[2], Arndt Lindner[1], Ernst Hoffman[1], Josef Martz[1], Young-Jun Lee[2], Jong-Gyun Choi[2], Jang-Yeol Kim[2], Kyoung-Ho Cha[2], Se-Woo Cheon[2], Cheol-Kwon Lee[2], Gee-Yong Park[2], and Kee-Choon Kwon[2]

[1] Institut fuer Sicherheitstechnologie, Postfach 12 13,
85748 Garching, Germany
{Horst.Miedl,Arndt.Lindner,Ernst-Walter.Hoffmann,
Josef.Maertz}@istec.grs.de
[2] KAERI: Korea Atomic Energy Research Institute,
Daejeon, Korea
{jslee,jylee426,choijg,jykim,khcha,swcheon,cklee1,gypark,
kckwon}@kaeri.re.kr

Abstract. In the Korea Nuclear instrumentation and control (I&C) System (KNICS) project, a digital safety system including Reactor Protection System (RPS) and Engineered Safety Features-Component Control System (ESF-CCS) is developed. It is based on a safety grade Programmable Logic Controller (PLC) as a platform for the safety critical I&C systems. The software used in the digital safety system is classified as safety-critical, and it is qualified according to an appropriate lifecycle. This lifecycle includes design and qualification activities for the components and the system. In the KNICS project a defense-in-depth organizational structure for qualification is implemented based on different teams including 3rd party review teams. Each team uses diverse techniques, methods, and tools for their qualification tasks. Using the Korean KNICS project as an example, the results of Defence-in-Depth and Diversity (D3) qualification of safety-critical software are presented.

1 Introduction

Software is one of the main sources of Common Cause Failures (CCF) in digital Instrumentation and Control (I&C) systems for nuclear power plants. This paper describes a strategy to cope with CCF by Defense-in-Depth and Diversity (D3) in the qualification process. In the KNICS project a defense-in-depth and diverse organizational structure for qualification is implemented.

The objectives of the D3 qualification are mainly to ensure high quality in the development process of the programmable logic controllers (PLC) for the safety-critical I&C systems. Those PLCs are then applied to develop a prototype of a safety-critical software based digital protection system for nuclear power plants.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 258–263, 2007.

2 D3 Approaches

Following the terminology of the International Atomic Energy Agency (IAEA) Defense-in-Depth is defined as "*The application of more than one protective measure for a given safety objective, such that the objective is achieved even if one of the protective measures fails.*" This definition is similar to the definition of Diversity: "*The presence of two or more redundant systems or components to perform an identified function, where the different systems or components have different attributes so as to reduce the possibility of common cause failure, including common mode failure.*" [1].

The principles of D3 approaches for I&C systems are given in several publications [2], [3], [4]. Additionally, projects in this field are under way like the IEC standard IEC 62340 "Requirements to cope with Common Cause Failure (CCF)" [5] and the IAEA TECDOC project "Avoiding Common-Cause Failures in Digital I&C Systems of NPPs" [6].

In the KNICS project the principles of DiD and Diversity (D3) have been applied to both, the system architecture and the development and qualification processes. The following gives a brief overview about the main issues.

3 The KNICS Project

The KNICS project is a research and development project starting in 2001 to develop a safety I&C system for nuclear power plants. About 25 Korean organizations (industrial organizations, research institutes, universities and the Korean Institute for Nuclear Safety (KINS)) participate in the project. The project will be finished in 2008. Besides the Korean participants organizations from outside Korea have been involved to diversify the development and qualification processes.

As shown in Fig. 1, the plant protection system (PPS) in the KNICS project is designed in a PLC-based architecture with four redundant channels/divisions (A, B, C, and D). The PPS in KNICS design consists of the reactor protection system (RPS) and the engineered safety feature – component control system (ESF-CCS). In order to avoid CCF in the four channels, different kinds of diversity (e.g. functional diversity, signal diversity, design diversity, equipment diversity, software diversity), are applied to a greater or lesser extend.

To achieve the objective of human diversity, different design and qualification organizations designed and qualified independently the PLC platform, the RPS, and the ESF-CCS as shown in Table 1.

Through the lifecycle used for the PLC platform including the RPS, and the ESF-CCS different design technologies were applied as shown in Table 2.

To achieve the objective of equipment diversity, different CPUs and memory manufacturer have been chosen. Software diversity is achieved by different operating systems, algorithms, compilers/languages and the different communication protocols.

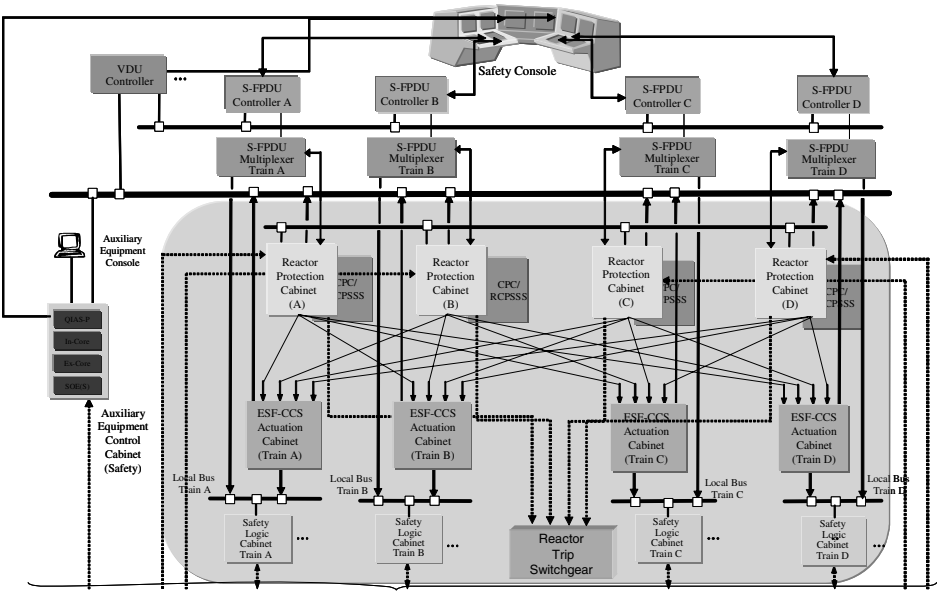


Fig. 1. KNICS Plant Protection System - Overview

Table 1. Application of human diversity

		Design organization	Programmer, Internal Tester	Management teams (QA, CM)	External Testers, Certifiers
1	RPS	DOOSAN, Enesys	DOOSAN, Enesys	KAERI	KAERI
2	ESF-CCS	DOOSAN, BNF	DOOSAN, BNF	KAERI	KAERI
3	POSAFE-Q PLC platform				
3.1	Operating system (pCOS)	POSCON	POSCON, KAERI	KAERI	KAERI, ISTec
3.2	Communication system (HR-SDL, HR-SDN, FMS)	POSCON, CREVIS	POSCON, CREVIS	KAERI	KAERI, ISTec
3.3	I/O modules (AD/DA, DI/DO)	POSCON, CREVIS	POSCON, CREVIS	KAERI	KAERI

To achieve diversity of design criteria, different sets of standards and regulatory criteria have been applied for PLC platform, the RPS and the ESF-CCS. Both, IAEA and IEC guidelines and standards as well as Nuclear Regulatory Commission (NRC) and IEEE guidelines and standards have been used for development and qualification of the POSAFE-Q PLC. In the KNICS project, IEEE 7-4.3.2-2003 [7], NUREG-6303 [8], NRC Branch Technical Position BTP19 [4], draft of IAEA TECDOC on avoiding common-cause failures in digital I&C systems of NPPs [6], drafts of IEC 62340 [5], and IEC 60880 [9] have been referenced to as D3 criteria.

Table 2. Application of design diversity through the lifecycle

		Software requirements	Software design	Implementation	Testing
		Models	Models	Compiler / Language	Testing / Analysis
1	RPS	NuSEE (Nu-SRS)	Nu-SDS	pSET	Cantata++/ LDRA Testbed
2	ESF-CCS	SCADE (DFD, CFD, SSM)	FBD	pSET	Cantata++/ LDRA Testbed
3	POSAFE-Q PLC platform				
3.1	Operating system (pCOS)	Statechart	Structure chart, Modulechart, Flowchart	TI Code Composer C	Cantata++/ LDRA Testbed
3.2	Communication system (HR-SDL, HR-SDN, FMS)	SDL	Flowchart	TI Code Composer C, Paradigm C++	Cantata++/ LDRA Testbed
3.3	I/O modules (AD/DA, DI/DO)	DFD	Structure chart	Keil/C	Tessy/McCabe /ModelSim

4 Application to D3 Qualification in KNICS Project

The qualification of the safety-critical software for the KNICS project has been conducted by a defense-in-depth organizational structure, that consists of the development team, the performance and availability analysis team, the internal testing team, the independent testing team, the independent verification and validation team, the dependability (safety, security, and reliability) analysis team, the quality assurance team, the configuration management team, and the 3rd party review team as a safety envelope. Each team has used diverse techniques, methods, and tools for their tasks.

KAERI analysis and V&V

The software qualification approach applied by KAERI includes dependability analysis (safety and security), verification and validation (V&V) tasks, independent tests, quality assurance and configuration control activities.

To assure high quality of the software, V&V tasks are performed through out all the phases of the life cycle. The V&V activities for the software requirement specifications and for the software design specifications consist of review of the licensing suitability, Fagan inspection, traceability analysis, and fault detection with formal verification methods to avoid subsequent errors. The V&V processes for the code comprise traceability analysis, source code inspection, test case and test procedure generation. Different teams tested the software with diverse tools and methods through the life cycle. Testing is the main V&V action for software integration and system integration phases to detect faults [7].

Failure mode and effect analysis (FMEA) was used to analyze the behavior of the system in cases of failures. Hazard and operability (HAZOP) method was applied to software requirements, design, and implementation analysis. Fault tree analysis (FTA) method was used for the safety analysis of the RPS and the analysis of the final code.

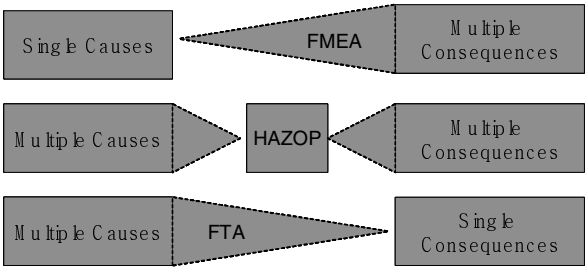


Fig. 2. Cause-consequence coverage of the diverse methods

Due to the application of these three methods, cause-consequence coverage was achieved in the safety analysis as shown in Fig 2 [10].

The software configuration management (SCM) activities comprise configuration identification, configuration control, status accounting, audits and reviews. Configuration audits were performed to verify compliance to specified requirements, standards and contract conditions. They also verify the identification of manuals and traceability of modification requests. The audits are part of the process whereby baselines are accepted or rejected. All SCM, V&V, design and testing activities are controlled by the quality assurance team.

3rd party review

The 3rd party review was performed as software type test in compliance with the procedures developed and applied by ISTec for assessment of safety critical software [11]. The assessment work was restricted to the software of the real time operating system and the safety communication modules. It was performed parallel with the software components' development process. It was not issue of the qualification to assess certain application tasks, i.e. the code of I&C functions.

For practical reasons and to enhance effectivity of the assessment process the work was done in close collaboration between ISTec and KAERI. After each development step the documentation was circulated for assessment to ISTec by KAERI. A subset of documents was evaluated by KAERI itself. In these cases ISTec checked the results of the evaluation (i.e. the verification reports). Additionally, spot checks of the development documents were performed.

All documents were evaluated with respect to ISTec's assessment procedure for safety critical software [11], [12]. The results of document evaluation were discussed at project meetings in detail. During the project meetings refinements of the qualification process in both involved organizations were discussed and improvements were implemented.

5 Conclusions

The paper presents the strategy to cope with CCF by Defense-in-Depth and Diversity (D3) in the qualification process of the Korea Nuclear I&C System (KNICS). In the project different kinds of diversity were applied to the I&C system itself as well as the

qualification process. None of the methods can be claimed to be complete and sufficient to dominate totally every kind of CCF.

To apply D3 methods to design and qualification of a safety critical system needs huge human, technical and financial resources. Therefore, it seems to be necessary to optimize - or at least to improve - criteria for a D3 approach to ensure the required safety and to limit the resources to reasonable expense. One step to this direction should be the quantification of the effectiveness of Defence-in-Depth and Diversity measures. The quantification has to consider the correlation of different D3 methods.

However, the experiences of the project show that D3 approaches provide a wide variety of methods to dominate CCF in safety I&C systems as well as in qualification processes.

References

1. IAEA Safety Glossary, Terminology Used in Nuclear, Radiation, Radioactive Waste and Transport Safety, Version 2.0, IAEA, Department of Nuclear Safety and Security (2006)
2. VDI Richtlinie VDI/VDE 3527: Kriterien zur Gewährleistung der Unabhängigkeit von Sicherheitsfunktionen bei der Leittechnik-Auslegung (2002)
3. Preckshot, G.G.: Methods for Performing Diversity and Defence-in-Depth Analyses of Reactor Protection Systems, UCRL-ID-119239, Lawrence Livermore National Laboratory (1994)
4. NUREG 0800, Branch Technical Position HICB-19, Guidance for Evaluation of Defence-in-Depth and Diversity in Digital Computer-Based Instrumentation and Control Systems, Rev. 4 (1997)
5. FDIS IEC 62340 Nuclear power plants – Instrumentation and Control Systems Important to Safety - Requirements to cope with Common Cause Failure (CCF) (2006)
6. Draft IAEA TECDOC on Avoiding Common-Cause Failures in Digital I&C Systems of NPPs (2006)
7. IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations, IEEE Std. 7-4.3.2- 2003 (2003)
8. NUREG/CR-6303 Method for Performing Diversity and Defense-in-Depth Analyses of Reactor Protection Systems (1994)
9. IEC 60880 Ed. 2 Nuclear power plants – Instrumentation and Control Systems Important to Safety - Software Aspects for Computer-Based Systems Performing Category A Functions (2006)
10. Lee, J.-S., Lindner, A., Choi, J.-G., Miedl, H., Kwon, K.-C.: Software Safety Lifecycle and Methods of Programmable Electronic Safety System for Nuclear Power Plant. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, Springer, Heidelberg (2006)
11. Lindner, A., Wach, D.: Experiences Gained from Independent Assessment in Licensing of Advanced I&C Systems in Nuclear Power Plants. Nuclear Technology 143, 197–207 (2003)
12. Lindner, A., Hoffmann, E., Wach, D.: Softwareprüfplan für sicherheitsrelevante Produkte, ISTec - A - 1068, Rev. 00 (2005)

Experimental Evaluation of the DECOS Fault-Tolerant Communication Layer

Jonny Vinter[1], Henrik Eriksson[1], Astrit Ademaj[2], Bernhard Leiner[3], and Martin Schlager[3]

[1] SP Technical Research Institute of Sweden
{jonny.vinter,henrik.eriksson}@sp.se
[2] Vienna University of Technology
ademaj@vmars.tuvien.ac.at
[3] TTTech Computertechnik AG
{bernhard.leiner,martin.schlager}@tttech.com

Abstract. This paper presents an experimental evaluation of the fault-tolerant communication (FTCOM) layer of the DECOS integrated architecture. The FTCOM layer implements different agreement functions that detect and mask errors sent either by one node using replicated communication channels or by redundant nodes. DECOS facilitates a move from a federated to an integrated architecture which means that non-safety and safety-related applications run on the same hardware infrastructure and use the same network. Due to the increased amount of data caused by the integration, the FTCOM is partly implemented in hardware to speed up packing and unpacking of messages. A cluster of DECOS nodes is interconnected via a time-triggered bus where transient faults with varying duration are injected on the bus. The goal of the experiments is to evaluate the fault-handling mechanisms and different agreement functions of the FTCOM layer.

1 Introduction

In a modern upper class car the number of electronic control units (ECUs) has passed 50 and the total cable length is more than two km. As a consequence, the weight of the electronics systems constitutes a non-negligible part of the total weight, and the cost of trouble-not-identified problems (TNIs) associated with connector faults is raising [1]. The trend of using a single ECU for each new function is costly and for mass produced systems indefensible. Additionally, several emerging subsystems will require data from and control over other subsystems. An example could be a collision avoidance subsystem which has to simultaneously control both the steer-by-wire and brake-by-wire subsystems [2]. This becomes very complex to implement on an architecture based on the federated design approach where a single ECU is required for each function. All these issues imply that a move towards an integrated architecture is necessary and this is the goal of DECOS (Dependable Embedded Components and Systems) [3], and also for similar initiatives such as AUTOSAR [4]. In an integrated architecture several applications can and will share the same ECU which requires temporal and spatial separation of applications. Within DECOS such encapsulation is

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 264–269, 2007.

ensured by high-level services like encapsulated execution environments [5] as well as virtual networks [6]. Other high-level services provided by the DECOS architecture are integrated services for diagnosis [7] and a new FTCOM layer based on the one used in TTP/C [8]. The goal of the DECOS FTCOM layer, whose validation is the focus of this paper, is to implement different agreement functions that detect and mask errors sent either by one node using replicated communication or by redundant nodes. Since the communication to and from a node will be increased in an integrated architecture, parts of the DECOS FTCOM layer are implemented in hardware to accelerate message packing and unpacking. The remainder of the paper is organized as follows. Section 2 briefly describes the DECOS fault-tolerant communication layer. The experimental setup used for the dependability evaluation is described in Section 3, and the results of the evaluation are presented and discussed in Section 4. Finally, the conclusions are given in Section 5.

2 DECOS FTCOM Layer

The FTCOM layer consists of one part which is implemented in hardware (Xilinx Virtex-4) together with the communication controller to accelerate frame sending and receiving as well as message packing and unpacking. During frame receiving, the two frame replicas received on the two replicated channels are checked by a CRC for validity and merged into a single virtual valid frame which is handed over to the upper FTCOM layers. The other parts of the FTCOM layer are implemented in software and are automatically generated by the node design tool [9]. The software-implemented parts have services for sender and receiver status as well as message age. However, the most important service is the provision of replica deterministic agreement (RDA) functions to handle replication by node redundancy. A number of useful agreement functions are provided. For *fail-silent subsystems*, where the subsystems produce correct messages or no messages at all, the agreement function '*one valid*' is provided. It uses the first incoming message as its agreed value. For *fail-consistent* subsystems, where incorrect messages can be received, the agreement function '*vote*' is provided. It uses majority voting to establish the agreed message and thus an odd number of replicas are needed. For *range-consistent* subsystems, e.g. redundant sensor values, the agreement function '*average*' is provided. It takes the arithmetic mean of the messages as its agreed value. For a case where the message is e.g. a boolean representing that the corresponding subsystem is alive, the agreement function '*add*' is provided, which sums up the living number of replicas and can therefore be used to get subsystem membership information. The function '*valid strict*' compares all valid raw values and only uses the (agreed) value if they are identical.

3 Experimental Setup

The DECOS cluster consists of a set of nodes that communicate with each other using replicated communication channels. A node knows the message sending instants of all the other nodes. Each node consists of a communication controller and a host application. The communication controller executes a time-triggered communication

protocol, whereas user applications and the software part of the FTCOM layer are executed in the host computer. Each DECOS node has a FPGA where the communication controller and the hardware part of the FTCOM layer are implemented. Application software, DECOS middleware and high-level services are implemented in the Infineon Tricore 1796 which is the host computer of the node. During the experiments, relevant data is logged in an external SRAM and after the experiments are finished data is downloaded via the debugger to a PC for analysis [10] (see Fig. 1). The disturbances are injected by the TTX - Disturbance Node [11] which is running synchronously with the cluster. A wide variety of protocol-independent faults can be injected such as bus breaks, stuck-at faults, babbling idiot faults and e.g. mismatched bus terminations. Faults can have a random, but bounded or specified duration, repeated at a random or specified frequency. The shortest pulse which can be injected has duration of 1 µs which corresponds to roughly 10-15 bits on the bus at a bandwidth of 10 Mbps. A counter application is used in this study to evaluate the FTCOM layer. The application toggles one boolean state variable and increments one float, and one integer variable, before the values of these variables are sent on the bus. The counter workload is executed on three redundant nodes. A fourth node is executing its RDA functions on the three replicated messages read on the bus to recover or mask an erroneous message. Each fault injection campaign is configured in an XML file where parameters such as fault type, target channel(s), fault duration and time between faults are defined.

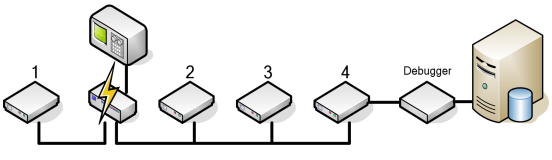


Fig. 1. Fault injection platform

4 Results

Several fault injection campaigns are carried out to validate that the FTCOM layer handles a message failure at replicated channels or at redundant nodes. The results are presented in the following sub sections.

4.1 Message Failure at Replicated Channels

The goal of this campaign is to validate the error detection and masking mechanisms of frames sent in replicated channels. Transient stuck-at-0 faults (denoted SA0), stuck-at-1 faults (denoted SA1), and white noise (denoted WN) with different burst lengths (from 1µs up to 500µs) are injected on communication channel A or channel B. The types of erroneous frames observed are *Invalid* frames and *Incorrect* frames. An Invalid frame is a case when a frame was expected but not received, or the frame does not have the specified header, or a wrong length. An Incorrect frame is the case when there is a mismatch between the data and the calculated CRC value (which means that data in the frame or the CRC are corrupted by a fault).

The expected number of erroneous frames *nef* due to injected disturbances can be calculated as:

$$nef = nf \cdot \frac{(bl - 1 + fl)}{sl} \cdot \frac{3}{4} \quad (1)$$

Where *nf* denotes the number of fault injected and *bl*, *fl*, and *sl* denotes the burst, length, frame length and slot length respectively (in µs). The ratio ¾ is needed because only three out of four nodes are analyzed while the fourth logs the results. In the counter workload, the frame and slot lengths are 35 and 625 µs, respectively and the number of injected faults is 310. For a burst length of 1 µs, the equation gives *nef* = 13 which agrees well with the experimental data. Equation (1) is also experimentally validated by a comparison between theoretical and practical results. The percentage of activated faults (Invalid and Incorrect frames) is shown in Fig. 2.

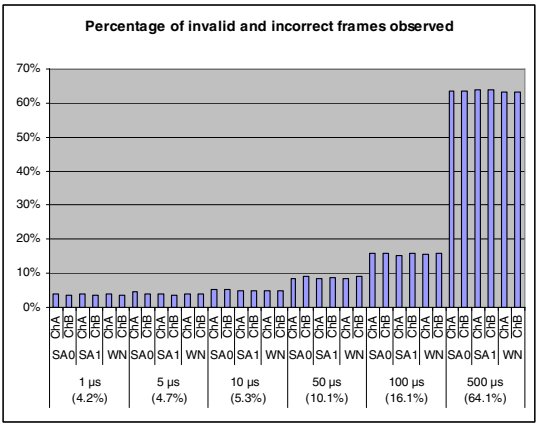


Fig. 2. Percentage of erroneous frames caused by different faults and burst lengths (presented both in micro seconds and as the probability of erroneous frames according to Equation 1)

4.2 Message Failure at Redundant Nodes

The goals of the campaigns presented in this section are to test the error detection and masking capabilities of the FTCOM layer in the case of failure of one of the redundant nodes. Here, the fault is injected in one of the redundant nodes, for example by disturbing the frames of the same node in both channels (campaign RN1). An additionally test was performed (RN2) to check the correctness at application level. In campaign RN3, faults are injected in the application layer of one node, forcing the node to produce value failures.

4.2.1 Campaign RN1

The type of faults used in RN1 could be caused by an erroneous packing of frames in the sending node. Fig. 3 shows the percentage of detected erroneous frames caused by simultaneous faults on both channels.

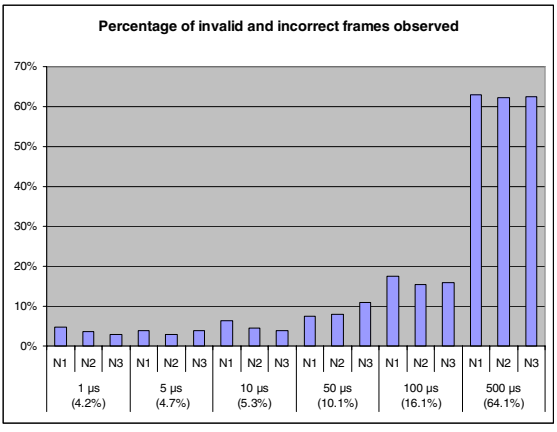


Fig. 3. Percentage of erroneous frames (observed at different nodes) caused by a uniform distribution of SA0, SA1 and WN faults with different burst lengths

4.2.2 Campaign RN2

The objective of this campaign is to not rely directly on the detection and masking capabilities of the FTCOM layer as in campaign RN1, but to check the correctness at the application level. During approximately 64 hours more than 4.5 million white noise faults of duration 500 µs were injected on both channels. According to Equation 1, nearly 2.9 million effective faults have thus been injected. The correctness was validated at the application level by checking the continuous increment of the counters and not a single application failure was observed.

4.2.3 Campaign RN3

Erroneous frames that are syntactically correct but semantically incorrect (i.e. correctly built frames including e.g. value failures) will not result in e.g. Invalid or Incorrect frames and should be recovered or masked by the RDA algorithms. The objective of this campaign is to evaluate how well the RDA algorithms can detect and handle a message failure caused by one of the redundant nodes with or without replicated channels. Five different RDA functions operating on three data types are evaluated. One node in the cluster is deliberately forced to send non-expected data such as min and max values, NaN (Not a number) and Infinity floats as well as arbitrary float, boolean and integer values. All RDA functions behaved as expected but some cases needs to be discussed. A valid float value [12] can be changed into NaN or Infinity floats by e.g. a single bit-flip fault (e.g. if a 32-bit float value is 1.5 and bit 30 is altered). Arithmetic with NaN floats will produce more NaN floats and the error may propagate quickly in the software [13]. Thus, the result of the RDA algorithms, '*average*' and '*add*' operating on NaN floats are therefore also NaN and this should be considered during the choice of appropriate RDA functions. The RDA function '*valid strict*' demands that all three replicated values must be exact copies and performs therefore a roll-back recovery each time one or two faulty nodes are detected. Thus, this RDA function seems powerful to recover from transient or intermittent node errors, even such errors causing e.g. NaN floats.

5 Conclusions

An experimental evaluation of the fault-tolerant communication layer of the DECOS architecture is presented. The evaluation shows that built-in error detection and recovery mechanisms including different RDA functions are able to detect, mask or recover from errors both internal in a redundant node or on a replicated communication network. However, some erroneous data values such as NaN floats must be handled by the RDA algorithm before they are passed to the application level. This is automatically handled e.g. by the RDA function '*valid strict*' which detects and recovers from a transient or an intermittent node value failure by using the previously agreed value instead (roll-back recovery). A final observation is that a '*median*' RDA function could be useful as an alternative to the *average* function.

Acknowledgments. This work has been supported by the European IST project DECOS under project No. IST-511764. The authors also give their credit to Guillaume Oléron who has been involved in the experiments as a part of his studies.

References

1. von Tils, V.: Trends and challenges in automotive engineering. In: Proceedings of the 18th International Symp. on Power Semiconductor Devices & IC's (2006)
2. Broy, M.: Automotive software and systems engineering. In: Proc. of the third ACM and IEEE International Conference on Formal Methods and Models for Co-Design (2005)
3. Kopetz, H., et al.: From a federated to an integrated architecture for dependable embedded real-time systems, Technical Report 22, Institut für Technische Informatik, Technische Universität Wien (2004)
4. AUTOSAR - Automotive Open System Architecture (2006), http://www.autosar.org
5. Schlager, M., et al.: Encapsulating application subsystems using the DECOS Core OS. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 386–397. Springer, Heidelberg (2006)
6. Obermaisser, R., Peti, P.: Realization of virtual networks in the DECOS integrated architecture. In: Proceedings of the 20th Intern. Parallel and Distributed Processing Symposium (2006)
7. Peti, P., Obermaisser, R.: A diagnostic framework for integrated time-triggered architectures. In: Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (2006)
8. Bauer, G., Kopetz, H.: Transparent redundancy in the Time-Triggered Architecture. In: Proceedings of the International Conference on Dependable Systems and Networks (2000)
9. The DECOS Cluster Design Tool for Layered TTP (Prototype), Edition 5.3.69a, TTTech Computertechnik AG (2006)
10. TTX-Disturbance Node User Manual, Edition 1.0.4, TTTech Computertechnik AG (2006)
11. Eriksson, H., et al.: Towards a DECOS Fault Injection Platform for Time-Triggered Systems. In: Proceedings of the 5th IEEE International Conference on Industrial Informatics (2007)
12. ANSI/IEEE Std 754 – IEEE Standard for binary floating-point arithmetic, IEEE (1985)
13. Vinter, J., et al.: Experimental dependability evaluation of a fail-bounded jet engine control system for unmanned aerial vehicles. In: Proceedings of the International Conference on Dependable Systems and Networks, Japan, pp. 666–671 (2005)

Achieving Highly Reliable Embedded Software: An Empirical Evaluation of Different Approaches

Falk Salewski and Stefan Kowalewski

Embedded Software Laboratory, RWTH Aachen University, Germany

Abstract. Designing highly reliable embedded software is a challenge and several approaches are known to improve the reliability of this software. However, all approaches have their advantages and disadvantages which makes empirical evaluations investigating their potentials necessary. In this paper, different approaches of software reliability improvement for embedded systems were compared on basis of experiments conducted at our institute. The first approach is an instance of N-version programming based on forced diversity. Two fundamentally diverse hardware platforms (microcontroller and CPLD/FPGA) were used to force diversity. Another experiment was conducted in which participants designed their software on one hardware platform only. The second half of this experiment was used for review and testing. Based on our experiments, the potentials of our application of N-version programming, review and testing are compared with respect to different fault categories (specification, implementation, application) identified during evaluation.

1 Introduction

In the domain of embedded systems, more and more systems require certain levels of reliability as for example future drive-by-wire systems in the automotive industry. These embedded systems are subject to strong development constraints as low cost and (hard) real-time requirements. In this context, applications of specific software reliability improvement measures are needed. In this paper, we will have a closer look at the application of three important approaches, namely N-version programming (NVP), testing and review.

The approach of NVP, firstly introduced by [3] seems very promising, but in the well known experiment of Knight and Leveson [5] it has been shown that developers tend to make the same faults. Different approaches modeling this dependency structure (e.g. [6]) and corresponding empirical studies [1,2] are known which allow certain (model-based) predictions of failure probabilities in NVP systems. Other approaches try to decrease the dependencies between the different software versions. One of these approaches is "forced diversity" introduced by [6] with an empirical evaluation in [8]. The basic assumption is that different development methodologies lead to diversity in decision and thus diversity in the behavior of the resulting product. However, recent publications [7,13] show, that even these improved approaches can lead to undesired dependencies between the diverse software versions. The approach of diverse NVP used for the evaluation

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 270–275, 2007.

in this paper is based on different hardware platforms used in today's embedded systems: microcontrollers (MCU) and programmable logic devices (PLD). The effect of this hardware based diversity on NVP has been analyzed with the help of experiments which are described in Section 2.

Testing, as the second approach investigated, can be applied at different stages in the design cycle and is currently one of the most important means of verification in embedded systems [9]. The disadvantage of all testing approaches is that in typical applications not every possible input combination can be checked. Embedded systems are often real-time systems which complicates the testing process additionally. The evaluation in this paper applies black box testing with an automatic test environment designed for this purpose and focuses on the general potentials of testing.

Different approaches of reviews are well known [4,9] and applied in industry. The idea is to inspect code, written by another person, to reveal problems in the code and to identify inconsistencies with the specification. Reviews offer chances to identify problems with respect to functional requirements, but also with respect to non-functional requirements as maintainability and reliability. However, the result of each review process depends a lot on the persons used as reviewers and their review performance is hard to quantify. In this paper we investigate the potentials of *code inspection* and details of this review-technique can be found in Section 2.

The option of using as many of these approaches as possible seems promising but is resulting in high costs (development time and human resources). Since in many embedded applications reasonable costs of the resulting products have to be achieved, evaluations which investigate the potentials of different approaches and their combinations to prevent failures are needed.

2 Design of Experiments

Three experiments were conducted to obtain the empirical data needed for the comparing evaluation of the different approaches of reliability improvement accomplished in later sections of this paper. This section gives a brief description of these experiments while a more detailed description can be found in [10].

The first experiment with respect to NVP was based on two different hardware platforms, namely MCUs and CPLDs and has been described in [13]. In order to validate the results and to investigate additional aspects, the experiment was replicated in a modified form. In this experiment, MCUs and FPGAs were used as diverse hardware platforms and each group had to program both hardware platforms starting with a platform picked randomly. While the task of the first experiment was mainly the frequency measurement of four independent speed signals and a communication via CAN bus (see [12,13] for details), the task of the second experiment had been extended with 6 additional tasks in order to increase the complexity of the application.

The aim of the third experiment was to identify which types of failures could be identified by review and by testing respectively. Each group had to program

the same task as used in the second NVP-experiment described above. However, only the microcontroller hardware was used for implementation and the second half of the lab course was used for review and testing. For testing, all test groups were equipped with an own automated test environment similar to the one used for evaluation. Each group received the machine code for testing and an empty test report form which had to be filled out. As in the case of testing, every review group received an empty review report form, a short review instruction and the source code to review. Additionally they received the corresponding documentation which should help them to understand the code if necessary.

In case of all three experiments, each version had to pass an automated acceptance test in order to receive versions with a certain minimum level of quality. In the following evaluation test, the accepted versions were tested by an automatic real-time test environment designed for this purpose. Further details of the evaluation and challenges resulting from real-time requirements and specific properties of embedded systems can be found in [10,11].

3 Experiment Results

For the analysis in this paper, we took a closer look at the most recent common mode failures and listed them in Fig. 1 for all three experiments (cell is colored if amount of versions with this failure is $> 50\%$). It has to be noted that a stronger acceptance test was used for the 2nd and 3rd experiment. Further details cannot be listed here for space reasons, but can be found in [10].

4 Fault Classification and Potentials of the Approaches

During evaluation of the NVP experiment results, it became obvious that different sources exist for the failures found. Those failure sources (faults) have been identified as follows:

- **Specification specific faults:** the specification was misleading or ambiguously.
- **Application specific faults:** application specific problems and challenges have not been understood and thus have not been handled sufficiently (e.g. forget to handle a certain scenario/input constellation).
- **Implementation specific faults:** specification and application specific problems have been identified correctly, but faults have been made during implementation (e.g. incomplete case structure).

The failures found in our three experiments have been analyzed with respect to these fault categories and the results, presented in Fig.2 qualitatively, are discussed in the following (see [10] for a more detailed failure description).

As expected, diverse hardware NVP allowed to mitigate most but not all of the *implementation specific* faults (e.g. No.7-9, 11, 12 in Fig. 1 were handled while No. 10 was faulty in several versions). In case of non real-time tasks reviews uncovered many implementation specific faults while testing was more

No.	Failure describtion	Type of problem	# versions with this failure				
			Exp. 1*		Exp. 2		Exp.3
			MCU	CPLD	MCU	FPGA	MCU
1	Wrong or delayed CAN messages after reset, especially if the input signal frequency is high.	Impl./ Spec.	100%	10%	36%	20%	25%
2	Wrong values in the CAN message as soon as input signals are of very low frequency (<5Hz). → measurement interval probably too short	Appl.	33%	80%	18%	40%	25%
3	Wrong values in the CAN message as soon as input signals are close above the maximum frequency explicilty specified. → Overflows or wrong determination of maximum output value	Impl./ Appl.	42%	20%	**0%**	**0%**	**0%**
4	Missing or delayed CAN messages or messages with wrong values if subsequent input signal values change quickly.	Appl.	83%	40%	91%	100%	75%
5	Wrong or delayed results as soon as different values are fed into the four measurement channels while changes of subsequent input signal values are limited to ~3.5% (200Hz) of the max. frequency → et al.: faulty determination of the measurement interval	Appl.	n.a.	n.a.	82%	90%	75%
6	Test cases with only equal test values at all inputs do not always lead to 4 identical results (not necessarily a failure).	Impl.	33%	30%	64%	20%	42%
7	Device stops sending of CAN messages as soon as the input frequency has been above a certain threshold**	Impl.	**0%**	**0%**	9%	**0%**	**0%**
8	No messages after reset if the input signal frequency is low	Impl.	**0%**	**0%**	**0%**	**0%**	8%
9	Device stops sending of CAN messages as soon as two buttons are pressed sequentially with high frequency → probably leading to an undefined state in state machine	Impl.	n.a.	n.a.	**0%**	10%	**0%**
10	Testmessage counter does not always start with 0 as specified → at least in some cases: faulty interaction between testmessage counter and sending method	Impl.	n.a.	n.a.	27%	60%	17%
11	Testmessage counter is not incremented correctly (is incremented by more than 1) → faulty interaction between testmessage counter and sending method	Impl.	n.a.	n.a.	**0%**	60%	**0%**
12	User input via buttons changes the number of wrong values for the worse → real-time properties affected by user input	Impl.	n.a.	n.a.	27%	**0%**	8%
	number of versions used for analysis:		12	10	11	10	12

* only elementary acceptance test.
** close above the maximum input signal frequency explicitly specified if no button in pressed.
Impl. = Implementation, Appl. = Application, Spec. = Specification.
Exp.3: versions have been debugged according to individual test and review reports.

Fig. 1. Failures found during evaluation

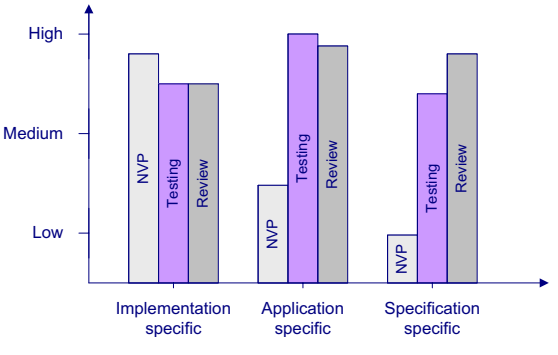


Fig. 2. Failure mitigation potentials with respect to failure categories

successful in case of real-time functionalities. For this reason, high potentials of implementation specific fault mitigation are assigned for NVP (not the best value according to common mode failure described), closely followed by test and review, since many implementation specific problems had not been identified by all reviewers/testers.

Despite the immense effort put into different development processes, languages and programming styles by using completely different hardware platforms in our NVP experiments, several problems remained the same in all implementations leading to identical wrong results in several cases. These problems (No. 2-5, Fig. 1) result from the application itself, are implementation independent and thus cannot be avoided by this approach of NVP. Some of these *application specific* faults occurred less often on the first hardware while others were present less often on the other hardware. However, these differences were usually small (exceptions are No.2, Exp.1 and No.6, Exp2. in Fig. 1). For this reason, only low to medium potentials are seen for diverse hardware NVP to mitigate application specific faults. On the other hand, testing and review discovered most of the known application specific problems. With respect to application specific faults, testing showed a little more advantages in comparison to review since also unexpected faults were identified during testing while reviewers typically concentrated on finding known problems in the code.

Finally, *specification specific* faults were a problem for all three approaches. In case of diverse hardware NVP, only few specification specific faults could be avoided (if one hardware platform was guiding to the correct implementation, as in Exp.1, No.1, Fig. 1). In case of testing and review, all known specification problems had been identified, but in several cases only by a minority of the groups. Review seemed to have the highest potentials to reveal specification specific problems, since the specification had been analyzed closely for review, while it was used only for test case generation in the test process.

5 Conclusion

The three different approaches analyzed in this paper showed different potentials with respect to the three fault categories identified.

During the analysis of the two NVP experiments, high numbers of dependent failures had been found which resulted from application specific faults. The reason for this common mode failures are application specific difficulties, which exist independently from the implementation. A similar problem exists for specification specific problems. Several ambiguities could be uncovered by NVP, or even mitigated by an implementation on one hardware platform (No.1, Exp1, Fig. 1). However, our results show no hint that the **majority** of implementations delivers correct results with respect to the intended behavior. For the third failure category, namely implementation specific faults, it had been expected that diverse hardware NVP would allow maximum diversity between the faults in the different versions. However, even in this case at least one common mode failure has been introduced in several versions (No.10 in Fig. 1).

Review and testing showed medium to high potentials with respect to all three fault categories. With respect to specification specific faults, reviews showed the highest potentials. It seems that during review the specification is read more intensely than during testing in which the specification is used only for test case generation. In case of application specific faults, testing showed the highest potentials. A reason might be that, while review focuses on finding known problems in the code, testing could reveal unexpected faults. In the case of implementation specific faults, testing and review showed lower potentials for reliability improvement than our NVP approach. These implementation specific faults were often related to real-time requirements which are comparatively hard to detect by testing and review.

References

1. Bentley, J., Bishop, P., van der Meulen, M.: An empirical exploration of the difficulty function. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, Springer, Heidelberg (2004)
2. Cai, X., Lyu, M.R.: An empirical study on reliability modeling for diverse software systems. In: 15th International Symposium on Software Reliability Engineering (ISSRE) (2004)
3. Chen, L., Avizienis, A.: On the implementation of n-version programming for software fault tolerance during program execution. In: International Computer Software and Applications Conference (COMPSAC) (1977)
4. Fagan, M.: Design and code inspections to reduce errors in program development. Technical report, IBM (1976)
5. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Softw. Eng. 12 (1986)
6. Littlewood, B., Miller, D.R.: Conceptual modeling of coincident failures in multiversion software. IEEE Trans. Softw. Eng. (1989)
7. Littlewood, B., Popov, P., Strigini, L.: A note on modelling functional diversity. Reliability Engineering an System Safety (1999)
8. Lyu, M.R., He, Y.-T.: Improving the n-version programming process through the evolution of a design paradigm. IEEE Transactions on Reliability 42 (1993)
9. Parnas, D.L., van Schouwen, J., Kwan, S.P.: Evaluation of safety-critical software. Communications of the ACM 33, 636–648 (1990)
10. Salewski, F., Kowalewski, S.: Achieving highly reliable embedded software: An empirical evaluation of different approaches. Technical Report AIB-2007-08, Dep. of Computer Science, RWTH Aachen University (2007)
11. Salewski, F., Kowalewski, S.: Testing issues in empirical reliability evaluation of embedded real-time systems. Technical Report WUCSE-2007-17: Proceedings of the Work-In-Progress Session of RTAS'07, Dep. of Computer Science & Engineering, Washington University in St. Louis (2007)
12. Salewski, F., Wilking, D., Kowalewski, S.: Diverse hardware platforms in embedded systems lab courses: A way to teach the differences. In: First Workshop on Embedded System Education (WESE), vol. 2, SIGBED Review (2005)
13. Salewski, F., Wilking, D., Kowalewski, S.: The effect of diverse hardware platforms on n-version programming in embedded systems - an empirical evaluation. In: 3rd International Workshop on Dependable Embedded Sytems (WDES) (2006)

Modeling, Analysis and Testing of Safety Issues - An Event-Based Approach and Case Study

Fevzi Belli[1], Axel Hollmann[1], and Nimal Nissanke[2]

[1] University of Paderborn, Germany
[2] London South Bank University, London, UK

Abstract. This paper proposes an event-based approach with an intuitive simple graphical representation of the system and its environment for designing, analysis and testing safety-critical systems. The events are user actions and system responses, and are ordered according to the threats posed by the resulting system states. This ordering is an integral aspect of the graphical representation, making it possible to directly identify the risks associated with each and every functionally desirable, and undesirable, event relative to one another. Tests that target safety requirements are devised by examining possible traces of these events, represented compactly by regular expressions, exhibiting particular risk patterns such as human error and system failures.

Keywords: Safety, Analysis and Testing, Event Sequence Graphs, Risk Graphs, Regular Expressions, User Interactions.

1 Introduction, Related Work

Modern safety critical systems with automated monitoring and control, sophisticated man-machine interfaces place much greater demands on software to deliver timely response to failures, correct performance of complex tasks and required level of reliability. These are complex requirements vital to assuring system safety [7, 12]. While enhancing the reliability of safety critical systems, man-machine interfaces are known to introduce additional vulnerabilities, requiring careful consideration of various factors such as aspects of automation, psychological issues and ergonomics.

In the face of such vulnerabilities, analysis and testing form an important part of the system development process in revealing and eliminating system's faults. Because of the substantial costs involved in testing, particularly in critical applications requiring extensive testing lasting several months, both testability and the choice of tests to be conducted become an important design consideration. Because of the conflicting demands of minimizing the extent of tests and maximizing the coverage of faults, it is therefore critically important to follow a systematic approach to identifying the test sets that focus on safety, as well as tests that address specific safety requirements.

State-based and event-based methods have been used for almost four decades for specification and testing of software and system behavior, e.g., for conformance testing [3], as well as for specification and testing of system behavior [4] and more recently by [10]. A different approach for testing [8] deploys knowledge engineering methods to generate test cases, test oracles, etc.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 276–282, 2007.

Our approach is black-box-oriented. Tests that target safety requirements are devised by examining possible traces of events exhibiting particular risk patterns. These sequences are represented by regular expressions, the undesirable events in them representing human error and system failures, while the desirable events including, in addition to functional ones, various recovery measures to be undertaken following undesirable events. The approach can be used not only for requirements analysis and validation before implementation, but also for analysis and testing of existing code, detecting output faults, detecting erroneous internal states, etc., at low levels of abstraction. This paper is an introduction to our approach and unlike our prior work [2] focuses on safety and testability and demonstrates its use in designing tests that target safety aspects as part of an integrated system development process.

2 Event-Based Modeling of System Vulnerabilities

This work uses *Event Sequence Graphs (ESG)* [2] for representing the user and system behavior. Due to the lack of space, we will give only a brief introduction. Abstractly, an ESG is a digraph $M = (\alpha, E)$, α being a set of nodes uniquely labeled by some *input symbols*, also denoted here by α, and E a non-empty relation on α, with elements in E representing directed arcs between the nodes in α. Each arc represents a pair of consecutive legal events called a *legal event pair (EP)*. *Faulty event pairs* (*FEP*) are the edges of the corresponding $\overline{ESG}$. The set α can be partitioned into two disjoint subsets α_{env} (*environmental* events, e.g., user inputs) and α_{sys} (*system* signals). This distinction is important because events in the latter are controllable within the system, whereas events in the former are not subject to such control.

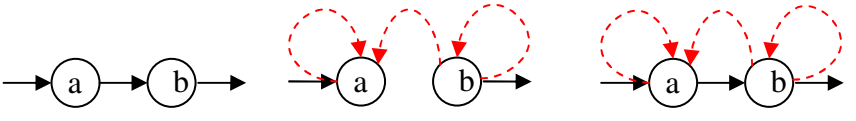


Fig. 1. Example of an ESG, its complement $\overline{ESG}$ and CESG (Completed ESG)

Regular expressions based on alphabet α are used for describing patterns of interactivity between the system and its environment. System *functions* (F), as well as the threats to safety, may each be described using two disjoint subsets of strings, one belonging to the language $L(M)$ and another not belonging to $L(M)$, respectively. *Legal* state transitions are brought about by *desirable* events, leading to symbol sequences belonging to $L(M)$. *Illegal* transitions represent *undesirable* events, leading to faulty symbol sequences not belonging to $L(M)$, signifying breaches to *vulnerabilities* (V).

$$F \subseteq L(M) \text{ and } V \subseteq \overline{L(M)} \tag{1}$$

Risks to safety are often related to the system state. Our approach refers to states indirectly in terms of strings in $L(M)$. Thus, a string $s \in L(M)$ may also be treated as an interchangeable notation for the state reached by the execution of the events in s. The remainder of the vulnerability specification consists of a *risk ordering relation*

$\sqsubseteq$ - a relation on $L(M)xL(M)$. It is defined such that, given states $s_1, s_2 \in L(M)$, $s_1 \sqsubseteq s_2$ is true if and only if the risk level of s_1 to breaches of safety is known to be less than, or equal to, the risk level of s_2 [9]. In this context, *risk level* quantifies the *"degree of the undesirability"* of an event from the safety perspective. A specific benefit of risk ordering in our framework is that it allows a more systematic approach to selection of test cases by focusing on particular vulnerability attributes. The risk ordering relation is intended as a guide to decision making upon the detection of a threat and on how to react to it. The required response to breaches of vulnerability needs to be specified in a *defense matrix* $D \in L(M)xV \rightarrow L(M)$, which is a partial function. The defense matrix utilizes the risk ordering relation to revert the system state from its current one to a less, or the least, risky state. In this sense, D is subject to the following constraint:

$$\forall\, s_1, s_2, v: (s_1,v) \in D \wedge D(s_1,v) = s_2 \Rightarrow s_2 \sqsubseteq s_1 \tag{2}$$

This expresses the requirement that, should it encounter the vulnerability v in any given state s_1, the system must be brought down to a state s_2, which is of a lower risk level than s_1. The means by which this is brought about is called an *exception handler*, or a defensive action, which is an appropriately enforced sequence of events. The actual definition of the defense matrix and the appropriate set X of exception handlers is the responsibility of a domain expert specializing in the risks to a given vulnerability. If x is a defense action, then $s_1 x = s_2$. Finally, the model of an application *defended* against vulnerabilities is defined as $M_d = (\alpha, E, F, V, \sqsubseteq, D, X)$.

3 Safety Critical Features and a Case Study

The system under test is a terminal which controls a marginal strip mower (Fig. 2, left top), a vehicle that takes optimum advantage of mowing around guide poles, road signs and trees, etc. The buttons on the control desk (Fig. 2, left bottom) simplify the operation, so that, e.g., the mow head (with revolving knives) returns to working or to transport position when a button is pressed. Beside the positioning, the inclination and the support pressure of the mowing unit can be controlled.

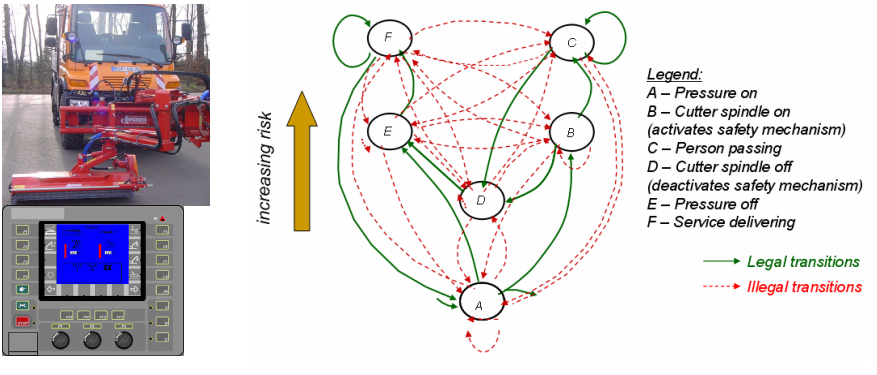


Fig. 2. Marginal strip mower, control desk and its completed ESG model

The ESG in Fig. 2, right, represents the control desk. The set of events α is given by $\alpha_{sys} = \{A,B,D,E\}$ and $\alpha_{env} = \{C,F\}$. Here, the symbol C stands for the event of a passing object, e.g., a person, whereas symbol F denotes a service delivery, e.g., a mechanic repairing the mower unit. The pressure of the mowing unit can be switched on and off and the two events are denoted by symbols A and E. The cutter spindle of the revolving knives is controlled by symbols B and D. When the cutter spindle is switched on (B), a safety mechanism is activated. This mechanism recognizes objects which approach to the mower unit. When the spindle is switched off (D), the safety mechanism is also deactivated. These events bring about hazardous states posing different risks. For example, state C (a person passing) represents a state with the highest risk. As is shown by directed arcs in Fig. 3, the EPs in this example are

$$AE,\ EF,\ FF,\ FA,\ AB,\ BC,\ BD,\ CC,\ CD,\ DE \tag{3}$$

while the complete event sequences (CESs) in any complete cycle of system operation can be represented by the regular expression

$$RegEx = (AEF^{+})^{*}A+((AEF^{+})^{*}ABC^{*}DEF^{+})^{*}A = ((AEF^{+})^{*}\ (\lambda + ABC^{*}DEF^{+}))^{*}A \tag{4}$$

The FEPs shown as dashed lines in Fig. 2 are given by

$$AA,\ DD,\ BB,\ EE,\ AD,\ AC,\ AF,\ DA,\ DB,\ DC,\ DF,\ BA,\ BF,\ BE,\ CA,\ CB,\ CF,\ CE,\ FD,\ FB,\ FC,\ FE,\ ED,\ EB,\ EA,\ EC \tag{5}$$

Expression (4) constitutes *system function F*, while those of (5) the *vulnerability threats V* posed at the junctures corresponding to any matching sub-expression among the ESs that can be generated from (4), e.g., $(AEF^{+})^{*}A$. Each FEP in (5) represents the leading pair of events of an emerging faulty behavioral pattern, with the first event being an acceptable one and the second an unacceptable one. Should the first event of any of the FEPs, e.g., AC, thus happen to match the last event in any of the ESs that can be generated by such a sub-expression, e.g., $(AEF^{+})^{*}A$, then the corresponding pair of ES and FEP, e.g., $(AEF^{+})^{*}\ AC$, describes, or signifies, the occurrence of a specific form of a faulty behavioral pattern.

Table 1. Mower vulnerabilities, the level of the threats posed, and possible defense action

ES (Col. 1)	FEP (Col. 2)	Interpretation (Col. 3)	Comment (Col. 4)	Defense action (Col. 5)
$(AEF^{+})^{*}$ AB	BA	Pressure switched on, though already on.	Ignored	–
	BF	Delivering a service, e.g., repair at running system.	Danger	BD
	BE	Switches of pressure though cutter spindle active.	Danger	BD

Table 1 presents an extract of the vulnerabilities relevant to the model given in Fig. 2. In order to overcome the inadequacy of the representation in Table 1, a *risk graph* of the form in Fig. 3 may be used to express the relative risk levels of states with a greater

degree of formality and precision. Each node in the risk graph represents a state described by a regular expression. A directed arc running from a state s_1 to s_2 in Fig. 3 is equivalent to $s_1 \sqsubseteq s_2$, signifying that the risks posed by state s_2 is known to be at the same level as, or exceed, the risks by s_1. In the diagram, an upward running arc signifies that the state lying above poses a greater risk than the one lying below. Arcs drawn in solid lines, as well as the states denoted by underscored ES (regular expressions), refer to the normal functional behavior, while those in dashed lines and other (non-underscored) FES (regular expressions) refer to vulnerability states. For reducing clutter, loops at nodes have been omitted in the diagram.

Having identified potential vulnerabilities of safety concerns, it is possible to provide measures that counteract them. This was what is intended by the concept of *defense matrix* and *exception handlers*. An attempt has been made in Table 1 to propose the defensive actions that may be taken, although due to the limited scope of our model these actions address the potential vulnerabilities only partially.

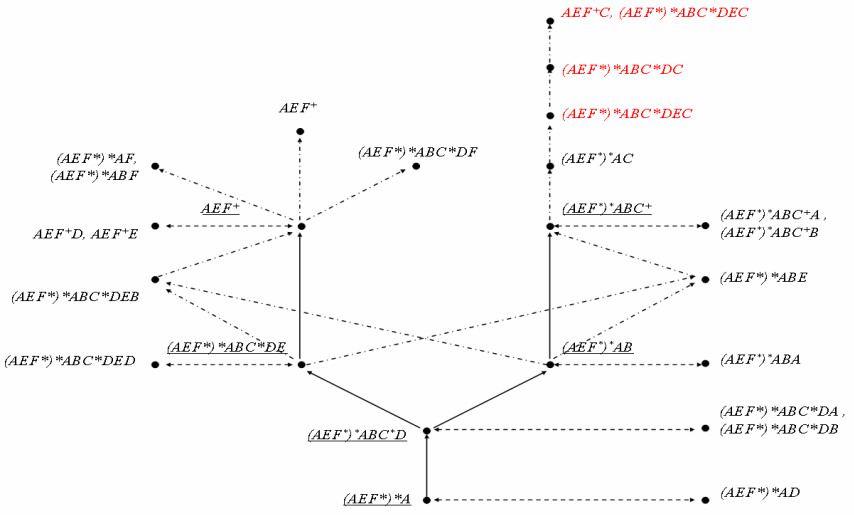


Fig. 3. Risk graph

Should Table 1 have been complete in these respects, the event sequences listed in column 5 would have been equivalent the set of the exception handlers X, while columns 1,2 and 5 would have amounted to a definition of the required defense matrix implicitly, provided that the data in these columns satisfies condition (2). Note that the concatenation of expressions for the current state, columns 1, 2 and 5 in the appropriate manner (by dropping common signals as appropriate) gives the state aimed at by the defense matrix as a result of invoking the corresponding exception handler.

We can use the CESG in Fig. 3 to simulate all potential test scenarios. All what is required is the information given in EPs (3) and FEPs (5), which can be generated by

Table 2. Excerpt of issues detected

No.	Faulty Behavioral Pattern Detected
1	When the function *PressureOn* is deactivated, the function *CutterSpindleOn* can only then be activated if the function *PressureOn* is activated (contradiction!).
2	A change from the view *RSM_Mode_I* to the view *RSM_TranspWorkpos* while function *PressureOn* is active is possible only if the function *PressureOn* is deactivated.
3	When the function *CutterSpindleOn* is activated, the function *PressureOn* can only be deactivated if the function *CutterSpindleOn* is deactivated.

expanding the RegEx (4), applying wellknown algorithms [11]. An excerpt of faulty behavioral patterns detected by the tests is given in Table 2.

4 Conclusion and Future Work

Based on [1, 5, 9], this paper has introduced an integrated design approach capable of addressing system design against system vulnerabilities threatening safety. It allows the consideration of further vulnerabilities threatening other system attributes, such as security and usability, in a single framework and in a similar manner. Incorporation of both the desired and the undesired features of the system in the model allows a practical way to realize the "design for testability" in software design. The degree of undesirability is represented in the form of a risk ordering relation – an expression of relative levels of risks posed by hazardous states. This allows targeting the design of tests at specific system attributes. The framework is based on the concept of 'event sequence graphs'. This could form the basis for the adoption of the approach in other software modeling approaches and tools such as Statecharts [6].

References

[1] Belli, F., Grosspietsch, K.-E.: Specification of Fault-Tolerant System Issues by Predicate/Transition Nets and Regular Expressions – Approach and Case Study. IEEE Trans. On Softw. Eng. 17/6, 513–526 (1991)
[2] Belli, F.: Finite-State Testing and Analysis of Graphical User Interfaces. In: Proc. 12th Internat'l. Symp. Software Reliability Engineering, pp. 34–43 (2001)
[3] Bochmann, G.V., Petrenko, A.: Protocol Testing: Review of Methods and Relevance for Software Testing. Softw. Eng. Notes, ACM SIGSOFT, 109–124 (1994)
[4] Chow, T.S.: Testing Software Designed Modeled by Finite-State Machines. IEEE Trans. Softw. Eng. 4, 178–187 (1978)
[5] Eggers, B., Belli, F.: A Theory on Analysis and Construction of Fault-Tolerant Systems (in German). In: Informatik-Fachberichte 84, pp. 139–149. Springer, Berlin (1984)
[6] Harel, D., Namaad, A.: The STATEMATE Semantics of Statecharts. ACM Trans. Softw. Eng. Meth. 5, 293–333 (1996)
[7] Leveson, N.G.: Safeware, System Safety and Computers. Addison-Wesley, Reading (1995)
[8] Memon, A.M., Pollack, M.E., Soffa, M.L.: Automated Test Oracles for GUIs. In: SIGSOFT 2000, pp. 30–39 (2000)

[9] Nissanke, N., Dammag, H.: Design for Safety in Safecharts With Risk Ordering of States. Safety Science 40, 753–763 (2002)
[10] Offutt, J., Shaoying, L., Abdurazik, A., Ammann, P.: Generating Test Data From State-Based Specifications. The Journal of STVR 13(1), 25–53 (2003)
[11] Salomaa, A.: Theory of Automata. Pergamon Press, Oxford (1969)
[12] Storey, N.: Safety-critical computer systems. Addison-Wesley, Reading (1996)

A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications*

Jürgen Mottok[1], Frank Schiller[2], Thomas Völkl[3], and Thomas Zeitler[3]

[1] Regensburg University of Applied Sciences, Faculty of Electronics and Information Technology, Seybothstr. 2, D-93049 Regensburg, Germany
juergen.mottok@e-technik.fh-regensburg.de
[2] Technical University Munich, Institute of Information Technology in Mechanical Engineering, Boltzmannstr. 15, D-85748 Garching near Munich, Germany
schiller@itm.tum.de
[3] SIEMENS VDO Automotive AG, Siemensstr. 12, D-93055 Regensburg, Germany
thomas.zeitler@siemens.com, voelkl.thomas@siemensvdo.com

Abstract. Currently, both fail safe and fail operational architectures are based on hardware redundancy in automotive embedded systems. In contrast to this approach, safety is either a result of diverse software channels or of one channel of specifically coded software within the framework of Safely Embedded Software. Product costs are reduced and flexibility is increased. The overall concept is inspired by the well-known Vital Coded Processor approach. Since Mealy state machines are frequently used in embedded automotive systems, application software with a general Mealy state machine is realized differently with Safely Embedded Software starting from the high level programming language C with corresponding measurements.

Keywords: Safely Embedded Software, Safe State Machine, Diverse Instructions, Safety Code Weaving, Safety Supervisor.

1 Introduction and Related Work

The importance of the non-functional requirement safety is more and more recognized in the automotive industry and therewith in the automotive embedded systems area. In contrast to functions without a relation to safety, the execution of safety-related functions necessitates additional considerations and efforts.

The normative regulations of the generic industrial safety standard IEC 61508 [5] can be applied to automotive safety functions as well. In the future, the new automotive safety standard ISO/WD 26262 will be available.

In the paper, the concept of Safely Embedded Software (SES) is proposed. This concept is capable to reduce redundancy in hardware by adding diverse redundancy in software. Safely Embedded Software enables the realization and the

* This work is supported by the FHprofUnd program of the German Federal Ministry of Education and Research(FKZ 1752X07).

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 283–288, 2007.

proof of safety properties of software by specific coding with the result of diverse data and diverse instructions. Software diversity is one technique to fulfill the single fault criterion [1,2]. The coding avoids non-detectable common-cause failures in the software components. Safely Embedded Software does not constrict capabilities but can supplement multi-version software fault tolerance techniques [7] like N version programming, consensus recovery block techniques, or N self-checking programming.

In 1988, the Vital Coded Processor [3] was published as an approach to design typically used operators and to process and compute vital data with non-redundant hardware and software. But in particular, the Vital Coded Processor approach can not be handled as standard embedded hardware.

The paper is structured as follows. In Chapter 2 the Safely Embedded Software approach is introduced. Chapter 3 contains a case study with a typical automotive state machine. The paper ends with some conclusions in chapter 4.

2 The Safely Embedded Software Approach

Safely Embedded Software (SES) can establish safety independently of the type of the processing unit. It is possible to detect permanent errors, e. g. errors in the Arithmetic Logical Unit (ALU) as well as temporary errors, e. g. bit-flips, and their impact on data and the control flow. SES runs on the application software layer. Several application tasks have to be safeguarded like e. g. the evaluation of some diagnosis data and the check of the data from the sensors. Because of the underlying principles, SES is a processor and operating system independent safety approach.

Furthermore, SES is a programming language independent approach. Its implementation is possible in assembler language as well as in an intermediate or a high programming language like C. When using an intermediate or higher implementation language, the compiler has to be used without code optimization. A code review has to assure, that neither a compiler code optimization nor removal of diverse instructions happened. Basically, the certification process is based on the assembler program or a similar machine language. Since programming language C is the de facto implementation language in automotive industry, the C programming language is used in this study exclusively.

Figure 1 shows the method of safety code weaving. Safety code weaving is the procedure of adding a second software channel to an existing software channel. In this way, SES adds a second channel of the transformed domain to the software channel of the original domain. In dedicated nodes of the control flow graph, comparator functionality is added. Though, the second channel comprises diverse data, diverse instructions, comparator and monitoring functionality. The comparator or voter, respectively, on the same Electonic Control Unit (ECU) has to be safeguarded with voter diversity [1].

Safely Embedded Software is based on the (AN+B)-code of the Vital Coded Processor [3] transformation of original integer data x_f to diverse coded data x_c (see Table 1).

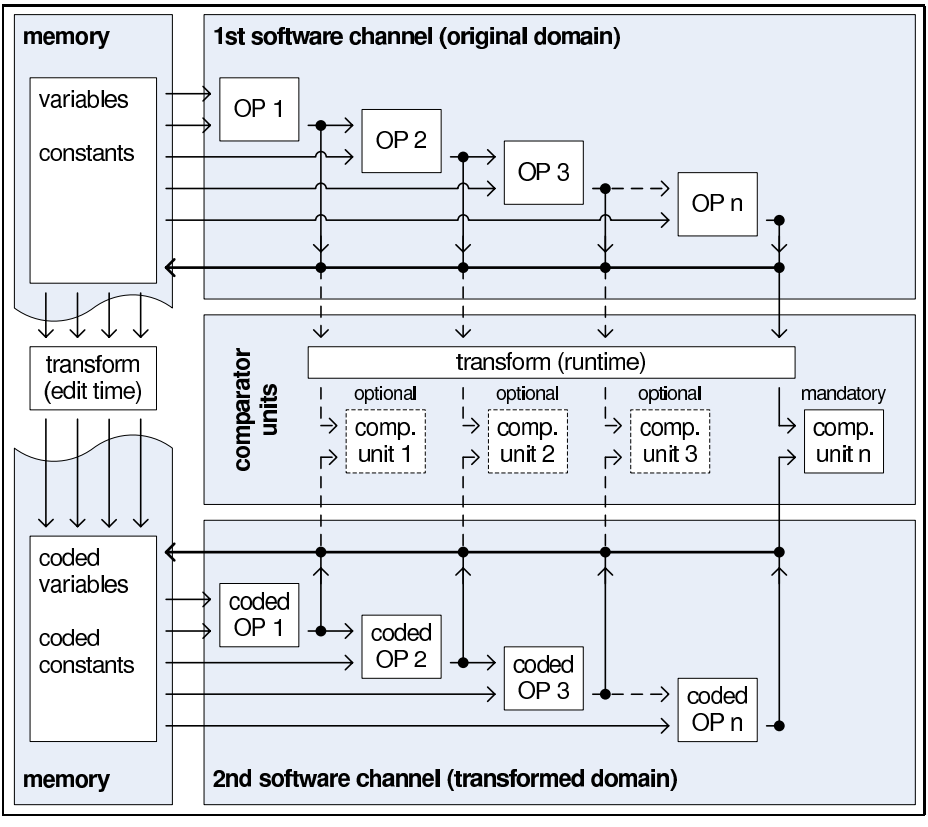


Fig. 1. Safety Code Weaving

Table 1. Selected rules of SES whereas $x_c, x_f \in \mathbb{Z} \ \wedge \ A \in \mathbb{N}^+ \ \wedge \ B_x, D \in \mathbb{N}_0$

Area of SES	Used Technique	Rule
Coded Data x_c	$x_c := A * x_f + B_x + D$	1
Coded Operator $\mathrm{OP_c}$ (common case)	$z_c := x_c \, \mathrm{OP_c} \, y_c$	2
Coded Addition $\oplus$ (special case of $\mathrm{OP_c}$)	$z_c := x_c \oplus y_c$ $z_c := x_c + y_c + (B_z - B_x - B_y) - D$	2
Coded logical Operator $\mathrm{geqz_c}$ (special case of an unary $\mathrm{OP_c}$)	$\mathrm{geqz_c}(x_c) := \mathrm{TRUE_c}$, if $x_f \geq 0$ $\mathrm{geqz_c}(x_c) := \mathrm{FALSE_c}$, if $x_f < 0$ $\mathrm{geqz_c}(x_c) := \mathrm{ERROR_c}$, if x_c is invalid	2
Local program flow monitoring	Safeguarding of C control structures by means of coded data	4
Global program flow monitoring	Classical key value method [6]	5

In the following, the principal procedure of safety code weaving is demonstrated in compliance with nine rules:

1. *Diverse Data.* The prime number A determines important safety characteristics like the Hamming Distance and the residual error probability of the code and some other properties like the bit field size necessary for the coded data. The static signature B_x ensures the correct memory address of variables by using the memory address of the variable or any other variable specific number. The dynamic signature D ensures that the variable is used in the correct task cycle. It can be calculated by a clocked counter.
2. *Diverse Operations.* A coded operator $\mathrm{OP_c}$ is an operator in the transformed domain that corresponds to an operator OP in the original domain. Each transformed operation follows directly the original operation. Original and safety code are interweaved in this way.
3. *Update of dynamic signature.* In each task cycle, the dynamic signature of each variable has to be incremented.
4. *Local (logical) program flow monitoring.* The C control structures are safeguarded against local program flow errors. The branch condition of the control structure is transformed and checked inside the branch.
5. *Global (logical) program flow monitoring.* This technique includes a specific initial key value and a key process within the program function to assure that the program function has been processed completely in the given parts and in the correct order [6].
6. *Temporal program flow monitoring.* Dedicated checkpoints have to be added for monitoring periodicity and deadlines. The specified execution time is safeguarded.
7. *Comparator function.* Comparator functions or check functions, respectively, have to be added in the specified granularity in the program flow for each task cycle. Either a comparator verifies the diverse channel results $z_c = A * z_f + B_z + D?$, or the coded channel is checked directly by checking the condition $(z_c - B_z - D) \bmod A = 0?$.
8. *Safety protocol.* Safety critical and safety related software modules (in the application software layer) communicate intra or inter ECU via a safety protocol [4]. Therefore a safety interface is added to the functional interface.
9. *Safe communication with a safety supervisor.* Fault status information is communicated to a global safety supervisor. The safety supervisor can initiate the appropriate (global) fault reaction [4].

According to rule 1 and rule 2 in Listing 1 the example code is transformed. All C control structures are transformed following the given rule set. It can be realized that the *Greater or equal zero* operator($\mathrm{geqz_c}$) of the transformed domain is frequently applied for safeguarding C control structures. Its usage in principal is presented in Listing 1.

Listing 1. Example code after applying rule 1 and rule 2

```
1  int af;                  int ac;                                    // <= rule 1
2  int xf;                  int xc;                                    // <= rule 1
3  int tmpf;                int tmpc;                                  // <= rule 1
4
5  af    = 1;               ac    = 1*A + Ba + D; //coded 1            // <= rule 2
6  xf    = 5;               xc    = 5*A + Bx + D; //coded 5            // <= rule 2
7  tmpf = ( xf >= 0 );      tmpc = geqz_c( xc ); //greater/equal zero  // <= rule 2
8
9  if ( tmpf )
10 {
11     af = 4;              ac    = 4*A + Ba + D; //coded 4            // <= rule 2
12 }
13 else
14 {
15     af = 9;              ac    = 9*A + Ba + D; //coded 9            // <= rule 2
16 }
```

3 State Machine Case Study

In the case study, a simplified sensor actuator state machine is used. The behavior of a sensor actuator chain is managed by control techniques and Mealy state machines. Acquisition and diagnosis of sensor signals is managed outside the state machine in the input management, whereas the output management is responsible for control techniques and for distributing the actuator signals. The input management processes the sensor values, generates an event, and saves them on a blackboard, a managed global variable.

The state machine reads the current state and the event from the blackboard, if necessary executes a transition, and saves the next state and the action on the blackboard. If a fault is detected, the blackboard is saved in a fault storage for diagnosis purpose. Finally, the output management executes the action. This is repeated in each cycle of the task.

The Safety Supervisor supervises the correct work of the state machine in the application software [4].

A simplified state machine was implemented in the Safely Embedded Software approach. The two classical implementation variants given by nested switch statement and table driven design are verified. The runtime and the file size of the state machine are measured and compared with the non-coded original one.

As result a rise up factor for runtime of 4.56 and a rise up factor for filesize of 5.75 for the nested switch variant was measured. The table driven variant was also implemented and measured. However, the table driven variant takes a factor of 2.5 more compared to the nested switch statement one. This growth is caused in the additional instructions of state table management that is also safeguarded by SES, e. g. updating of dynamic signatures.

The executable code was generated for a test computer with an Intel(R) Pentium(R) 4 CPU 1.60 GHz processor. The used integrated development environment Dev-C++ 4.9.9.2 covers the compiler "GCC (MinGW) 3.4.2" and the linker "GNU ld version 2.15.91 20040904 Supported emulations: i386pe".

4 Conclusion

Safely Embedded Software gives a guideline to diversify application software. The fault detection is realized locally by SES. Whereas the fault reaction is globally managed by a Safety Supervisor.

An overall safety architecture comprises diversity of application software realized with the nine rules of Safely Embedded Software, remaining hardware diagnosis, and hardware redundancy e. g. a clock time watchdog. Moreover, environmental monitoring (supply voltage, temperature) has to be provided by hardware means. Temporal control flow monitoring needs control hooks maintained by the operating system or by specialized basic software. Classical RAM test techniques can be replaced by SES since fault propagation techniques ensures the propagation of the detectability up to the check. A system partitioning is possible, the comparator function might be located on another ECU. In this case, a safety protocol is necessary for inter ECU communication.

An application of SES can be motivated by the model driven approach in the automotive industry. State machines are modeled with tools like Matlab or Rhapsody. A dedicated safety code weaving compiler for the given tools is proposed. The intention is to develop a single channel state chart model in the functional design phase. A preprocessor will add the duplex channel and comparator function to the model. Afterwards, the tool based code generation can be performed to produce the desired C code. Either a safety certification [5] of the used tools will be necessary, or the assembler code will be reviewed.

This work will be published in a forthcoming paper.

References

1. Ehrenberger, W.: Software-Verifikation. Hanser, Munich (2002)
2. Douglass, B.P.: Safety-Critical Systems Design. i-Logix, Whitepaper
3. Forin, P.: Vital Coded Microprocessor Principles and Application for Various Transit Systems. IFAC Control, Computers, Communications, Paris, 79–84 (1989)
4. Hummel, M., Egen, R., Mottok, J., Schiller, F., Mattes, T., Blum, M., Duckstein, F.: Generische Safety-Architektur für KFZ-Software. Hanser Automotive 11, 52–54 (2006)
5. International Electrotechnical Commission (IEC): Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems (1998)
6. Leaphart, E.G., Czerny, B.J., D'Ambrosio, J.G., Denlinger, C.L., Littlejohn, D.: Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications. In: SAE World Congress, Detroit, pp. 1–16 (2005)
7. Torres-Pomales, W.: Software Fault Tolerance: A Tutorial, NASA, Langley Research Center, Hampton, Virginia (2000)

Safety Demonstration and Software Development

Jean-Claude Laprie

LAAS-CNRS – Université de Toulouse
7 avenue Colonel Roche
31077 Toulouse, France
laprie@laas.fr

Abstract. The paper reports about a study conducted for RATP, the utility organisation for public transportation in Paris and region.

RATP has developed since the mid eighties a mathematically formal approach for the development of safety-critical software, based on the B method.

The question raised, in the context of evolutions in software development, was:

> Is it possible to demonstrate the same level of safety without resorting to mathematically formal approaches?

In order to respond this question, several steps were considered: 1) reminding the infeasibility of quantifying safety-critical software, and its consequences on the development process, and on the system vision, 2) situating the current RATP approach with respect to other safety-critical domains, 3) examining and comparing alternate approaches for developing safety-critical software, 4) coming back to the RATP approach, for examining underlying assumptions.

The conclusion was the recommendation to pursue the mathematically formal development approach.

1 Objectives and Summary

RATP (Régie Autonome des Transports Parisiens) is the utility organisation for public transportation in Paris and region. RATP has established since the mid-eighties a mathematically formal approach for the development of safety-critical software, based on the B method [Abrial 1996, Abrial 2003]. The approach has been first defined and implemented for the SACEM system [Hennebert & Guiho 1993], and has then been applied to several generations of speed control systems, either for subways or for trains [Dollé et al. 2003, Chapront 1993]. It is noteworthy that SACEM, operationally deployed in 1989, was the first operational safety system in the world that had been subjected to formal verifications [Craigen et al. 1993].

The current RATP approach has reached maturity with the Meteor automatic subway line [Dollé et al. 2003], in the sense that, going beyond verification, the development has been mathematically formal. In the context of automating several

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 289–300, 2007.

lines of the Paris subway, the question discussed within RATP with respect to the evolution of software development approaches is:

> Is it possible to *demonstrate* the same level of safety without resorting to mathematically formal approaches for developing safety-critical software?

In order to respond to this question, we consider the following steps:

1) Reminding the infeasibility of quantifying the reliability of safety-critical software, which leads to approaches combining a) the development process for software reliability, and b) a system vision for safety demonstration.
2) Situating the current RATP approach with respect to other safety-critical domains.
3) Examining and comparing alternate approaches for developing safety-critical software.
4) Coming back to the RATP approach, for examining a) the underlying assumptions of the correctness of the specification and of the static data , and b) the structure of the development process.
5) Concluding in recommending to pursue the mathematically formal development approach.

2 The Infeasibility of the Reliability Quantification of Safety-Critical Software, and Its Consequences

Safety software for subways, and more generally for railway safety equipments, must satisfy the requirements of the SIL 4 level (Safety Integrity Level) of the CENELEC EN 50128 standard [CENELEC 2001]. The system failure rate for such a safety level must be lower than probabilities in the range 10^{-9} - 10^{-12}/h.

Demonstrating via 'zero-failure' tests, i.e., via experimentations with no failure observed, conformance to such requirements before operational life is practically infeasible. A general result of the probability theory (be it by classical frequentist courses of reasoning [Cho 1987, Howden 1987], or by Bayesian formulations [Littlewood & Wright 1997]) is that the number of tests for demonstrating a 10^{-n} rate is of the order of $n.10^{n}$ (the exact number is a function of the confidence interval in a frequentist formulation), be this rate expressed in continuous time (per hour) or in discrete time (per execution). Furthermore, this result assumes a total coverage of the input space, which is still more difficult to demonstrate, even when constraining the digital representation of the inputs to integers or fractional numbers. The input domain is defined by the possible variations of the inputs, either dynamic data, acquired by sensors, or static data, describing the environment.

It is also worth emphasising that numerous years in operation generally do not enable to demonstrate the conformance to safety requirements, because of the insufficient number of operational hours, even for largely deployed equipments in large fleets, whether failures occurred during operational life or not [Shoomann 1996]. For instance, if we estimate the cumulative operational duration of the SACEM system to 10^{7}h, and assuming that no failure due to software has been observed, we are led to a *measured* failure rate of 10^{-6}/h, thus largely higher than the requested failure rate, but without prejudging of the actual value, lower than 10^{-6}/h.

A classical approach for evaluating software reliability, based on the reliability growth consecutive to a progressive decrease of the failure intensity, is by its very nature unsuitable for safety software, since it is based on failures that are numerous enough for being statistically significant.

Finally, following a course similar to the usual practice for physical failures affecting hardware, i.e., evaluating a system reliability from the reliability of its components, is equally impractical for safety-critical software, be the software fault tolerant or not [Butler & Finelli 1993, Littlewood et al. 2001]. It is noteworthy that approaches aimed at improving software reliability predictions via explicitly incorporating past operational experience [Laprie 1992, Laprie & Littlewood 1992] do not pretend applicability to safety software.

The consequences of what has been presented in this section, i.e., the infeasability of the quantified evaluation of the reliability of safety software, are two-fold:

- 1) Emphasis put on the development *process*, which, in all cases, is aimed at producing fault free software.
- 2) *System* vision for demonstrating safety.

3 Situation of the RATP Current Approach

The combination process-system we have just mentioned can be found in the various safety-critical domains, e.g., beyond railway signalling, civil avionics or nuclear protection. We address briefly these domains in order to situate the adopted solutions with respect to essential characters of these domains. Such characters can be termed as cultural: diversity for civil avionics, in-depth defences for nuclear control, end-to-end control for railway signalling.

3.1 Civil Avionics

An essential character which can be found all along the civil avionics history is *diversity*, or dissimilarity. Regarding the flight control system, diversity means the combination of:

a) Resorting to different technologies for insuring redundancy, e.g., hydraulics sparing electro-mechanics before the advent of fly-by-wire; software, and often hardware, diversity since.
b) Different control channels for a given movement of the plane, thanks to control surfaces (ailerons, spoilers, flaps, rudder) that are split and redundant.

This combination of approaches exists in current fly-by-wire airplanes:

- software and hardware diversity at two levels for all Airbus generations since the A-320 [Brière & Traverse 1993], the software codes being automatically generated from detailed specifications,
- diversity of the compilers and of hardware for the Boeing 777 [Yeh 1998].

The development of safety-critical software is governed by the DO-178 standard [RTCA/EUROCAE 1992]. It is interesting to note that this standard does not grant diversity any specific advantage for certification, in the sense of accepting a lower criticality for diversified software than for non-diversified ones.

3.2 Nuclear Protection

In this domain, an essential character is in-depth defences, with successive confinement barriers of different technologies. In addition, operators are always present in the control loop, and, taking into account the inertia of the phenomena that come into play, have to observe a delay before reacting to an incident.

Considering the protection systems (i.e., the control of halting the fission), which are systems whose criticality is at the SIL4 level, approaches differ according to the countries where the protection systems are computerised. For instance, emphasis is put in France on producing fault free software [Remus 1982, Pilaud 1990, Nguyen & Ourghanlian 2003]. In the UK, the protection systems are diversified: a computerised primary protection system, and a hardwired secondary protection system, whose functionalities are less than the primary system ones [Hunns & Wainwright 1991].

Development of safety-critical software is governed by the IEC 880 standard [CEI 1986].

3.3 Railway Signalling — The RATP Approach

Railway signalling has two major functions: speed control and itinerary control. The essential character that was prevailing before computerisation is *intrinsic safety*, which, in system terms leads to *end-to-end control.* This end-to-end control can be implemented by a 'safety net' for the itinerary control as in the Elektra system [Kantz & Koza 1995]. For the speed control, RATP choose the *coded processor* approach [Forin 1989, Hennebert & Guiho 1993], where the end-to-end control is insured by a *signature* of the software execution flow, computed from the software source code, and which includes the processed data and the execution timing. An undetected error can be caused only by the production of a signature identical to the pre-computed one, against which protection is insured by the length of the coding key. The approach provides a *quantifiable* end-to-end control, and is free from any assumption about a) the hardware failure modes, b) the execution of the executive software supporting the execution of the application software, and c) the compiler. It however assumes a) the absence of faults in the application software, b) the correctness of the static data describing the environment, and c) the independence of the failures of the signature checking hardware and of the processor which executes the software. Confidence in the absence of software faults is based on the use of the B method for its development.

Development by the B method is mathematically formal from the specifications. It is based on logically rigorous reasoning by theorem proving, where main proofs relate to consistency checking at a given step (checking invariant preservation by a machine), and to the refinement checking (checking that a machine is a correct refinement of a more abstract machine) [Abrial 2003, Potet 2003]. Such a development brings a maximum confidence in the absence of faults in the produced software, because the development is based on *calculi* in mathematical logic. Such a development therefore brings, when compared with pure process approaches (control, monitoring and documentation of the process), a rigorous dimension about the product correctness. It can thus be said that the freedom from fault becomes *objective*, as opposed to pure process approaches, which can only bring a *subjective* confidence.

This 'objectivisation' of the confidence in the freedom from faults is based on assumptions pertaining to a) mathematical logic (decidability), to b) the practice of proving (interpretation of unsuccessful proofs), and to c) the correctness of the specifications and of their formalisation.

4 Alternate Approaches for Safety-Critical Software Development

We examine two alternate approaches in this section:

- approaches aiming at fault freedom by mathematically formal verification,
- approaches aiming at fault detection during execution by diversified software.

We do not consider traditional approaches, either purely informal, or without any detection means during execution.

4.1 Other Approaches of Mathematically Formal Verification

Without pretending being exhaustive, because of the richness of the studies in formal verification, such other approaches can be put in two broad classes: model checking and static analysis.

Model checking [Clarke & Wing 1996, Schnoebelen 1999] is based on representing the specifications via a transition system (finite state machine, Petri nets), and expressing the properties that must be satisfied in temporal logic. Checking that the properties are indeed satisfied is performed by enumerating the reachable states. Model checking is usually supplemented by automatic generation of the software code from the model. Taking into account the processed data leading to infinite or unbounded models, the model and the properties have to be simplified. Because of such simplifications, model checking does not alleviate from testing. Tests can be generated from the model upon which verification is based (especially exploiting the refutations), leading to the so-called 'formal testing' [Carrington & Stocks 1994, Callahan et al. 1996, Howden 1998, Hamon et al. 2004]. Theorem proving directly incorporates the processed data [Rushby 2000], and is not subjected to the above-mentioned limitation. The synchronous language approach [Benveniste & Berry 1991, Benveniste et al. 2003] can be classed as a model checking approach, although it occupies a specific position due to a) its focus on reactive systems, and to b) the synchrony constructs that underly the approach. It is worth mentioning in the context of this paper the approach developed by Union Switch & Signal [Profeta et al. 1996], which presents similarities with the RATP approach, since it is also based on a coded processor, but whose software is based on the Lustre synchronous language [Halbwachs et al. 1991].

Static analysis [Pilaud 1999, Nguyen & Ourghanlian 2003, Cousot & Cousot 2004] is performed on the software source code, and is aimed at detecting errors that would be produced at execution. Static analysis is thus aimed at supplementing, or at replacing, tests performed on the software source code.

A schematic representation of the confidence brought by these approaches as a function of the verification effort is given by Figure 1, adapted from [Rushby 2001]. In addition to the above-mentioned approaches, this figure also positions type control,

which can be termed as an 'invisible formal approach'. The figure illustrates an important distinction between the formal verification approaches that have been (briefly) surveyed and the formal development approach constituted by the B method: verification approaches are aimed at *uncovering and removing* faults that have been introduced during software production, whereas formal development aims at *preventing* fault introduction.

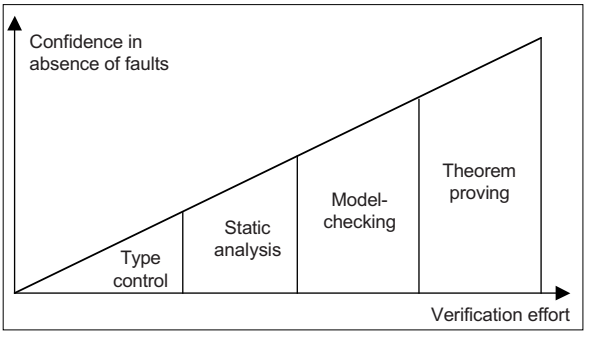


Fig. 1. The formal verification approaches

4.2 Diversified Software

Software diversity results from the production, by *separate development teams*, of functionally equivalent (since they satisfy common specifications) software systems. Software diversity aims at insuring *failure independence* of the versions or variants so produced, i.e., at avoiding that faults are simultaneously activated in the variants and produce similar errors, non distinguishable by comparators or voters. Although diversity covers the whole spectrum of fault tolerance techniques [Laprie et al. 1990], we restrict in the context of this paper to two-variant software, thus focusing on error detection via back-to-back comparison.

Diversity does not assume any development style, and especially whether verification is formal or not. Diversity can be free, assuming different choices performed by separate teams, or can be forced by imposing constraints aimed at favouring it, such as different formats for data structures, different programming languages.

Deploying two-variant software needs a two-processor hardware platform. Presence of residual development faults in processors [Avizienis & He 1999] leads to diversified processors, unless it is assumed that executing diversified variants on identical processors leads to different states of the processors, and thus enables assuming that residual development faults of the processors are not activated simultaneously. Although this may look as a reasonable assumption, analysis can hardly confirm it at the level requested for safety-critical systems.

A major difficulty with diversified software is the evaluation of the reliability gain, given that total failure independence cannot be assumed, because of a) the hard-core that the common specification is, and of b) the fact that faults can be correlated because of design difficulties brought about by some software components [Bishop et

al. 1986, Knight & Leveson 1986]. The various experimentations conducted in software diversity, including the latter two references, demonstrate reliability increases. However, these increases are not commensurate with the requirements of safety-critical systems at the SIL4 level. Reliability models [Arlat et al. 1990, Littlewood et al. 2001] take into account correlated faults, but the precise quantification of their influence when evaluating reliability comes up against the infeasibility reminded in section 2, and this is all the more true as there are very few published experimental data relative to operational diversified software [Lindenberg 1993].

It is important to emphasise in the context of this study that resorting to diversified software would mean abandoning the end-to-end control via the signatures associated to the coded processor, unless it is envisaged that both processors supporting the execution of the variants are coded processors. Although such a solution would lead to a very significant cost increase, the end-to-end control would then be performed via the comparison of the outputs delivered by the two variants, without however the possibility of performing a realistic probabilistic evaluation, as reminded above.

Finally, the presence of two variants executed by two processors for the on-board system would lead to a system failure rate higher than the current solution (even when accounting for the large overhead in memory size brought by the coded processor), thus penalising availability, and this independently of the obtained safety level.

5 Flashback on the RATP Approach

As announced in Section 1, we address a) two underlying assumptions, the correctness of the specifications and of the static data, and b) the structure of the development process.

5.1 Specification Correctness

The initial step of specification writing is, by essence, informal. It is thus this initial step that can benefit most of developments in terms of the so-called 'semi-formal' methods (in the sense of notations without formally defined semantics). Such approaches are already practised by RATP, with the SADT notation, supported by the ASA+ environment. Among recent developments in the domain of semi-formal methods, the current de facto standard modelling method that UML is cannot be ignored. RATP has indeed interest in UML. This is strengthened by works aimed at establishing a link between UML and B [Ledang & Souquières 2001, Snook & Butler 2003].

Important facets of specifications are the safety specifications, deduced from the behaviour of the controlled system. This step traditionally involves check-lists [Lutz 1996], and models and simulations of the behaviour in the presence of failures, especially via fault trees and state machines. It is worth mentioning in this respect:

- studies aimed at giving formal definitions of fault trees for removing their ambiguities, especially temporal ones [Hansen et al. 1998], as well as studies

aimed at establishing a relationship between automata and fault trees [Rauzy 2002],
- the possible resorting to models or simulations based on formal approaches, whose validation can then be performed by model checking [Bieber et al. 2002, Bohn et al. 2002].

5.2 Static Data Correctness

Static data constitute the Achilles tendon of any control system, as their verification can only be performed via reviews and inspections. In addition, they typically involve a significant memory size, possibly larger than the memory size necessitated by the programs. It is worth noting the special attention devoted to the derivation of the parameters and constraints, from the basic static data, that are used by the control programmes [Delebarre et al. 1999].

5.3 Structure of the Development Process

The structure of the development process of a safety-critical system is bound to be heavy and complex. Such heavy and complex structures stem directly from the recommendations of the standards, especially in terms of independence of the verifications and validations that have to be undertaken at the various steps of the development.

Giving up a formal development would not, in our opinion, lower the development costs, for two major reasons, common to both alternate approaches examined in section 4 (formal verification and diversity):

- reintroduction of unit tests, that are not necessary with the B method,
- effort necessary for removing faults more numerous than they are presently.

In addition, within a scope of constant total cost, that would lead inevitably to lower the effort devoted to specifications, thus going in the opposite direction of one of the widely recognised benefits of formal methods, i.e., enabling to devote an increased proportion of the effort to specification[1].

It is worth noting that, in the development of the information system of the air traffic control in the UK [Hall 1996], it has been estimated that using formal methods has led to decreased costs when compared to traditional, informal, approaches, although the extent of formal methods was then more limited than it is in the case of RATP.

A stumbling block of any process is the actual application of its guidelines and recommendations in practice, and as a consequence the related control means. From this viewpoint, resorting to mathematical formalism can only be favourable, due to the corresponding reduction of the ambiguities.

6 Conclusion

As announced in section 1, and argued in the paper, the recommendation has been to pursue the *mathematically formal development approach associated to the coded*

[1] Granting an increasing proportion of the development effort to specifications is a general trend in software development [NIST 2002].

processor, as this association provides a safety demonstration that is based on *weaker assumptions* than the other envisaged approaches.

The B method is particularly well adapted to safety applications of RATP, as it has initially been developed for those applications. The method has since been employed in other fields of application, and is currently a research topic in several countries [Bert 2003], and continues to evolve. As any mathematically formal approach, it suffers from difficulties relating either to the mathematics culture necessitated from its practicioners, or to the limits of the tools that implement it (as those tools have to guided interactively, they may lead to focus on their technicalities rather than on the subject of the proof).

The recommendation does not mean to stay in an immutable framework. Significant developments are still on going in formal methods (see for instance the virtual library <http://vl.fmnet.info>).

Acknowledgements

This study reported in this paper has been performed at the request, and with the support of RATP. The context was the choice of a development approach for the safety-critical control system for the automation of the Line 1 of the Paris subway. Another study was concurrently and independently performed [Caspi 2005]. Both studies reached the same conclusions and recommendations. The automation of Line 1 has been adjudicated to an industrial proposal which was offering a mathematically formal development approach.

I want to thank the following persons of RATP for their open-mindedness during the development of the study: Yves Ramette, deputy General Director of RATP, and his collaborators Didier Bense, Marcel Huertas, Loïc Pelhate, Jean-Pierre Riff, Jacques Valancogne.

References

[Abrial 1996] Abrial, J.R.: The B book — Assigning programs to meanings. Cambridge University Press, Cambridge (1996)

[Abrial 2003] Abrial, J.R.: B: passé, présent, futu (B: past, present, future). Technique et science informatiques 22(1), 89–118 (1993)

[Arlat et al. 1990] Arlat, J., Kanoun, K., Laprie, J.C.: Dependability modeling and evaluation of software-fault tolerant systems. IEEE Transactions on Computers 39(4), 504–513 (1990)

[Avizienis & He 1999] Avizienis, A., He, Y.: Microprocessor entomology: a taxonomy of design faults in COTS microprocessors. In: Weinstock, C.B., Rushby, J. (eds.) Dependable Computing for Critical Applications 7, pp. 3–23. IEEE CS Press, Los Alamitos (1999)

[Benveniste & Berry 1991] Benveniste, A., Berry, G.: The synchronous approach to reactive real-time systems. Proceedings of the IEEE 79, 1270–1282 (1999)

[Benveniste et al. 2003] Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91, 64–83 (2003)

[Bert 2003] Bert, D.: La recherche en B (The research in B). Technique et science informatiques 22(1), 129–130 (1993)

[Bieber et al. 2002] Bieber, P., Castel, C., Seguin, C.: Combination of fault tree analysis and model checking for safety assessment of complex systems. In: Proc. 4th European Dependable Computing Conference (EDCC-4), October 2002, Toulouse, France (2002)

[Bishop et al. 1986] Bishop, P.G., Esp, D.G., Barnes, M., Humphrey, P., Dahll, G., Lahti, J.: PODS — A project on diverse software. IEEE Trans. on Software Engineering SE-12(9), 929–940 (1986)

[Bohn et al. 2002] Bohn, J., Damn, W., Wittke, H., Klose, J., Moik, A.: Modeling and validating train system applications using Statemate and live sequence charts. In: Proc. Integrated Design and Process Technology (IPDT 2002) (June 2002)

[Brière & Traverse 1993] Brière, D., Traverse, P.: Airbus A/320/A330/A340 electrical flight controls — A family of fault-tolerant systems. In: Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 1993, pp. 616–623 (1993)

[Butler & Finelli 1993] Butler, R.W., Finelli, G.B.: The infeasibility of quantifying the reliability of life-critical real-time software. IEEE Trans. on Software Engineering 19(1), 3–12 (1993)

[Callahan et al. 1996] Callahan, J., Schneider, F., Easterbrook, S.: Automated testing using model-checking. In: Proc. 1996 SPIN Workshop, August 1996, Rutgers, NJ., USA (1996)

[Carrington & Stocks 1994] Carrington, D., Stocks, P.: A tale of two paradigms: formal methods and software testing. In: Proc. 8th Z User Meeting, Springer, Heidelberg (1994)

[Caspi 2005] Caspi, P.: Eléments pour le choix de méthodes de développement de systèmes logiciels critiques. Elements for selecting development methods of critical software systems, 2005, Rapport de recherche Verimag no TR-2005-17 (Novembre 2005) (In French)

[CENELEC 2001] Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems, EN 50128, European Committee for Electrotechnical Standardization (2001)

[Chapront 1993] Chapront, P.: Vital coded processor and safety related software design. In: Proc. Safety of Computer Control Systems 1992 (SAFECOMP'92), October 1992, pp. 141–145. Pergamon Press, Zurich (1992)

[Cho 1987] Cho, C.K.: Quality programming. In: Developing and testing software with statistical quality control, Wiley J. & Sons, Chichester (1987)

[Clarke & Wing 1996] Clarke, E.M., Wing, J.M.: Formal methods: state-of-the-art and future directions. ACM Computing Surveys 28(4), 626–643 (1996)

[Craigen et al. 1993] Craigen, D., Gerhart, S., Ralston, T.: An International Survey of Industrial Applications of Formal Methods, NIST report no. 93/626, p. 327

[Cousot & Cousot 2004] Cousot, P., Cousot, R.: Basic concepts of abstract interpretation, in Building the Information Society. In: Jacquart, R. (ed.) Proc. of the 18th IFIP World Computer Congress (WCC 2004), Toulouse, August 2004, pp. 359–366. Kluwer, Dordrecht (2004)

[Delebarre et al. 1999] Delebarre, V., Gallardo, M., Juppeaux, E., Natkin, S.: Validation des constantes de sécurité du pilote automatique de METEOR (Validation of the safety parameters of the automatic pilot of METEOR). In: Proc. 12th Int. Conf. Software and Systems Engineering and Applications (ICSSEA'99), December 1999, Paris, France, paper 13-4. (In French) (1999)

[Dollé et al. 2003] Dollé, B., Essamé, D., Falampin, J.: B dans le transport ferroviaire. l'expérience de Siemens Transportation Systems (B in railway transportation, the Siemens Transportation Systems experience). Technique et science informatiques, (In French) 22(1), 11–32 (1993)

[Forin 1989] Forin, P.: Vital coded processor: principles and applications. In: Proc. IFAC-GCCT, Paris, France, pp. 79–84 (1989)

[Halbwachs et al. 1991] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79, 1305–1320 (1999)

[Hall 1996] Hall, A.: Using formal methods to develop an ATC information system. IEEE Software 13(2), 66–76 (1996)

[Hamon et al. 2004] Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: Proc. 2nd IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM), Beijing, China, September 2004, pp. 261–270 (2004)

[Hansen et al. 1998] Hansen, K.M., Ravn, A.P., Stavridou, V.: From safety analysis to software requirements. IEEE Trans. on Software Engineering 24(7), 573–584 (1998)

[Hennebert & Guiho 1993] Hennebert, C., Guiho, G.: SACEM: a fault tolerant system for train speed control. In: Proc. 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 1993, pp. 624–628 (1993)

[Howden 1987] Howden, W.E.: Functional program testing and analysis. Mc Graw Hill (1987)

[Howden 1998] Howden, W.E.: Good enough versus high assurance software testing and analysis methods. In: Proc. 3rd Int. Symp. on high Assurance Systems Engineering (HASE'98), November 1996, pp. 166–175 (1996)

[Hunns & Wainwright 1991] Hunns, D.M., Wainwright, N.: Software-based protection for Sizewell B: the regulator's perspective. Nuclear Engineering International, 38–40 (September 1991)

[IEC 1986] Software for computers in the safety systems of nuclear power stations / Logiciel pour les calculateurs utilisés dans les systèmes de sûreté des centrales nucléaires, CEI 880 (1986)

[Kantz & Koza 1995] Kantz, H., Koza, C.: The Elektra railway signalling system: field experience with an actively replicated system with diversity. In: Proc. 25th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-25), Pasadena, CA, USA, June 1995, pp. 453–458 (1995)

[Knight & Leveson 1986] Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multi-version programming. IEEE Trans. on Software Engineering 12(1), 96–109 (1986)

[Laprie et al. 1990] Laprie, J.C., Arlat, J., Beounes, C., Kanoun, K.: Definition and analysis of hardware-and-software fault-tolerant architectures. IEEE Computer, Special Issue on Fault-Tolerant Systems 23(7), 39–51 (1990)

[Laprie 1992] Laprie, J.C.: For a Product-in-a-Process Approach to Software Reliability Evaluation. In: Proc. 3rd IEEE International Symposium on Software Reliability Engineering (ISSRE'92), October 1992, pp. 134–139. Research Triangle Park (1992)

[Laprie & Littlewood 1992] Laprie, J.C., Littlewood, B.: Quantitative assessment of safety-critical software: why and how? Communications of the ACM 35(2), 13–21 (1992)

[Ledang & Souquières 2001] Ledang, H., Souquières, J.: Modeling class operations in B: application to UML behavorial diagrams. In: Proc. 16th Annual Int. Conf. on Auitomated Software Engineering (ASE 2001), November 2001, pp. 289–296 (2001)

[Lindeberg 1993] The Swedish state railways experience with n-version programming. In: Redmill, F., Anderson, T. (eds.) Directions in Safety-Critical Systems, pp. 36–42. Springer, Heidelberg (1993)

[Littlewood et al. 2001] Littlewood, B., Popov, P., Strigini, L.: Modeling software diversity — A review. ACM Computing Surveys 33(2), 177–208 (2001)

[Littlewood & Wright 1997] Littlewood, B., Wright, D.: Some conservative stopping rules for the operational testing of safety-critical software. IEEE Trans. on Software Engineering 23(11), 673–683 (1997)

[Lutz 1996] Lutz, R.R.: Targeting safety-related errors during software requirements analysis. Journal of Systems and Software 34, 223–230 (1996)

[Nguyen & Ourghanlian 2003] Nguyen, T., Ourghanlian, A.: Dependability assessment of safet-critical system software by static analysis methods. In: Proc. IEEE/IFIP 2003 Int. Symp. on Dependable Systems and Networks (DSN 2003), San Francisco, CA, USA, June 2003, pp. 75–79 (2003)

[NIST 2002] The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST Planning Report 02-3 (May 2002)

[Pilaud 1990] Pilaud, E.: Some experiences of critical software development. In: Proc. 12th Int. Conf. on Software Engineering, Nice, France, March 1990, pp. 225–226 (1990)

[Pilaud 1999] Pilaud, D.: Vérification statique de programmes par interprétation abstraite: principes et expériences industrielles (Static verification of programmes by abstract interpretation: principles and industrial experience). In: Polyspace Technologies / Prod270999 (In French)

[Potet 2003] Potet, M.L.: Spécifications et développements structurés dans la méthode B (Structured specifications and development in B method). Technique et science informatiques (In French) 22(1), 62–88 (1993)

[Profeta et al. 1996] Profeta III, J.A., Andrianos, N.P., Yu, B., Johnson, B.W., DeLong, T.A., Guaspari, D., Jamsek, D.: Safety-critical systems built with COTS. Computer, 54–60 (November 1996)

[Rauzy 2002] Rauzy, A.: Modes automata and their compilation into fault trees. Reliability Engineering and System Safety 78(2), 1–12 (2002)

[Remus 1982] Remus, L.: Methodology for software development of a digital integrated protection system. In: presented at the EWICS TC-7 meeting, Brussels, January 1982, p. 19 (1982)

[RTCA/EUROCAE 1992] Software considerations in airborne systems and equipment certification, DO-178-B/ED-12-B, Requirements and Technical Concepts for Aviation/ European Organisation for Civil Aviation Equipment (1992)

[Rushby 2000] Rushby, J.: Theorem proving for verification. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, Springer, Heidelberg (2001)

[Rushby 2001] Rushby, J.: A practical introduction to formal methods. In: Workshop Utilisation of formal methods in dependable systems, IFIP 10.4, July 2001, Stenungsund, Sweden (2001), http://www.csl.sri.com/~rushby/slides/fmtutorial.pdf

[Shooman 1996] Shooman, M.L.: Avionics software problem occurrence rates. In: Proc. 7th Int. IEEE Symp. on Software Reliability Engineering (ISSRE'96), White Plains, NY, USA, November 1996, pp. 55–64 (1996)

[Schnoebelen 1999] Schnoebelen, P. (coord.).: Vérification de logiciels — Techniques et outils du model-checking (Software verification – Model-checking techniques and tools), Vuibert (In French) (1999)

[Snook & Butler 2003] Snook, C., Butler, M.: U2B Manual, University of Southampton, Electronics and Computer Science
http://www.soton.ac.uk/ cds/U2Bdownmoads/U2B_Manual.pdf

[Yeh 1998] Yeh, Y.C.: Dependability of the 777 primary flight control system. In: Iyer, R.K., Morganti, M., Fuchs, W.K., Gligor, V. (eds.) Dependable Computing for Critical Applications 5, pp. 3–18. IEEE Computer Society Press, Los Alamitos (1998)

Improving Test Coverage for UML State Machines Using Transition Instrumentation

Mario Friske and Bernd-Holger Schlingloff

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
{mario.friske,holger.schlingloff}@first.fraunhofer.de

Abstract. We discuss the problem of generating test suites from UML state machines and present a method to extend the capabilities of existing automated test case generators. Current tools provide only a limited coverage for different testing objectives. We argue that a better coverage can be achieved by instrumenting transitions, and performing an appropriate pre- and postprocessing. We describe the necessary enhancements of the UML model and demonstrate our method on a simple example. We further report on an industrial case study where we successfully applied our method for generating a validation test suite for a safety-relevant communication protocol.

1 Introduction

Testing is one of the most time-consuming tasks in the development of complex reactive systems. Thus, it is highly desirable to obtain as much tool support as possible. In code-based testing, the tester derives test sequences from the actual program code of the implementation. Code-based testing has some major drawbacks: First, mere code-based testing cannot ensure that the observed behaviour is equivalent to the intended behaviour, second, testing can only start when an actual implementation is available, which is usually rather late in the development process, and third, whenever the implementation is modified, the test suite has to be adapted anew. In contrast, specification-based testing focuses on required properties rather than on a particular implementation. The tester regards the implementation as a black box with hidden content. The development of specification-based tests can begin as soon as there is a requirements document (i. e., even before writing the first line of code).

The effectiveness of automated specification-based test case derivation methods largely depends on the specification formalism, which is used to denote system properties. Model-based testing assumes that system properties are represented in a formal or semiformal modelling language. Often, state-based formalisms such as finite state machines [1], Statecharts [2], UML state machines [3] or Stateflow models [4] are used. These formalisms are easy to use and have a well-understood semantics. Moreover, in a model-based development process diagrams can be used to derive an implementation by stepwise refinement. In such a context, the *system model* should represent the specification (i. e., focus on

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 301–314, 2007.

the functional requirements only). The *implementation model* is a refinement of the system model and takes implementation aspects such as data representation, efficiency, and scheduling into account.

For several modelling formalisms there exist code and test generation tools. Whereas for the generation of production code a detailed implementation model is necessary, test cases can already be generated from the system model. Various algorithms can be used for the construction of test cases. For example, one strategy is to use a breadth-first or depth-first search to generate all paths through the state graph up to a certain length. More elaborated strategies try to satisfy coverage criteria such as *All Transitions* [5] or *Modified Condition / Decision Coverage (MCDC)* [6] on the model. Even more advanced techniques use temporal logic model checking and fault injection to generate error traces, which can then be used as test sequences. A main topic in all of these approaches is to construct test suites which are both meaningful and manageable (i. e., provide a sufficient coverage and can be generated, executed and evaluated in reasonable time).

In the literature, many methods for generating test cases from state-based specifications have been proposed. Among them are methods based on FSM (e. g. [7,8]), extended FSM (e. g. [9,10]) or various variations of Statecharts (e. g. [11,12]). For practical application, there are a few commercial tools (Rhapsody/ATG [13], Conformiq [14], and Reactis [15]) and a number of experimental research tools (Agedis [16], Teager [17], TGV [18], TORX [19], AGATHA [20], and others) available. While research prototypes cover a wide range of coverage criteria, commercial tools only support a very limited selection of coverage criteria.

In this paper, we present two methods for improving the test coverage of test case generators, *Transparent Transition Instrumentation* and *Extended Transition Instrumentation*. We achieve a better coverage than the originally supported criterion *MCDC* by instrumenting transitions, and performing an appropriate pre- and postprocessing. Transition Instrumentation allows us to realize additional transition-based coverage criteria (e. g., *All n-Transition Sequences* [5]) without modifying the generator. We describe the necessary enhancements of the model and demonstrate our method on a simple example. We further report on an industrial case study where we successfully applied our method for generating a validation test suite for a safety-relevant communication protocol.

This paper is structured as follows: In the next section, we discuss capabilities and limitations of currently available test case generators for UML state machines. In Sect. 3, we present the two methods *Transparent Transition Instrumentation* and *Extended Transition Instrumentation*. Section 4 describes the industrial application of these methods. Finally, in Sect. 5 we draw some conclusions and give an outlook on future research work.

2 Test Generation from UML State Machines

We use the sample state machine shown in Fig. 1 to discuss abilities and limitations of current commercial test case generators for deterministic UML state

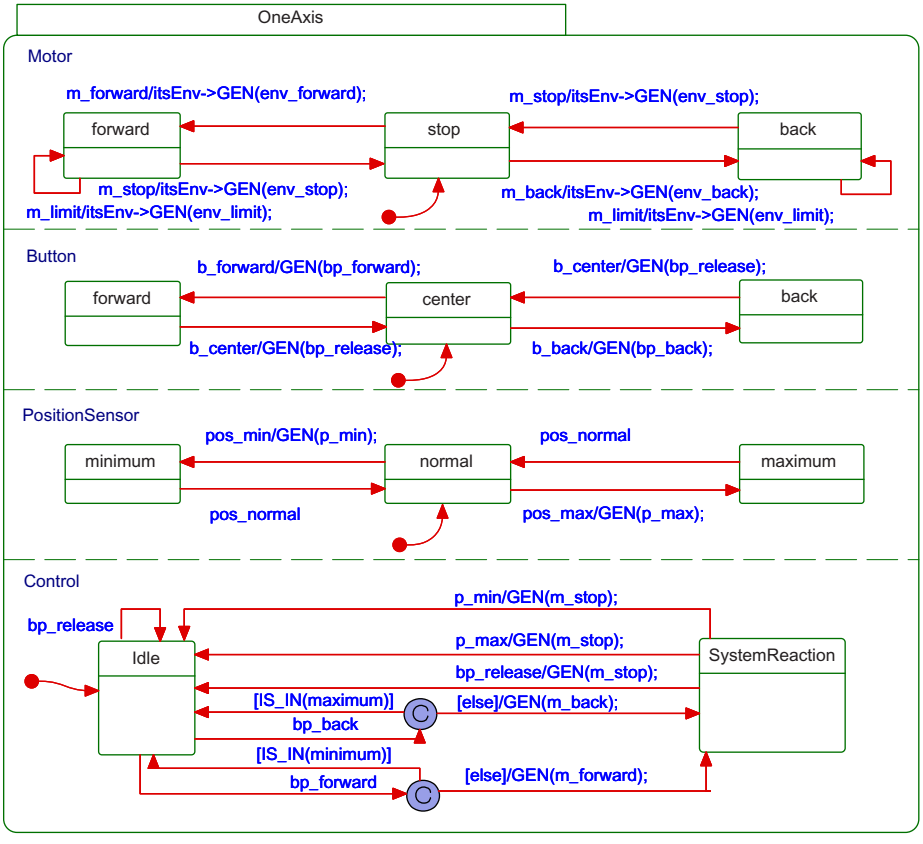


Fig. 1. State machine of simplified seat control (one axis)

machines. The state machine is a realization of one axis of the seat control specified in [21]. It allows moving a seat forward or backward by pressing a three-positions-button. When the button is pressed, the seat should move in the corresponding direction. The seat should stop moving when it reaches the final position. The seat should not move further in this direction before it has moved back into the opposite direction.

We manually created the model in a systematic approach following the modelling guidelines described in [22]. In contrast to the original guidelines, which suggest creating separate classes with their own Statechart for the control and for each sensor and actuator, we created just one class and one state machine with an orthogonal state comprising concurrent regions. As a modelling tool, we used Rhapsody in C++ [23], which utilizes C++ statements and macros as action language in UML state machines. For each sensor and each actuator, we modelled possible transitions between valid equivalence classes of input and output in a state machine. An additional state machine models the reactive

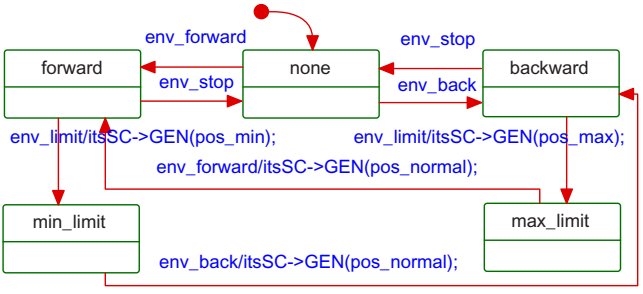


Fig. 2. Environment model of seat control

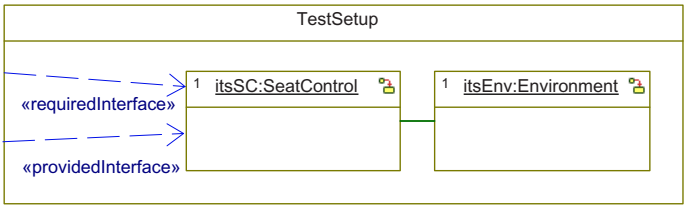


Fig. 3. Test setup including system model and environment model

behaviour of the controller. The parallel composition of these four state machines (see Fig. 1) models the overall behaviour of the seat control.

An *environment model*, which describes how the environment behaves if the system is integrated into it, complements the *system model*. The environment model covers only physical feedback but does not include behaviour of the user. The state machine of the environment model displayed in Fig. 2 defines possible transitions between equivalence classes of seat movement (i. e., the feedback between motor movement and resulting positions detected by the position sensor). Whenever the motor is switched on or off, an indicating event is sent from the seat control to the environment model. There it causes a transition between states indicating the current position of the seat. Transitions in the environment model define valid changes of positions. For example, only when the seat moves backward it will reach its maximum position.

As displayed in Fig. 3, the test setup consists of one instance `itsSC` of the class `SeatControl` which communicates with an instance `itsEnv` of the class `Environment`. The state machines displayed in Fig. 1 and Fig. 2 define the behaviour of the classes `SeatControl` and `Environment`, respectively. Stereotyped dependencies specify provided interfaces (inputs) and required interfaces (outputs) for test case generation.

We used Rhapsody's event generation mechanism [23] to realize the communication between these concurrent state machines. For example, the expression `m_forward/itsEnv->GEN(env_forward)` in the upper left corner of Fig. 1

signifies that the corresponding transition fires when the trigger event `m_forward` occurs and as a result, an event `env_forward` is generated and sent to the environment model `itsEnv`. Using such a combination of system and environment model allows modelling the interaction resulting from feedback between actuators and sensors.

For the following experiment, we use the commercially available test case generator ATG [13] to generate test cases from the combined model. The generator supports the coverage criterion *MCDC* [6] on the code generated from the state machine. *MCDC* means that every point of entry and exit in the program has been reached at least once, every condition in a decision in the program has taken on each possible outcome at least once, and each condition has been shown to affect that decision outcome independently [24]. This includes the coverage criterion *All Transitions* [5] which in turn includes the coverage criteria *All States* [5] and *All Events* [5]. *MCDC* is the generally acknowledged coverage criterion for white box testing of safety critical systems in avionics, as required by certain standards such as DO178B [24]. ATG does not explicitly handle the state machine's data space.

We convert the resulting test suite into simple sequences of events observable at the interfaces of the black box, which is the system under test. Then we free this test suite from duplicates and inclusions. Figure 4 shows the resulting test suite. It consists of six test cases which specify sequences of input events to be generated and output events to be observed. They are shown in six columns and should be read top-down.

`b_back`		`b_forward`		`b_forward`	`b_back`
`m_back`	`b_forward`	`m_forward`	`b_back`	`m_forward`	`m_back`
`m_limit`	`m_forward`	`m_limit`	`m_back`	`m_limit`	`m_limit`
`m_stop`	`b_center`	`m_stop`	`b_center`	`m_stop`	`m_stop`
`b_center`	`m_stop`	`b_center`	`m_stop`	`b_center`	`b_center`
`b_forward`		`b_back`		`b_forward`	`b_back`
`m_forward`		`m_back`			

Fig. 4. Test cases generated from Fig. 1 according to coverage criterion MCDC

Although the test suite shown in Fig. 4 satisfies the coverage criteria *MCDC*, *All States*, *All Events* and *All Transitions* in the specification, it is not sufficient for black box testing. In particular, the normal behaviour of the system is only tested in very short sequences. Many typical testing objectives for black box testing such as detecting missing or invalid states are not covered by this approach. For example, the test suite would not detect an erroneous implementation of the upper right transition with trigger event `m_stop` in Fig. 1 that leads to a different state than specified. The test suite does not contain a test case that shows the feasibility of moving the seat back again after reaching it's maximum position and subsequently moving it forward. Hence executing this test suite and observing input-output-conformance is not sufficient for ensuring that the transition reaches its target state.

Other conformance criteria and test case generation algorithms have been proposed in the literature. Amongst these are the transition tour method [25], the W-method [7] and UIO-method [8]. These methods construct test sequences to check the isomorphism or bisimilarity of finite IO-automata; that is, for each state, possible input and corresponding output of one automaton there must be an appropriate state in the other automaton which under the same input yields the same output and goes to an appropriate successor state. For black box testing these methods are only partially adequate. Firstly, they assume that both specification and implementation are given as finite automata. In our case, the specification is given as a UML state machine which may contain local variables and parameterized events, and the implementation is a black box. Secondly, these methods aim at the construction of test sequences of minimal length; during test execution, the implementation is reset after each sequence. In our case, reset is a costly operation which requires manual intervention and should be avoided. Thirdly, the goal of these methods is to construct test suites which are "complete" for some conformance criterion, such that the testing process must stop when all test cases are executed. In our case, we want to be able to adjust the coverage criterion such that the implementation can be exercised for a certain amount of time with well-defined test coverage.

In summary, for black box test generation from UML state machines the coverage which can be achieved by current commercial test case generators or other available tools is not sufficient. Hence, we are looking for ways to achieve a better coverage. Two options are apparent: One is to build a homemade test case generator. Another one is to modify a commercial test case generator.

Unfortunately, these two approaches are normally not applicable. On the one hand, it takes too much effort to build a test case generator. On the other hand, source code of commercial generators and extension APIs are mostly not available. Another aspect is that certification requires applying tools which are proven-in-use. Certification normally prevents engineers from using homemade or academic prototypes and forbids modifications of established test case generators. Hence, we have been looking for an alternative option for improving test coverage of commercial test generators. As a result, we have developed two methods that we present in the next section.

3 Improving Test Coverage

As discussed in the previous section, the coverage that can be achieved using currently available commercial test case generators is not sufficient for black box testing. Therefore, we developed an alternative approach that offers a third option. First, we present the overall approach and then we present two specific methods relying on this approach.

3.1 Extending Coverage Functions of Test Case Generators

Applying a coverage criterion to a model results in a set of test goals. A test generator generates a corresponding test suite by generating a test case for each

test goal. Test case generation might fail for some of these goals, for example resulting from unreachable code.

The basic idea of our approach is as follows: As depicted in Fig. 5, first, we enhance the state machine using a preprocessor by inserting additional elements to the model. The additional elements result in additional test goals for a test case generator leading to an enhanced coverage on the model. Then we generate test cases with an available test case generator, and finally we process the resulting test cases using a postprocessor. This allows us to generate test suites that satisfy more complex coverage criteria than the test case generator originally provides.

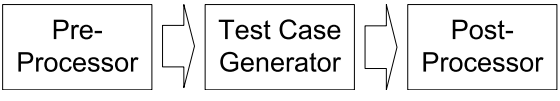


Fig. 5. Extending capabilities of test case generators through pre- and postprocessing

A test case generator provides a *generator coverage function* on a model. We augmented the input model by using an *enhancement function*. Then we generated test cases from the enhanced model. The test coverage of the resulting model is the composition of both functions, the *extended coverage function*.

In order to apply this general approach to our state-machine-based example shown in Fig. 1, we need to find a reasonable *enhancement function*. The *generator coverage function* of ATG [13] is *MCDC* on the generated code. *MCDC* includes coverage of all states, all transitions and all events in the model. Hence augmenting the state machine by including additional states, transitions, events or conditions would possibly result in larger test suites. Using additional states and transitions has the drawback of resulting in larger models while not directly leading to a larger coverage. Therefore, we did pursue the following two options for enhancing the state machine:

1. Additional variables and conditions in action code, and
2. additional events.

The first option can be used to implement counters. The main advantage of this option is transparency (i. e., no structural modification of the model is necessary and there are no extra events in the generated output). This is similar to the idea of *User Defined Test Goals* in Reactis [15], which are boolean expressions augmenting a test model.

The second option can be used to write additional information into event traces in test cases for further processing. Such information can be for instance information on the current allocation of global variables. It can also be used to pass other meta information (e. g., information about traversed paths) into test cases.

Based on these results we developed two methods for achieving extended coverage criteria: (1) *Transparent Transition Instrumentation*, resulting in test

cases, which contain longer sequences of events and comply with a given coverage criteria without changing the set of used events, and (2) *Extended Transition Instrumentation*, including additional events providing information for postprocessing. In the following subsections, we present these two methods in detail.

3.2 Transparent Transition Instrumentation

The test case generator generates test suites satisfying the coverage criterion *MCDC*. This means that for each condition in the action code two test cases are generated: in one the condition evaluates to true and in the other to false. Adding a C++-Statement to the action code of a transition that compares an additional variable with a given value results in two extra test goals for the test case generator. We can exploit this fact to implement a counter mechanism that allows sets of extra test goals resulting in generation of a desired sequence of transitions.

For a desired sequence of transitions, we use a counter variable and compare this variable with given values within condition statements and conditionally increment the variable. In other words, we add C++-Statements in the form `if(counter==n){counter++;}` to the action code of each transition in the sequence, where `n` is set according to the position of the transition within the sequence. These additional statements let a desired sequence of transitions become a goal for the test case generator. The mechanism can be exploited to force the test case generator to generate test cases that cover designated sequences of transitions and consequently to generate test suites that satisfy other test coverage criteria than *MCDC*.

Being able to generate specific sequences of transitions allows realizing all test case generation strategies relying on sequences of transitions. Sequences of transitions can be calculated based on the length of sequences as in the generation strategy *All n-Transition Sequences* [5], characterization sets as in the *W-method* [7], unique input/output sequences as in the UIO-method [8], or other criteria.

Most of these strategies based on sequences of transitions originally have been proposed for FSMs. Specific problems arise when applying these strategies to UML state machines which can be hierarchical and parallel. A strategy can be applied to either the entire state machine model with interleaved firing of transitions from all regions or to a single region. Calculation of transition sequences for complex hierarchical and parallel state machines is a non-trivial task and usually requires simulation of the state machine. Executing a transition sequence in a single region requires execution of the overall state machine and usually requires firing of several transitions in all regions.

Applying a transition-sequence-based strategy to one or more regions appears very eligible from the tester's perspective. Testers usually give more importance to certain aspects than to other aspects. Often a state machine has regions providing synchronization of other regions by communicating with them. For the previously presented state machine realizing a one-axis seat control, such central part is the region `Control` which processes inputs provided by the two sensors (regions `Button` and `PositionSensor`) and controls the actuator (region

`Motor`). The task of determining a transition sequence resulting from the overall state machine that contains the specific transition sequence in a region will be delegated to the test case generator.

In the following we explain *Transition Instrumentation* by applying the strategy *All n-Transition Sequences* [5] with $n = 2$ to the region `Control`. *All 2-Transition Sequences* requires that every specified transition sequence of length two has to be exercised at least once. The resulting test suite will achieve this coverage criterion on the region `Control` in addition to the previously achieved coverage criterion *MCDC* on the model.

Therefore, we create a more abstract alternative representation of this region with all transitions labelled using letters as shown in Fig. 6. Then we determine all sequences of length two from this representation: `ad`, `ae`, `af`, `ag`, `ah`, `bd`, `be`, `bf`, `bg`, `bh`, `cd`, `ce`, `cf`, `cg`, `ch`, `da`, `db`, `dc`, `ea`, `eb`, `ec`, `fd`, `fe`, `ff`, `fg`, `fh`, `gd`, `ge`, `gf`, `gg`, `gh`, `hd`, `he`, `hf`, `hg`, `hh`.

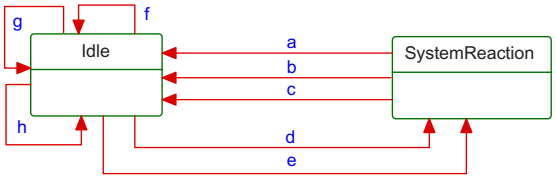


Fig. 6. Abstraction of seat control

Now we enhance the model with one counter variable of type integer for each sequence simply naming the variable after the sequence (e. g., `ad`). Then we instrument the transitions with extra code resulting in additional test goals required for achieving *MCDC*. The code is added using the following algorithm:

for each sequence of transitions
  for each transition within the sequence
    add `if(counter==n){counter++;}`

In the pseudo code `counter` stands for the integer counting variable and `n` for the number of transitions firing. For example applying this algorithm to determine the instrumentation code for exercising the transition sequence `ad` yields the following result: to the state machine presented in Fig. 1 we have to add to transition `a` the additional statement `if(ad==0){ad++;}`, and to transition `d` the statement `if(ad==1){ad++;}`.

In case that a transition has to fire twice or more, sequential order of additional statements is important, because it has to be ensured that the counter is not incremented to the maximum by a single firing. For example for achieving transition sequence `hh` the statement `if(hh==1){hh++;}` must appear before `if(hh==0){hh++;}` within the action code added to transition `h`.

After all transitions have been instrumented, generating test cases with the test case generator again will result in an extended test suite. If all transition

sequences required for achieving a strategy have been coded into the model, then the resulting test suite will also fulfil the corresponding coverage criterion. The extended test suite resulting from instrumenting the state machine in Fig. 1 according to *All n-Transition Sequences* with $n = 2$ is shown in Fig. 7.

As discussed at the end of Sect. 2, the previously generated test suite displayed in Fig. 4 cannot detect erroneous implementations of the upper right transition with trigger event `m_stop` in Fig. 1. The extended set of test cases shown in Fig. 7 can detect this error (e. g., by the test case in the lower left corner).

```
b_forward
m_forward
m_limit
m_stop
b_center
b_forward
b_center
b_forward
m_forward
b_center
m_stop
b_back
m_back
m_limit
m_stop

b_back
m_back
b_center
m_stop
b_back
m_back
b_center
m_stop

b_forward
m_forward
b_center
m_stop
b_back
m_back
m_limit
m_stop

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
b_center
m_stop
b_forward
m_forward
m_limit
m_stop

b_back
m_back
b_center
m_stop
b_back
m_back
m_limit
m_stop
b_center
b_forward
m_forward
m_limit
m_stop
b_center
b_forward

b_back
m_back
m_limit
m_stop
b_center
b_back
b_center
b_forward
m_forward
m_limit
m_stop
b_center
b_forward

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
m_limit
m_stop
b_center
b_forward

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
b_center
m_stop
b_back
m_back
b_center
m_stop
b_back
m_back
m_limit
m_stop
b_center
b_forward

b_forward
m_forward
m_limit
m_stop
b_center
b_back

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
b_center
m_stop
b_back
m_back
b_center
m_stop
b_back
m_back
m_limit
m_stop
b_center
b_back
```

Fig. 7. Extended set of test cases generated from Fig. 1

3.3 Extended Transition Instrumentation

The previously explained method *Transparent Transition Instrumentation* allows to generate a set of test cases achieving a given transition-sequence-based coverage criterion. The test generator generates one test case for each goal. Thus, the number of test cases contained in the test suite can get large while each test case is quite short. In certain situations, it is required to minimize the actual number of test cases while not reducing the coverage (i. e., to have a set of few, but long test cases). One approach to achieve this goal is concatenation of test cases.

Another aspect is that depending on the model and test strategy, applying *Transparent Transition Instrumentation* for very long sequences might fail because of limited resources such as CPU-time and memory.

For example, it is possible to include all sequences of transitions of length two presented in the previous section in one test case. One possible solution is the sequence `adaeafdbdcecffgghdagfebeahfhebfhhgecgdcdbgdbhech`. This can be done by concatenating test cases individually generated for each sequence of length two.

Usually concatenation of generated test cases is more complex than simply appending the sequences displayed in Fig. 7. In a UML state machine, the current configuration of the system is not just determined by the set of current states of all sub-automata, but also by the configuration of all global variables. Hence, the concatenation of fragments cannot be done by overlapping transitions only. Such overlapping without regarding global variables might result in invalid traces. For concatenating two test cases, we must ensure that at the concatenation point the state machine has the same configuration in both test cases.

In order to take into account information about global variables during concatenation, we introduce a new event with one parameter corresponding to each global variable. We generate this event on each transition that is a potential concatenation point. If the state machine contains concurrent regions, then for each concurrent region we add one more parameter to this event. Each of these parameters represents the current state of one region as enumeration. We also introduce another type of event to record information about subsequences of events contained in this test case. The postprocessor reads both events and uses this information during test case concatenation. After the concatenation is finished, the postprocessor removes all additional events.

The sample UML state machine that we introduced in Sect. 2 does not include global variables but concurrent regions. In order to generate one sequence that covers *All n-Transition Sequences* with $n = 2$ for the region `Control`, first we generate one valid sequence[1], see above, that satisfies this criterion on the abstraction shown in Fig. 6. Then we instrument the model with counters using *Transparent Transition Instrumentation* as discussed in the previous section. Furthermore, we add the action code for generating additional events providing information about the current state of all concurrent regions.

Then we generate test cases using the test case generator and subsequently concatenate the test cases using our postprocessor. Therefore, we pass the previously generated string representing the valid target sequence as parameter to the postprocessor. We realized the concatenation by recursively selecting a test case that contains the next required transition sequence, checking if global configurations of the state machines match, cutting out the corresponding sequence of events, and pasting it to the tail of the concatenated test case. How to ensure matching configurations will be discussed at the end of the next section.

[1] Here we do not consider the initial transition leading into state `idle`. Inclusion of the initial transition would require a minimum of five test cases for achieving *All n-Transition Sequences* with $n = 2$.

4 Industrial Application

We used this approach within an industrial research project to generate test cases for the slave device for an automation protocol. Two state machines, one for the master device and one for the slave device, details of which are under NDA, specified the protocol. First, we translated the slave's Statechart-like specification with pseudo action code into an executable UML state machine with C++ action code. Then, we analysed the protocol and built an additional abstraction of the state machine of the slave. The original Statechart consisted of 8 states and 15 transitions, comprising two cycles, each consisting of three states. We could create an abstraction as depicted in Fig. 8. One cycle was the normal operation cycle without occurrence of faults and the other cycle was the fault handling cycle.

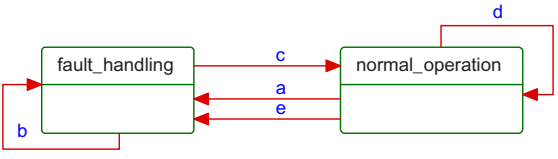


Fig. 8. Abstraction of automation protocol

For the automated generation of test cases we defined three coverage criteria:

CC_1**:** *MCDC* of the Code generated from the state machine.
CC_2**:** All sequences of length n from the abstraction of the state machine.
CC_3**:** Concatenations of m sequences obtained by CC_2 in random order.

Coverage criterion CC_1 was directly supported by the test case generator. We could achieve coverage criterion CC_2 by applying our method *Transition Instrumentation*. To this end, we calculated all possible sequences of length $n = 3$ from the abstraction in Fig. 8. We used these sequences to instrument the transitions of the state machine, as described above. Then, we generated a test suite satisfying CC_2 from it.

For achieving coverage criterion CC_3 we applied our second method *Extended Transition Instrumentation*. To this end, we extended the instrumentation for achieving CC_2 by adding extra events both for marking transitions and for writing the state of all global variables. After the test cases were generated we used our postprocessor to extract all fragments corresponding to the sequences of length $n = 3$. In order to extract fragments the postprocessor evaluated all extra events that have been added for marking sequences.

Then we calculated valid sequences of m transitions from the abstraction in random order and tried to concatenate the fragments accordingly. Comparing of global state machine configurations has shown that a simple concatenation of sequences was not always possible because of conflicting configurations. We could solve this problem by concatenation of sequences of length $n = 3$ with an overlapping of two transitions.

5 Conclusions and Further Work

We presented a method to improve the coverage capabilities of specification-based automated test generators. Whereas our considerations have been largely driven by particular application needs, there is the potential of extending them to a more abstract level.

We implemented a preprocessor which calculates sequences of transitions, determines counter variables and calculates corresponding instrumentation statements. Currently, we manually insert results of the calculations into the model. Although this is a relatively easy task, a further enhancement would be automating these steps (i. e., export the model using XMI [26], conduct an automated analysis of the model, calculate counter variables and instrumentation statements, instrument the model, reimport the model).

A further extension would be the realization of other test enhancement algorithms within our framework. We explained *Transition Instrumentation* using UML state machines, but the underlying principle is not restricted to transitions in state machines. The principle can be applied in all situations where a functional dependency between *generator coverage function*, *enhancement function*, and *extended coverage function* can be found.

A more fundamental question, which is tackled by our work, is the definition of appropriate numerical coverage criteria for specification-based testing. Currently, there are no generally acknowledged criteria for accrediting such test suites in safety-critical systems. By giving a possibility to experiment with different algorithms, our work can help in establishing such quantitative values for different safety layers.

References

1. Gill, A.: Introduction to Theory of Finite-state Machines. McGraw-Hill Education, New York (1962)
2. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
3. Object Management Group: Unified Modeling Language: Superstructure, version 2.0 (formal/05-07-04) (2005)
4. The Mathworks: Stateflow, http://www.mathworks.com/products/stateflow/
5. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Object Technology Series. Addison-Wesley, Reading (1999)
6. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA (2001)
7. Chow, T.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering SE-4, 178–187 (1978)
8. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Computer Networks and ISDN Systems 15(4), 285–297 (1988)
9. Wang, C.J., Liu, M.T.: Generating test cases for EFSM with given fault models. In: INFOCOM, pp. 774–781 (1993)

10. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Trans. Softw. Eng. 30(1), 29–42 (2004)
11. Hong, H.S., Kim, Y.G., Cha, S.D., Bae, D.H., Ural, H.: A test sequence selection method for statecharts. Software Testing, Verification and Reliability 10(4), 203–227 (2000)
12. Gnesi, S., Latella, D., Massink, M.: Formal test-case generation for UML statecharts. In: ICECCS '04: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS'04), pp. 75–84. IEEE Computer Society Press, Washington, DC, USA (2004)
13. I-Logix: Rhapsody Automatic Test Generator, Release 2.3, User Guide (2004)
14. Conformiq Software Ltd.: Conformiq test generator, http://www.conformiq.com/
15. Reactive Systems Inc.: Reactis, http://www.reactive-systems.com/
16. Hartman, A.: Agedis final project report. Technical report, AGEDIS Consortium (2004)
17. Santen, T., Seifert, D.: TEAGER - test automation for UML state machines. In: Biel, B., Book, M., Gruhn, V. (eds.) Software Engineering. LNI., GI, vol. 79, pp. 73–84 (2006)
18. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK, pp. 348–359. Springer, Heidelberg (1996)
19. Tretmans, J., Brinksma, E.: Côte de resyste – automated model based testing. In: Schweizer, M. (ed.) Progress 2002 – 3^{rd} Workshop on Embedded Systems, Utrecht, The Netherlands, pp. 246–255. STW Technology Foundation (October 24, 2002)
20. Lugato, D., Bigot, C., Valot, Y., Gallois, J.P., Gerard, S., Terrier, F.: Validation and automatic test generation on UML models: the AGATHA approach. Journal of Software Technology Transfer (2004)
21. Houdek, F., Paech, B.: Das Türsteuergerät - eine Beispielspezifikation. IESE-Report Nr. 002.02/D, Fraunhofer IESE (2002)
22. Denger, C., Kerkow, D., von Knethen, A., Medina Mora, M., Paech, B.: Richtlinien - Von Use Cases zu Statecharts in 7 Schritten. IESE-Report Nr. 086.02/D, Fraunhofer IESE (2002)
23. I-Logix: Rhapsody in C++, Version 6.0, User Guide (2004)
24. RTCA: DO-178B, Software considerations in airborne systems and equipment certification (1992)
25. Naito, S., Tsunoyama, M.: Fault detection for sequential machines by transition tours. In: Proccedings of the 11th. IEEE Fault Tolerant Computing Symposium, pp. 238–243. IEEE Computer Society Press, Los Alamitos (1981)
26. Object Management Group: XML Metadata Interchange (XMI) Specification. OMG (2003), http://www.omg.com/

Verification of Distributed Applications

Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan

German Research Center for Artificial Intelligence (DFKI GmbH)
Saarbrücken, Germany
{langenstein,nonnengart,rock,stephan}@dfki.de

Abstract. Safety and security guarantees for individual applications in almost all cases depend on assumptions on the given context provided by distributed instances of operating systems, hardware platforms, and other application level programs running on these. In particular for formal approaches the problem is to formalize these assumptions without looking at the (formal) model of the operating system (including the machines that execute applications) in all detail.

The work described in the paper proposes a modular approach which uses histories of observable events to specify runs of distributed instances of the system. The overall verification approach decomposes the given verification problem into local tasks along the lines of assume-guarantee reasoning.

As an example the paper discusses the specification and implementation of the SMTP scenario. It shows in detail how this methodology is utilized within the Verification Support Environment (VSE) to verify the SMTP server part.

1 Introduction

The theory developed in the following aims at a modular approach for the specification and verification of concurrent systems with heterogeneous components.

Concurrency typically results from the actual parallel execution of independent systems and the abstraction from a concrete scheduler within the context of a given platform.

Like the systems themselves their formal models will consist of various types of components specified by different types of state transitions systems. In the composed (global) system the components interact with each other by certain communication mechanisms.

The instantiation of the general approach considered in this paper is taken from the context of the Verisoft project where a pervasive formal model of a distributed collection of hardware-software platforms with application level programs running on each of these was established, [1].

It will be argued that for behavioral properties and in particular safety and security guarantees it is not appropriate to work directly on such models. Instead we propose to provide (possibly different) views by using traces of *observable events* that according to a given view are attached to steps of computations (or runs) of the system model. Since for a state in a run we collect all events that have

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 315–328, 2007.

happened so far we call these lists (of events) *histories*. The behavior of the global systems as well as that of single components will be specified by sets of histories thereby abstracting from the local state spaces of the various components and the local computations taking place on these. Like an input output specification for a sequential piece of of software sets of histories describe the concurrent, typically nonterminating computation of a global system or component.

In particular event traces defined by a certain view on a given model provide an appropriate interface for an inductive analysis of cryptographic protocols, [2,3] or an information flow analysis [4].

Our approach is modular in the sense that the task of verifying a history specification against runs of the global system can be decomposed into local verification tasks for its components. Following the general assume-guarantee approach, see [5] for comprehensive discussion of these approaches, for each event specified in a history there is exactly one component which may *violate* the specification at this point. This is the component considered to be *responsible* for the event. Events in a history a particular component is not responsible for are considered as assumptions of the component w.r.t. its environment.

For a history specification the components that are possibly restricted by it can easily be derived. For an operating system as considered in Verisoft this means that a *formal manual* would consist of a specification that only restricts the behavior (of instances) of the operating system itself. Having established this restriction locally it can be combined at the specification level with results, lets say for the SMTP server and client to derive desired properties of e-mail exchange.

The method proposed here proceeds in the following basic steps:

1. Define *events* and associate them with steps given by the transition R_g of a global system model.
2. Decompose R_g into *local transition relations* R_c for each component c.
3. Take R_c out of the system context by defining a *transformation* into $\tilde{R}_c$ which operates on histories.
4. Prove a theorem that establishes the faithful *simulation* of R_c by $\tilde{R}_c$.
5. Prove the local correctness of a component c w.r.t. to a given specification by using $\tilde{R}_c$. The simulation theorem guarantees that the correctness of R_g follows from the local results.

Note that the simulation theorem is about a transformation *tilde* which can be applied to certain type of transition systems. Proving it once and for all then allows for a local verification of all concrete instances. In our Verisoft application these instances are given by *programs* written in (a subset of) C.

In the next section we describe the general method as well as its instantiation in Verisoft more formally. Section 3 provides the transformation *tilde* for C programs and the events introduced before in section 2. Based on the specification of the SMTP scenario in Verisoft in section 4 we show how an efficient, purely sequential verification technique can be applied to prove the correctness of C programs with system calls. In the concluding remarks we in particular outline how this technique may be embedded into a (full) temporal framework.

2 Modular Verification of Distributed Systems

In the following we describe the method in general terms before we turn to the actual instantiation use din Verisoft.

2.1 Events and Histories

Let $\mathcal{S} = (S, R, I)$ be the global transition system where S is the global state space, $R \subseteq S \times S$ the global transition function, and $I \subseteq S$ the set of initial states. Typically the systems will not be considered to terminate and therefore we do not include final states here.

For a decomposition of $\mathcal{S}$ we fix a set C of indexes (or selectors) for components. Given $s \in S$ and $c \in C$ let $comp(s, c) = s_c$ be the local state of component c. Let S_c be the state space of component c, i.e. $s_c \in S_c$.

Since a transition in a component may depend on the states in or even the transitions of other components the local transition relations R_c cannot simply be $R_c \subseteq S_c \times S_c$. Instead a transition in component c may (sometimes) depend on transitions (not only states) in other components. Hence we have

$$R_{c_0} \subseteq \Pi((S_{c_1} \times S_{c_1}) \mid c_1 \in C_{c_0} \subseteq C - \{c_0\}) \times S_{c_0} \times S_{c_0} \ .$$

We have used C_{c_0} for the set of components c_0 depends on in the sense that the transitions of these components provide enough information.

The local transition system for c is then given by $\mathcal{S}_c = (S_c, R_c, \{s_c \mid s \in I\})$.

For a decomposition we expect that the global system is completely determined by the local ones, i.e.

$$sRs' \leftrightarrow \forall c_0 \in C.((((s_{c_1}, s'_{c_1}) \mid c_1 \in C_{c_0}), s_{c_0}, s'_{c_0}) \in R_{c_0})$$

To define a view on $\mathcal{S}$ with each such step $(s, s') \in R$ we associate a (small) list of (communication) events $[e_0, \ldots, e_{n-1}]$, each of the form $e = mk\_ev(c_0, c_1, m)$, where the component c_0 is the so-called *sender*, c_1 is the so-called *recipient* and m is the *message* to be sent. Each event $e = mk\_ev(c_0, c_1, m) \in [e_0, \ldots, e_{n-1}]$ results from a local state transition $(s_{c_0}, s'_{c_0}) \in R_{c_0}((s_{c_1}, s'_{c_1}) \mid c_1 \in C_{c_0})$. The corresponding message $m = mes(s_{c_0}, s'_{c_0})$ describes in an abstract and state independent way data that are made available by component c_0 to some other component c_1 where $c_0 \in C_{c_1}$. Note that the information given by m does not necessarily influence the transition $(s_{c_1}, s'_{c_1}) \in R_{c_1}$. It can also be the case that only some enabling condition for a transition in c_1 becomes true by this step in c_0 but the actual transition is carried out later.

We will not impose general rules on the assignment of events to steps of the global system. Instead a simulation theorem has to be proved for each decomposition and the associated assignment of events.

Let $\bar{s} = s_0, s_1, s_2, \ldots$, where $s_i R s_{i+1}$ is a run of the global system. Then for each k the *history* observed so far is the concatenation of all (small) lists of events, i.e. $hist(\bar{s}, k) = evts(s_0, s_1) \circ evts(s_1, s_2) \circ \ldots \circ evts(s_{k-1}, s_k)$.

For a fixed data type of events E_t the behavior of systems (both local and global) is specified by a unary predicate H on E_t^*. A system $\mathcal{S}$ *satisfies* H, written $\mathcal{S} \models H$ iff for all runs $\bar{s}$ of $\mathcal{S}$ and all k we have $hist(\bar{s}, k) \in H$. If $\mathcal{S}$ *violates* H in a run $\bar{s}$, then there is a unique component c that (in $\bar{s}$) violates H first by an event sent (or generated) by c. Let $\mathcal{S} \models_c H$ denote the fact that in any run $\bar{s}$ of $\mathcal{S}$ H will not first be violated by c. Clearly $\mathcal{S} \models H \leftrightarrow \forall c \in C.\mathcal{S} \models_c H$.

2.2 Verification by Replay and Extend

To verify a (history) specification H local to a given component $c \in C$ we uniformly transform (S_c, R_c) into $(\tilde{S}_c, \tilde{R}_c)$, where $\tilde{R}_c \subseteq ((E\_t^* \times E\_t^* \times \tilde{S}_c) \times (E\_t^* \times E\_t^* \times \tilde{S}_c))$.

The modified transition system in addition to states $\tilde{s}$ operates on two histories the current input and output history. The current input will always be a final section of the initial input. The computation stops when the input history becomes empty. The underlying idea is that whenever information is needed from the environment the (current) input history is scanned for suitable events providing information intended for c. Events actively sent (generated) by c during the replay have to coincide with the ones given by the input history. Otherwise due to a possible overspecification we have chosen a wrong input history. In this case we stop by appending the remaining input history to the (current) output thereby reconstructing the original input history. In some cases we will not only consume and reconstruct the complete (initial) input history but add some *new* c-events at the end. This is the *extend* case. Altogether we expect

$$
\begin{aligned}
&((h_0, [\,], \tilde{s}_0)\ \tilde{R}_c^*\ ([\,], h_f, \tilde{s}_f) \rightarrow \\
&h_f = h_0 \vee \exists h_1.(h_f = h_0 \circ h_1 \wedge \forall e \in h_1.get\_sender(e) = c)
\end{aligned}
$$

So far we have not established any relation between R_c (as part of global runs) and $\tilde{R}_c$. The simulation theorem expresses the fact that $\tilde{R}_c$ faithfully simulates initial segments of computations of R_c in the context of global runs.

$$
\begin{aligned}
&(s_0 R^* s \wedge hist(s) = h_0 \circ h_1 \\
&\wedge get\_sender(lst(h_0)) \neq c \wedge \forall e \in h_1.get\_sender(e) = c) \\
&\rightarrow \\
&\exists \tilde{s}_f.(h_0, [\,], init_c(s_0))\ \tilde{R}_c^*\ ([\,], h_0 \circ h_1, \tilde{s}_f)
\end{aligned}
$$

This means that in a situation where there is enough information about the past the modified (local) system will extend the history in the same way as it happens in real system context. Note that this is a general theorem basically depending (only) on the definition of *hist* and the transformation of $\mathcal{S}$ into $\tilde{\mathcal{S}}_c = (\tilde{S}_c, \tilde{R}_c)$.

Using $\mathcal{S}_c \models H$ for the fact that $\tilde{\mathcal{S}}_c$ preserves H we get $\mathcal{S}_c \models H \rightarrow \mathcal{S} \models_c H$ as the final soundness result for our decomposition.

2.3 The Verisoft Model

In the Verisoft project a formal model of an operating system was developed and verified with respect to its implementation, [6]. Application level programs run on (abstract) machines that are part of the model. They interact by a RPC like mechanism and access resources (of the OS) by external calls. Here we are interested in verifying *concrete* such programs given properties of the system environment.

A model of a distributed system consists of instances of (the same) system and an abstract network component connecting them. Each system is identified by a unique *network address* $na \in Na$. It basically consists of two parts. The (instance of the) *operating system* provides resources like input-output devices, a file system, and sockets for the communication over the network component. A number of *processes* each of them being identified be a *process identifier* $pid \in Pid$ is given by an abstract execution mechanism (machine) for certain kinds of application-level programs. In this paper we are interested in applications implemented in C0, the subset of C considered in Verisoft.

The configuration of a C0 machine $conf =< prog, mem > \in Conf$ consists of a program part $prog$ and a memory mem. In each step $< prog, mem > \; R_{C0} \; < prog', mem' >$ depending on the first statement of $prog$ the memory is updated and the computation proceeds with the continuation $prog'$ of the first statement in $prog$.

From point of view of an application programmer the system context consists of all operating systems and the network connecting them. Hence we consider this complete context as a single component in our decomposition. It will be denoted by the selector (symbol) sc. The states of this component $s_{sc} \in S_{sc}$ are made up of the states of all instances of the operating system and the state of the network. Processes are identified with pairs $p = mk\_proc(na, pid)$. Altogether we have $C = \{mk\_proc(na, pid) \mid na \in Na, pid \in Pid\} \cup \{sc\}$. For a process p the local state is the configuration $conf_p$ of the corresponding C0 machine, i.e. for all c we have $S_c = Conf$.

The view we have chosen is depicted in Figure 1.

The communication between user programs and the surrounding operating system (instance) is by so called *external calls*. External calls $c_{ext}(\bar{\tau} : \bar{z}, res)$ take the same syntax as ordinary function calls. We use $\bar{\tau}$ as a sequence of value parameters and $\bar{z}, res$ as a sequence of return parameters.

Whenever a system call is reached, i.e. for $conf_p \; R_{C0} \; conf'_p$ we have $conf \neq conf'_p =< r := c_{ext}(\bar{\tau} : \bar{z}); prog_{cont}, mem' >$, the normal execution as given by the (small step) semantics $R_{C0} \subseteq Conf \times Conf$ is interrupted (stopped).

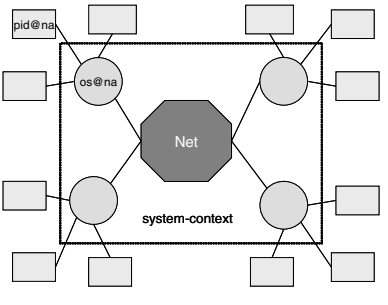


Fig. 1. Verisoft Model

With these steps we associate events of the form $mk\_ev(p, sc, m)$ where the message m encodes the particular call given by c_{ext} and the values of the parameters $(\bar{\tau} : \bar{z})$ in mem'. For a call of socket read `socket_read(sid : length, buffer, ec)` the corresponding message will by $Sread(sid, length)$ where $Sread$ is a constructor symbol for an abstract data type and $sid, length$ are the values of the programming variables `length, buffer` in mem'. They indicate the socket and the length of the string to be read.

To model the return of external calls the standard C0 machines have to be extended by steps where the resulting configuration is (also) determined by a transition $s_{sc}\ R_{sc}\ s'_{sc}$, that is we have $R^{ext}_{C0} \subseteq (S_{sc} \times S_{sc}) \times Conf \times Conf$. If an external call is processed by the system context leading to a step $s_{sc}\ R_{sc}\ s'_{sc}$ then the information intended for process p will be written to the return parameters of. For the transition in p this leads to a new memory mem'' and a new program $prog_{cont}$ to be executed further. The event we associate with these steps is of the form $mk_ev(sc, p, m)$ where the message m represents the return information. For example a successful call of socket read the message will be $Succ\_sread(length, buffer)$ where $length$ indicates the elements in the fixed length array $buffer$ that have actually been read. These values uniquely determine the values of the result parameters after return of the external call.

3 Application Level Programs

In this section we describe the construction of $\tilde{R}_{C0}$ out of R_{C0}. This will be done not be defining a new kind of machines but be transforming the C0 program π with external calls that is executed by a component into a program $\tilde{\pi}$ that takes histories as input and produces histories as output but uses only standard function calls. Since histories describe initial segments of nonterminating behaviors the new program is intended always to terminate. We consider its result as an *approximation* of the computation of π following the general replay and extend strategy outline above.

We suggest a uniform transformation of the program into an approximation exhibiting the same behavior as the original program with respect to prefixes of event histories. The transformation preserves the structure of the program. Thus it is possible to use a verification approach that follows the structure of the implementation. Moreover this approach enables us to employ well known verification techniques for sequential programs as described in, for instance, [7] and [8]. The latter system has been used for the verification of SMTP.

3.1 Computing Approximations

In this section the uniform procedure to convert programs π_i into their approximations $\tilde{\pi}_i$ is described. Let the program π_i be given as $\pi_i = (\delta_i | \alpha_i(\bar{x}))$, where $\bar{x}$ are the (program) variables occurring free in α_i and δ_i is the list of procedure

declarations used in α_i. The function $approx_{\pi_i}(p_0, h_0)$ will be computed by the program $\tilde{\pi_i}$ given by

$$\begin{array}{l}(\ approx\_i(p, h_{in} : h_{out}, mode) \Leftarrow \textbf{declare } h_c := h_{in}; \bar{x} := \bar{\sigma} \\ \quad \textbf{begin } mode := fin; h_{out} := []; start\_i(p : \bar{x}, h_c, h_{out}); \widetilde{\alpha_i}(\bar{x}); \\ \qquad stop(: h_{out}, h_c, mode) \textbf{ end}, \\ \quad ext\_call(\pi_i), \tilde{\delta_i} \mid approx\_i(p_0, h_0 : h_1, m_0) \ )\end{array}$$

where the initial values of h_1 and m_0 (used to return the results) are not relevant.

The sequence $ext\_call(\pi_i)$ contains declarations for the procedures that simulate the external calls occurring in π_i together with additional start and stop procedure, *start_i* and *stop*, respectively.

In the computation of the approximation a local variable h_c is used that contains the *currently remaining history* during the execution of $\widetilde{\alpha_i}$. It is set to h_{in} initially. The output history is collected in h_{out} as the computation proceeds while the mode is kept in *mode*.

At least for processes generated by a fork operation the variables in $\bar{x}$ have to be initialized *dynamically*. In these cases the corresponding create event will provide the necessary values. They are then assigned to $\bar{x}$ by the start procedure overwriting the values given by $\bar{\sigma}$. If the initial values are fixed the start procedure will leave $\bar{x}$ untouched.

The construction is guided by the following general idea. An initial segment of the computation of α_i executed by p is replayed using (consuming) h and extending h_{out}.

External calls $c(\bar{\tau} : \bar{z}, res)$ are replaced (or simulated) by procedures with declarations $c\_sim(p, \bar{x} : \bar{y}, res, h_c, h_{out}, mode) \Leftarrow body_c$. The simulating procedures analyze and shorten (consume) the current history h and extend the current output h_{out}. The first argument indicates the process that is executing α_i. Let $\bar{v}$ and $\bar{w}$ be the values of $\bar{\tau}$ and $\bar{z}$, respectively.

If in h_c there is no event generated by p, then the computation (of $\tilde{\alpha_i}$) stops with $h_{out} \circ h \circ [ev_c(p, \bar{v} : \bar{w})]$ as final output, where $ev_c(p, \bar{v} : \bar{w})$ is the event generated by this call of c.

If in h_c there is a further event generated by p, then it has to be $ev_c(p, \bar{v} : \bar{w})$. Otherwise the computation stops signalling a failure. In that case the particular h_c is not realized by π_i (and $\tilde{\pi_i}$) which might happen due to over specification.

For $h_c = h_0 \circ h_1'$, where in h_0 there is no event generated by p and $fst(h_1') = ev_c(p, \bar{v} : \bar{w})$, h_1' is scanned for a matching answer event. If there is no such answer, then the computation stops with $h_{out} \circ h$ as the final output history and *mode* being set to *stop*.

In all these cases the procedure simulating the external call leaves the result parameters untouched since they are not needed anymore.

For $h_1' = h_1 \circ h_2$, where $rst(h_1)$ contains no answer matching $fst(h_1') = fst(h_1) = ev_c(p, \bar{v} : \bar{w})$ and $fst(h_2) = e$ such that $Match\_ev(ev_c(p, \bar{v} : \bar{w}), e)$ the procedure returns values for the result parameters according to $mes\_of(e)$ and

the computation of $\tilde{\alpha_i}$ continues with $rst(h_2)$ as the new remaining history and $h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]$ as the new current output.

The above mentioned analysis of the current history h with respect to an external call $c(\bar{\tau} : \bar{z}, res)$ of p is given by $parse_c(p, h, \bar{v}, \bar{w}) \in His \times His \times His$, where again $\bar{v}$ and $\bar{w}$ are the values of $\bar{\tau}$ and $\bar{z}$, respectively.

$$
\begin{aligned}
parse_c(p, h, \bar{v}, \bar{w}) = (h_0, h_1, h_2) \leftrightarrow (h = h_0 \circ h_1 \circ h_2 \wedge \\
ev_c(p, \bar{v} : \bar{w}) \notin h_0 \wedge \\
(h_1 \neq [] \rightarrow (fst(h_1) = ev_c(p, \bar{v} : \bar{w}) \wedge \\
\forall e \in rst(h_1).\neg Match\_ev(ev_c(p, \bar{v} : \bar{w}), e))) \wedge \\
(h_2 \neq [] \rightarrow Match\_ev(ev_c(p, \bar{v} : \bar{w}), fst(h_2))))
\end{aligned}
$$

The body of the procedure is given below

$$
\begin{aligned}
body_c :\equiv \textbf{declare } & h_0 := parse_c(p, h_c, \bar{x}, \bar{y}).{}_0; \\
& h_1 := parse_c(p, h_c, \bar{x}, \bar{y}).{}_1; \\
& h_2 := parse_c(p, h_c, \bar{x}, \bar{y}).{}_2 \\
\textbf{begin} \\
\textbf{if } & mode \neq fin \textbf{ then skip else} \\
\textbf{if } & \exists e \in h_0.Gen(p, e) \textbf{ then } mode := fail \textbf{ else} \\
\textbf{if } & h_2 = [] \textbf{ then } mode := stop; \\
\textbf{if } & h_1 = [] \textbf{ then} \\
& h_{out} := h_{out} \circ h \circ [ev_c(p, \bar{x}, \bar{y})]; \\
& h_c := [] \textbf{ fi else} \\
& h_{out} := h_{out} \circ h_0 \circ h_1 \circ [fst(h_2)]; \\
& h_c := rst(h_2); y_0 := ret\_val_c^1(fst(h_2)) \\
& \ldots \\
& y_{n-1} := ret\_val_c^{n-1}(fst(h_2)) \\
& res := ret\_res_c(fst(h_2)) \\
\textbf{fi} \quad & \textbf{fi} \quad \textbf{fi}
\end{aligned}
$$

Before the execution of $\tilde{\alpha_i}$ is started the begin of an active thread of p has to be determined by the start procedure. If there is no p-thread executing π_i, then the given input history is returned as output and *mode* is set to *term*.

The procedure that simulates the start of a process π_i is given by the declaration $start\_i(p : \bar{y}, h_c, h\_out, mode) \Leftarrow body_{start\_i}$. It *parses* the given history h according to the definition of *Proc*.

$$
\begin{aligned}
parse_{start\_i}(p, h_c) = (h_0, h_1) \leftrightarrow h = h_0 \circ h_1 \wedge \\
(h_1 \neq [] \rightarrow (Create(i, p, fst(h_1)) \wedge \\
\forall e \in rst(h_1).\neg Term(i, p, e)))
\end{aligned}
$$

The procedure body then is given below

$$
\begin{aligned}
body_{start\_i} :\equiv\ & \textbf{declare } h_0 := parse_{start\_i}(p,h).{}_0; \\
& \qquad\quad h_1 := parse_{start\_i}(p,h_c).{}_1; \\
& \textbf{begin} \\
& \textbf{if } h_1 = [] \textbf{ then } mode := term; h_{out} := h_c; \textbf{ else} \\
& \quad y_0 := init_0^i(get\_mes(fst(h_1))); \\
& \quad \ldots \\
& \quad y_{n_i-1} := init_{n_i-1}^i(get\_mes(fst(h_1))); \\
& \quad h_c := rst(h_1); h_{out} := h_0 \circ [fst(h_1)] \\
& \textbf{fi}
\end{aligned}
$$

Finally we need a stop procedure $stop(: h_{out}, h, mode) \Leftarrow body_{stop}$ that finalizes the simulation. It restores the original history by appending the remaining h to h_{out}. Note that in those cases where a new (final) event was generated h_c will be $[]$. If we have reached the end of $\tilde{\alpha_i}$, indicated by $mode = fin$, we check whether according to the remaining history something needs to be done a signal the result by setting $mode$ to fin or $term$, respectively. This information is needed for decomposing verification problems. The body of the stop procedure is then given as

$$
\begin{aligned}
body_{stop} :\equiv\ & h_{out} := h_{out} \circ h; \\
& \textbf{if } mode = fin \wedge \forall e \in h.\neg Gen(p,e) \textbf{ then } mode := term \textbf{ fi}
\end{aligned}
$$

Whenever $mode$ is changed (to $m \in \{stop, fail\}$) by a procedure simulating an external call the rest of $\tilde{\alpha_i}$ has to be skipped. This is achieved by adding a kind of guards to while loops and (possibly) recursive procedures. In addition h, h_{out}, and $mode$ have to be passed as arguments to the procedures declared in δ_i.

For declarations we have

$$
\begin{aligned}
\emptyset &\mapsto_\sim \emptyset \\
q(\bar{x} : \bar{y}) \Leftarrow \beta\ ,\ \delta &\mapsto_\sim \tilde{q}(\bar{x} : \bar{y}, h_c, h_{out}, mode) \Leftarrow \\
& \qquad \textbf{if } mode \neq fin \textbf{ then skip else } \tilde{\beta} \textbf{ fi}\ ,\ \widetilde{\delta}
\end{aligned}
$$

Commands are modified as follows.

$$
\begin{aligned}
\textbf{skip} &\mapsto_\sim \textbf{skip} \\
x := \tau &\mapsto_\sim x := \tau \\
\alpha_0; \alpha_1 &\mapsto_\sim \widetilde{\alpha_0}; \widetilde{\alpha_1} \\
\textbf{if } \epsilon \textbf{ then } \alpha_0 \textbf{ else } \alpha_1 \textbf{ fi} &\mapsto_\sim \textbf{if } \epsilon \textbf{ then } \widetilde{\alpha_0} \textbf{ else } \widetilde{\alpha_1} \textbf{ fi} \\
\textbf{while } \epsilon \textbf{ do } \alpha \textbf{ od} &\mapsto_\sim \textbf{while } \epsilon \wedge mode \neq fin \textbf{ do } \widetilde{\alpha} \textbf{ od} \\
q(\bar{\tau} : \bar{z}) &\mapsto_\sim \tilde{q}(\bar{\tau} : \bar{z}, h_c, h_{out}, mode) \\
c(\bar{\tau} : \bar{z}, res) &\mapsto_\sim c\_sim(p, \bar{\tau} : \bar{z}, res, h_c, h_{out}, mode)
\end{aligned}
$$

4 SMTP-Server

As already mentioned earlier we considered a non-trivial example in the Verisoft context, namely the full implementation (in a C-like language) and the full specification (in terms of histories) of an SMTP-Server as part of an Simple Mail Transfer Scenario. All in all this implementation required about 7.500 lines of code.

The SMTP server listens for connections from SMTP clients. If a connection has been established, it spawns a child process, which inherits the socket granting access to that new connection. The child communicates with the remote SMTP client while obeying the so-called SMTP Protocol. In the meantime, the main SMTP server process listens again for new connections and spawns child processes to handle the session.

This behaviour can be formalised by a step by step description of the main process and its child processes. For the formalisation we fix the constant SOS (Simple Operating System) representing the operating system. Any process – and thus the SOS as well – is determined by a network address (the host) and a process id on this host.

For simplicity use the following abbreviations: For every history h and process p we define $h \downarrow p$ as the *projection* of the history h on process p. I. e.,

$$\begin{aligned}
() \downarrow p &= () \\
(\langle s, r, m\rangle \circ h) \downarrow p &= \langle s, r, m\rangle \circ h \downarrow p \text{ if s=p or r=p} \\
(\langle s, r, m\rangle \circ h) \downarrow p &= h \downarrow p \text{ otherwise}
\end{aligned}$$

With this we can describe for a given history h the set of histories whose projection on p is just h as

$$h^+ = \{h' \in Hist \mid h' \downarrow p = h\}$$

Recall that we defined the binary operator $\circ$ on histories as the concatenation of its two arguments. In what follows we also use this operator for the "concatenation" of two history sets, and that as

$$H_1 \circ H_2 = \{h_1 \circ h_2 \mid h_1 \in H_1, h_2 \in H_2\}$$

The top-level specification (in terms of histories) of the SMTP-Server then looks as follows: we consider the prefix-closure of the set H_{SMTP_Server} where

$$H_{SMTP\_Server} = H_{CREATE}(p) \circ H_{OPEN}(p, sid) \circ H_{LISTEN}(p, sid) \circ H_{LOOP}(p, sid)$$

for some process p and some socket id sid.

The history sets $H_{CREATE}(p)$, $H_{OPEN}(p, sid)$, and $H_{LISTEN}(p, sid)$ (in the successful case) are easily defined as

$$\begin{aligned}
H_{CREATE}(p) &= (\langle p, SOS, Create(SMTP - Server, SOS, 1)\rangle)^+ \\
H_{OPEN}(p, sid) &= (\langle p, SOS, Sopen(25)\rangle\langle SOS, p, Succ_sopen(sid)\rangle)^+ \\
H_{LISTEN}(p, sid) &= (\langle p, SOS, Slisten(sid)\rangle\langle SOS, p, Succ\rangle)^+
\end{aligned}$$

The history set $H_{LOOP}(p, sid)$ is supposed to cover the parent process of the SMTP-server together with all the children processes that might be initiated.

$$\begin{aligned} H_{\mathrm{LOOP}}(p, sid) = \quad & (\langle p, SOS, Saccept(sid, \infty)\rangle)^+ \cup \\ & H_{\mathrm{ACC}}(p, sid, sid') \circ H_{\mathrm{FORK\_CALL}}(p) \circ \\ & \quad (H_{\mathrm{FORK\_ANS\_C}}(p') \circ H_{CHILD}(p', sid, sid') \cap \\ & \quad\ H_{\mathrm{FORK\_ANS\_P}}(p) \circ H_{CLOSE}(p, sid') \circ H_{\mathrm{LOOP}}(p, sid)) \end{aligned}$$

for some socket id $sid' \neq sid$ and some process $p' \neq p$.

Again, the histories $H_{\mathrm{ACC}}(p, sid, sid')$, $H_{\mathrm{FORK}}(p)$, and $H_{CLOSE}(p, sid')$ (in the successful case) are fairly simple, namely

$$\begin{aligned} H_{\mathrm{ACC}}(p, sid, sid') &= (\langle p, SOS, Saccept(sid, \infty)\rangle\langle SOS, p, Succ\_saccept(sid, rna, rpn)\rangle)^+ \\ &\quad \text{for some remote network address } rna \text{ and port number } rpn \\ H_{\mathrm{FORK\_CALL}}(p) &= (\langle p, SOS, Afork(1)\rangle)^+ \\ H_{\mathrm{FORK\_ANS\_P}}(p) &= (\langle SOS, p, Succ\_afork(hdl)\rangle)^+ \text{ for some handle } hdl \neq none \\ H_{\mathrm{FORK\_ANS\_C}}(p') &= (\langle p', SOS, Create\_Clone(i_C, p, 1)\rangle\langle SOS, p', Succ\_afork(none)\rangle)^+ \\ H_{CLOSE}(p, sid) &= (\langle p, SOS, Sclose(sid)\rangle\langle SOS, p, Succ\rangle)^+ \end{aligned}$$

Remains the most complicated case, namely the specification of the child process which is responsible for carrying out the SMTP protocol. As above, we consider only the successful case here.

$$\begin{aligned} H_{CHILD}(p, sid', sid) = & H_{CLOSE}(p, sid') \circ H_{GREETING}(p, sid) \circ ReadEmails(p, sid) \circ \\ & H_{QUIT}(p, sid) \circ H_{CLOSE}(p, sid) \end{aligned}$$

The history set for $H_{CLOSE(p,sid)}$ is already defined above. $H_{GREETING}(p, sid)$ and $H_{QUIT}(p, sid)$ look as follows:

$$\begin{aligned} H_{GREETING}(p, sid) &= H_{READY}(p, sid) \circ H_{ReadLine}(p, sid, \text{"EHLO "} + ip_r) \circ \\ &\quad H_{GREETS}(p, sid, ip_r) \quad \text{for some remote ip address } ip_r \\ H_{READY}(p, sid) &= (\langle p, SOS, Swrite(sid, \text{"220 "} + get\_na(p) + \\ &\qquad \text{" SMT Service Ready"})\rangle\langle SOS, p, Succ\rangle)^+ \\ H_{GREETS}(p, sid, ip_r) &= (\langle p, SOS, Swrite(sid, \text{"250 "} + get\_na(p) + \text{" greets "} + ip_r)\rangle \\ &\quad \langle SOS, p, Succ\rangle)^+ \\ H_{QUIT}(p, sid) &= H_{ReadLine}(p, sid, \text{"QUIT"}) \circ \\ &\quad (\langle p, SOS, Swrite(sid, \text{"221 "} + \text{p} + \text{" closing"})\rangle\langle SOS, p, Succ\rangle)^+ \end{aligned}$$

ReadLine consists essentially of successively reading one character after the other. A slight complication arises as it may be possible that the attempt to read a single character may be successful, yet results in an empty string.

$$H_{ReadLine}(p, sid, string) = \begin{cases} ReadString(p, sid, string) & \text{if } \exists s : string = s \hat{} CR \hat{} LF \\ & \text{and } s \text{ does not contain } CR \hat{} LF \\ \emptyset & \text{otherwise} \end{cases}$$

i. e., reading a line means to read a string that (uniquely) ends with CR and LF.

$$\begin{aligned} ReadString(p, sid, \text{""}) &= ()^+ \\ ReadString(p, sid, c \hat{} s) &= ReadChar(p, sid, c) \circ ReadString(p, sid, s) \\ ReadChar(p, sid, c) &= ReadEmpty(p, sid) \circ ReadChar1(p, sid, c) \\ ReadChar1(p, sid, c) &= (\langle p, SOS, Sread(sid, 1)\rangle\langle SOS, p, Succ\_sread(1, \text{"}c\text{"})\rangle)^+ \\ ReadEmpty(p, sid) &= ()^+ \cup (ReadEmpty1(p, sid) \circ ReadEmpty(p, sid)) \\ ReadEmpty1(p, sid) &= (\langle p, SOS, Sread(sid, 1)\rangle\langle SOS, p, Succ\_sread(0, \text{""})\rangle)^+ \end{aligned}$$

Remains to specify the history set *ReadEmails* (which in addition covers writing the email to the Inbox file):

$$
\begin{aligned}
ReadEmails(p, sid) &= ()^{+} \cup (ReadEmail(p, sid) \circ ReadEmails(p, sid)) \\
ReadEmail(p, sid) &= ReadS(p, sid, s) \circ ReadR(p, sid, r) \circ ReadD(p, sid, d) \circ \\
&\quad WriteEmail(p, s \hat{} r \hat{} d) \quad \text{for some } s, r, d \\
ReadS(p, sid, s) &= ReadLine(p, sid, \text{"MAIL FROM: "} + s) \circ \\
&\quad (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle\langle SOS, p, Succ\rangle)^{+} \\
ReadR(p, sid, r) &= ReadLine(p, sid, \text{"RCPT TO: "} + r) \circ \\
&\quad (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle\langle SOS, p, Succ\rangle)^{+} \\
ReadD(p, sid, d) &= ReadLine(p, sid, \text{"DATA:"}) \circ \\
&\quad (\langle p, SOS, Swrite(sid, \text{"354 Start mail input;} \\
&\qquad \text{end with CRLF . CRLF"})\rangle\langle SOS, p, Succ\rangle)^{+} \circ \\
&\quad ReadD'(p, sid, d) \circ \\
&\quad (\langle p, SOS, Swrite(sid, \text{"}OK\text{"})\rangle\langle SOS, p, Succ\rangle)^{+} \\
ReadD'(p, sid, \text{"."}) &= ReadLine(p, sid, \text{"."}) \\
ReadD'(p, sid, l \hat{} d) &= ReadLine(p, sid, l) \circ ReadD'(p, sid, d) \quad \text{provided } l \neq \text{"."}
\end{aligned}
$$

The final step is to specify *WriteEmail*.

$$
\begin{aligned}
WriteEmail(p, e) = (&\langle p, SOS, Flock(Inbox, \infty)\rangle\langle SOS, p, Succ\rangle \\
&\langle p, SOS, Fseek(Inbox, 1, 0)\rangle\langle SOS, p, Succ\_fseek(pos_1)\rangle \\
&\langle p, SOS, Fwrite(Inbox, e)\rangle\langle SOS, p, Succ\_fwrite(pos_2, n)\rangle \\
&\langle p, SOS, Funlock(Inbox, \infty)\rangle\langle SOS, p, Succ\rangle)^{+}
\end{aligned}
$$

for some file positions pos_1 and pos_2.

It is certainly out of the scope of this paper to show all the verification details for the whole SMTP-Server. Instead, we emphasise on a small portion of it, namely the readLine procedure as specified above.
In a VSE-like fashion the procedure is listed below:

```
PROCEDURE readLine(sid:length,buffer,res)
    int length, ec;
    buffer buffer_array;
    char c, cprevious;
    bool res;
 BEGIN length := 1; res := true; c := null; cprevious := null;
  cl := nil;
  WHILE ((cprevious /= CR OR c /= LF) AND res = true) DO
   length := 1; socket_read(sid:length,buffer,ec);
   if (ec = SUCC) then res := true else res := false fi;
   if (length = 1 and res = t) then
    cprevious := c; c := buffer[0]; cl := write(cl,c) fi
  OD;
 END
```

The readline procedure is supposed to read characters from the given TCP/IP socket until it finds a CR followed by a LF. This behaviour is described by the history sets $H_{readLine}(p, sid, cl)$ for a procedure identifier p, socket id sid and a list of characters (string) cl. Now, the segments of the histories that are members of this set are the result of calling the *readLine* procedure from above. Therefore,

for the verification of the SMTP server, we need to make sure that this procedure (implementation) meets its intended semantics (the corresponding history sets from above), namely $H_{readLine}(p, sid, cl) = ReadLine(sid, cl)$.

According to the technique described above we have to prove the following property:

$$\begin{aligned}&\exists h : (h \circ h_c^0 = h_c \wedge h_{out}^0 = h_{out} \wedge mode = fin\\ &\quad \rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle h_c = h_c^0 \wedge h_{out} = h_{out}^0 \circ h \wedge\\ &\qquad ((mode = fin \wedge res = t) \leftrightarrow h \in H_{readLine}(p, sid, cl)))\end{aligned}$$

The proof of this property is split into three main lemmas (and several small lemmas about the data structures used): The first lemma is formulated close to an invariant used to deal with the (single) while loop occurring in the body of *readLine*. The history of past events every time the while loop is entered (end exited)

$$\begin{aligned}&\exists h : (h \circ h_c^0 = h_c \wedge h_{out}^0 = h_{out} \wedge mode = fin\\ &\quad \rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle\\ &\qquad h_c = h_c^0 \wedge h_{out} = h_{out}^0 \circ h\\ &\qquad \wedge ((mode = fin \wedge res = t)\\ &\qquad\quad \rightarrow h \in H_{readTCPString}(p, sid, cl) \circ H_{readTCPEmpty}(p, sid))\\ &\qquad \wedge (h \in H_{readTCPString}(p, sid, cl) \circ H_{readTCPEmpty}(p, sid) \wedge cl \neq \langle\rangle\\ &\qquad\quad \rightarrow (mode = fin \wedge res = t))\end{aligned}$$

The following lemma shows that we can drop the history sets $H_{readTCPEmpty}$, because $H_{readTCPEmpty}(p, sid) \setminus H_{readTCPEmpty1}(p, sid) = \{\langle\rangle\}$.

$$\begin{aligned}&\exists h : (h \circ h_c^0 = h_c \wedge h_{out}^0 = h_{out} \wedge mode = fin\\ &\rightarrow \langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle h_c = h_c^0 \wedge h_{out} = h_{out}^0 \circ h \wedge\\ &\quad ((mode = fin \wedge res = t) \rightarrow h \notin H_{readTCPEmpty1}(p, sid) \circ H\end{aligned}$$

Finally, we need a lemma that deals with the fact that end of lines are marked with $\langle CR, LF\rangle$. Notably, the proof for this lemma does not require any knowledge about the external call simulation *socket_read_sim*. Thus this example shows how a proof can be separated into parts dealing with concurrent communication and those dealing with properties independent of the communication, even if the properties are not separated by the program structure.

$$\begin{aligned}&\langle readLine(p, sid : h_c, h_{out}, mode, cl, res)\rangle\\ &\qquad\qquad mode = fin \wedge res = t \rightarrow \exists cl_0 : cl = cl_0 \circ \langle CR, LF\rangle\end{aligned}$$

5 Conclusions

We have demonstrated how properties of a complex (formal model of a) distributed system including application level programs of 7.500 lines can be specified in a way that abstracts from internal states of the components and allows for a decomposition of the verification task.

Instead of requiring a certain form of the model we have proposed a technique to a posteriori define a view that in our case represented the one of an application programmer. Other orthogonal views on the same model will be useful. For example to verify the protocol that underlies the network part of the model another decomposition and event structure is necessary.

The price we have to pay for this flexibility is that we have to prove a simulation theorem for each view. This can be justified in our case by the fact that the same application view can be used for all kinds of software implemented for the given (formally specified) operating system. It basically provides one interface between the internal structures and information needed by the outside world.

The theory as described in this paper is limited to safety assertions in the temporal sense. The idea to locally replay and extend histories can be used to embed the technique into full temporal logic. Using for example TLA, [9], one can define a concurrent system where a shared history is on the one side extended by approximations for a particular component and on the other by an oracle that guesses steps of the environment (of that component) according to a given history specification. The temporal treatment then is entirely at the level of histories as opposed to the state transitions of all the components involved.

References

1. The Verisoft Consortium: The verisoft project, http://www.verisoft.de
2. Cheikhrouhou, L., Rock, G., Stephan, W., Schwan, M., Lassmann, G.: Verifying a chip-card-based biometric identification protocol in vse. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, Springer, Heidelberg (2006)
3. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6, 85–128 (1998)
4. Mantel, H.: Information flow control and applications — bridging a gap. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, Springer, Heidelberg (2001)
5. De Roever, W.P.: Concurrency Verification – Introduction to Compositional and Noncompositional Methods. Cambridge University Press, Cambridge (2001)
6. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
7. Schirmer, N.: A verification environment for sequential imperative programs in isabelle/hol. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 398–414. Springer, Heidelberg (2005)
8. Hutter, D., Langenstein, B., Sengler, C., Siekmann, J.H., Stephan, W., Wolpers, A.: Deduction in the Verification Support Environment (VSE). In: Gaudel, M.-C., Woodcock, J.C.P. (eds.) FME 1996. LNCS, vol. 1051, Springer, Heidelberg (1996)
9. Lamport, L.: Specifying Systems – The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2003)

Analysis of Combinations of CRC in Industrial Communication

Tina Mattes[1], Jörg Pfahler[2], Frank Schiller[1], and Thomas Honold[2]

[1] Technical University Munich, Institute of Information Technology in Mechanical Engineering, Boltzmannstr. 15, D-85748 Garching near Munich, Germany
{mattes,schiller}@itm.tum.de
[2] Technical University Munich, Centre for Mathematical Sciences (M11), Boltzmannstr. 3, D-85748 Garching near Munich, Germany
joerg-pfahler@gmx.de, honold@ma.tum.de

Abstract. Cyclic Redundancy Check (CRC) is an established coding method to ensure a low probability of undetected errors in data transmission. In CRC, a checksum (Frame Check Sequence, FCS) is attached to the data. The FCS is a result of a polynomial division by a so called generator polynomial. CRC is widely used in industrial communication where the data are often transmitted through different layers. Each layer has usually its own CRC with its specific generator polynomial. The paper presents results of examinations of such cascades and other combinations of CRC. It is shown that residual error probability can be decreased by choosing the right combination and explained how the residual error probability of already existing cascades has to be determined in order to reduce the number of worst case assumptions in the overall safety proof. The combinations are illustrated by means of examples.

Keywords: Cyclic Redundancy Check, Residual Error Probability, Safety-critical Communication.

1 Introduction

Transmission of data is an essential function of automated plants. Sensors send data to data processing units like Programmable Logic Controllers (PLC) and these processing units send data to actuators affecting the plants under control. In safety-critical applications, the integrity of received data is very important, since undetected errors could lead to dangerous accidents. Therefore, data communication is to be considered as an essential part of the overall safety proof. Its residual error probability has to be involved in the safety calculations (cf. e.g. [2]) and may increase or decrease the overall safety tremendously.

The Cyclic Redundancy Check (CRC) is a widely used coding method to detect errors in data transmission because of three reasons:

- The on-line algorithm can be implemented relatively easy by means of a Linear Feedback Shift Register (LFSR) in hardware (cf. e.g. [9], [10]) or a corresponding software solution by e.g. predefined tables.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 329–341, 2007.

- The number of redundant bits is relatively small compared to approaches of e.g. sending data several times.
- Different methods of proof of the residual error probability have been developed (see [4], [6], [9], [10]).

Since CRC itself is very efficient and widely used it is obvious to analyze different types of existing or potential combinations of CRC because of the following advantages:

- Already implemented and proved solutions can be re-used.
- Specific combinations of CRC can potentially decrease the residual error probability efficiently.
- Existing combinations in the layer-oriented industrial communication can be involved in the proof in order to reduce unnecessary automation equipment costs.

The last mentioned cascading has not been considered explicitly yet for the calculation of the residual error probability. Usually, worst case assumptions are applied and lead to additional effort.

The paper is structured as follows. The mathematical principle of CRC is explained in Section 2. Some remarks about the calculation of the residual error probability are made. In Section 3, the analyzed combinations are introduced and discussed in detail. These combinations are discussed and compared in Section 4. Significant examples are presented there. It is shown that the residual error probability can be reduced by almost no additional effort depending on the chosen combination. Conclusions and statements about necessary future work are given in Section 5.

2 Principle of CRC

In this chapter, basic principles of CRC are summarized. For further and detailed information see e.g. [1], [6], [7], [8], [9].

2.1 Functionality of CRC

In order to detect errors in data transmission by CRC the original data (net data ND, information bits) consisting of m bits are processed in the sender as follows: ND is handled as a binary polynomial $nd(x)$. A so called generator polynomial $g(x)$ has to be chosen. Polynomial $nd(x)$ is multiplied by x^r, where r is the degree of $g(x)$. The result of this multiplication is divided by $g(x)$. The corresponding bit pattern consisting of r bits of the remaining polynomial $fcs(x)$ corresponds to the checksum FCS that is attached to ND:

$$nd(x) \cdot x^r \bmod g(x) = fcs(x). \tag{1}$$

For instance, the bit pattern of information bits ND = [1110011] and the generator polynomial $g(x) = x^3+x+1$ are given. The bit pattern ND leads to the binary polynomial $nd(x) = 1 \cdot x^6+1 \cdot x^5+1 \cdot x^4+0 \cdot x^3+0 \cdot x^2+1 \cdot x^1+1 \cdot x^0 = x^6+x^5+x^4+x+1$. The degree of $g(x)$ is $r = 3$. The polynomial counterpart of the checksum FCS is obtained by

application of (1): $fcs(x) = ((x^6+x^5+x^4+x+1) \cdot x^3) \bmod (x^3+x+1) = x$. Thus FCS = [010] and the telegram T that is sent to the receiver is T = [ND, FCS] = [110011010].

The receiver handles the received telegram T' as a binary polynomial $t'(x)$ and tests if $t'(x)$ is divisible by $g(x)$:

$$t'(x) \bmod g(x) = 0\,? \tag{2}$$

If equation (2) is not true, the telegram was changed during transmission, i.e. T ≠ T'; if equation (2) holds, T is regarded to be transmitted correctly, i.e. T = T' because:

$$\begin{aligned} t(x) \bmod g(x) &= (nd(x) \cdot x^r + fcs(x)) \bmod g(x) \\ &= nd(x) \cdot x^r \bmod g(x) + fcs(x) \bmod g(x) \\ &= fcs(x) \bmod g(x) + fcs(x) \bmod g(x) \\ &= 0\,. \end{aligned} \tag{3}$$

For instance, as in the example above, T = [1110011010] is sent, $g(x) = x^3+x+1$ and the received telegram T' is T' = [1110001011]. Since $t'(x) \bmod g(x) = x^2+1 \neq 0$, hence the falsification is detected.

The determination of FCS in the receiver and the check in the sender is often realized by a linear feedback shift register (LFSR). For the proof of the residual error probability, the LFSR can be modeled by matrix-vector-multiplication. Let I_m denote the unit matrix of dimension $m \times m$, $nd = (d_{m-1}\ d_{m-2}\ \ldots\ d_0)$ a vector whose coefficients are the bits of ND, and $t = (d_{m-1}\ d_{m-2}\ \ldots\ d_0\ c_{r-1}\ c_{r-2}\ \ldots\ c_0)$ a vector consisting of the bits of telegram T. Then t can be calculated by means of a matrix A of dimension $m \times r$ that depends on the used generator polynomial $g(x)$ by:

$$t = nd \cdot (I_m \mid A)\,. \tag{4}$$

The test in the receiver can be formulated as follows:

$$(A^T \mid I_r) \cdot t' = 0\,?$$

If this equation holds, T is regarded to be transmitted correctly.

The method of matrix-vector-multiplication is well described in [6], [7], [8]. It is applied and adapted in section 4.1.

2.2 Undetectable Errors

Obviously, CRC can not detect all errors. If in the example above T' is equal to T' = [1100101010] then equation (2) holds for $t'(x)$ and the falsification is not detectable.

Transmission errors can be modeled by superimposed error patterns F. These patterns have the same lengths (number of bits) like T. A bit of F is allocated by value 0, if the corresponding bit in T is transmitted correctly, and a bit of F is allocated by value 1, if the corresponding bit in T is falsified during the transmission. Consequently, T is superimposed by F such that T' = T+F holds[1]. A transmission

[1] Note that '+' stands for exclusive-or in the space of binary polynomials.

error is undetectable by CRC if and only if the polynomial corresponding to F, $f(x)$, is divisible by the generator polynomial $g(x)$, since:

$$\begin{aligned} t'(x) \bmod g(x) &= (t(x) + f(x)) \bmod g(x) \\ &= t(x) \bmod g(x) + f(x) \bmod g(x) \\ &= f(x) \bmod g(x) . \end{aligned}$$

Therefore, if $t'(x)$ mod $g(x)$ is equal to zero, the same holds for $f(x)$ and vice versa.

Since not all transmission errors are detectable it is necessary to define criteria to measure the quality of error detection. One important criterion is the Hamming Distance, which is the number of bits that at least have to be falsified to constitute an undetectable error. This conforms to the minimum number of entries 1 in an error pattern F over all possible error patterns.

More precise than Hamming Distance is the residual error probability (P_{re}) that is the probability that an erroneous telegram is regarded to be transmitted correctly. A rough estimate for P_{re} is the ratio of the number of undetectable errors to the number of all possible errors:

$$P_{re} \approx \frac{2^m - 1}{2^{m+r} - 1} < 2^{-r} . \tag{5}$$

This estimate is very imprecise since it is assumed implicitly that a bit is corrupted during transmission with probability 0.5. In fact, this probability (so called bit error probability p) is much smaller. The estimate (5) is sufficient for some applications like voice transmission but not for safety-critical applications. There the residual error probability has to be calculated for given generator polynomial $g(x)$ and given length of net data m for various bit error probabilities $p \in (0; 0.5]$.

2.3 Calculation of the Residual Error Probability

The exact calculation of the residual error probability is usually very complex. There are various methods to calculate P_{re}. Fig. 1 gives a survey of these methods.

Monte-Carlo-Simulation. Random samples are used to estimate P_{re} by the ratio of the number of undetectable error patterns to the number of samples. This method is an

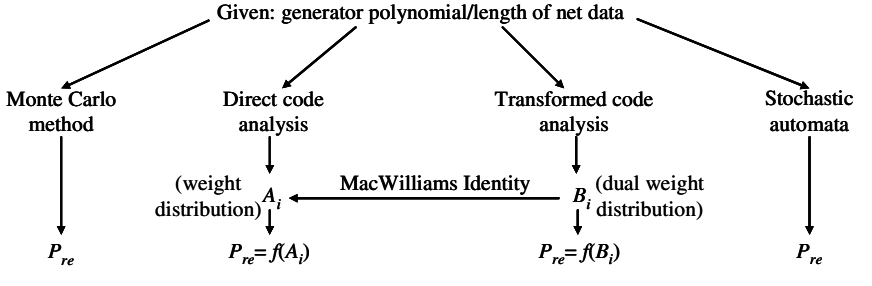


Fig. 1. Survey of methods for the determination of the residual error probability

incomplete determination of estimation (5) and therefore not feasible for safety-critical applications.

Direct code analysis. All 2^m-1 undetectable error patterns have to be generated explicitly. The numbers A_i of those of i erroneous bits have to be counted (A_i, i=1,…, n is the so-called weight distribution). Using the weight distribution, P_{re} is calculated by formula (6)

$$P_{re} = \sum_{i=1}^{n} A_i \cdot p^i \cdot (1-p)^{n-i} \quad . \tag{6}$$

Obviously, the generation of all these error patterns leads to a complexity of 2^m and the computation becomes feasible only for short telegrams.

Transformed code analysis. Instead of generating all undetectable error patterns of the original code, a much smaller set of patterns (2^r patterns instead of 2^m) of the corresponding dual code are generated. The weight distribution B_i of this code (so called dual weight distribution) is determined. Based on the dual weight distribution it is either possible to calculate P_{re} directly or to calculate the weight distribution A_i of the original code by means of the MacWilliams Identity (see [5]) which is the more precise alternative. Both options sometimes lead to numerical problems and to inaccurate results (cf. [6]).

Stochastic automata. This method avoids numerical problems which occur in the transformed code analysis. The idea is to model the behavior of the LFSR which is used to implement the calculation of FCS by a deterministic automaton. This automation is extended to a stochastic one by the explicit inclusion of bit error probability p. P_{re} is equal to the probability of one of the states of the stochastic automaton (for detailed information see [6], [9], [10]).

3 Investigated Combinations of CRC

In this section, three combinations of CRC are introduced.

3.1 CRC with a Generator Polynomial That Is a Product of Two Polynomials ("Product-CRC")

In this first combination, the generator polynomial $g(x)$ is a product of two polynomials $g_1(x)$ and $g_2(x)$ with degree r_1 and r_2. The FCS, consisting of r bits with $r = r_1+r_2$ is calculated by

$$nd(x) \cdot x^{r_1+r_2} \bmod g(x) = fcs(x) \, .$$

The sent telegram T = [ND, FCS] consists of $m+ r_1+r_2$ bits. The receiver checks if equation (2) holds.

This combination is depicted schematically in Fig. 2. Fig. 3 shows the frame of the telegram T.

Fig. 2. CRC with a generator polynomial that is a product of two polynomials

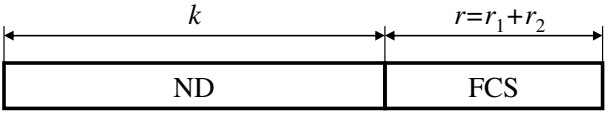


Fig. 3. Frame of the telegram of Product-CRC

This approach leading to a CRC with a new generator polynomial is investigated in order to compare it to the following combinations, since the telegrams have the same number of net data bits m and the same number of check bits $r = r_1+r_2$.

3.2 Twofold CRC with Two Different Polynomials ("Twofold-CRC")

This combination codes the net data ND twice. The first FCS, FCS_1, is calculated based on $g_1(x)$, and the second FCS, FCS_2, by generator polynomial $g_2(x)$:

$$nd(x) \cdot x^{r_1} \bmod g_1(x) = fcs_1(x)$$

$$nd(x) \cdot x^{r_2} \bmod g_2(x) = fcs_2(x)\,.$$

For a schematic illustration of the calculation algorithm see Fig. 4. The frame of the sent telegram T = [ND, FCS_1, FCS_2] is shown in Fig. 5.

The receiver check consists of two checks consequently:

$$(nd'(x) \cdot x^{r1} + fcs_1'(x)) \bmod g_1(x) = 0\ ?$$

$$(nd'(x) \cdot x^{r2} + fcs_2'(x)) \bmod g_2(x) = 0\ ?$$

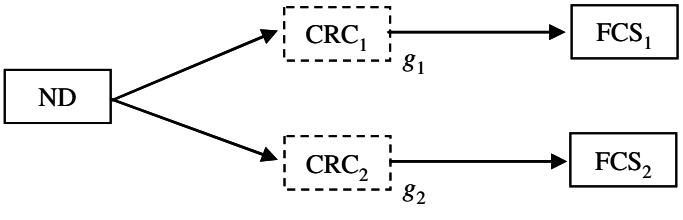


Fig. 4. Schematic illustration of Twofold-CRC

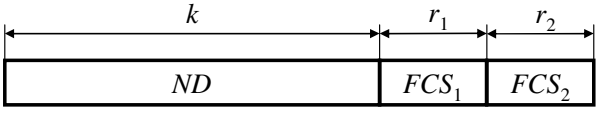


Fig. 5. Frame of the telegram of Twofold-CRC

If and only if both checks hold, the telegram is regarded to be transmitted correctly.

This combination is investigated in order to see how much P_{re} can be decreased by a second CRC and to be compared to the other combinations.

3.3 Cascaded CRC with Two Different Polynomials ("Cascaded-CRC")

In this kind of combination, two FCS have to be calculated like in Section 3.2, but the FCS_1 is handled like net data within the second CRC (see Fig. 6).

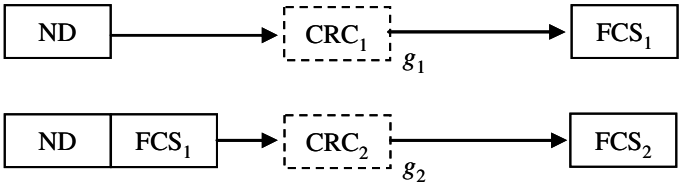


Fig. 6. Schematic illustration of Cascaded-CRC

The frame of the telegram is equal to the structure of the Twofold-CRC in Fig. 5.

First, FCS_1 is calculated. In contrast to Twofold-CRC, FCS_2 is calculated for the bit pattern consisting of ND and FCS_1:

$$nd(x) \cdot x^{r_1} \bmod g_1(x) = fcs_1(x)$$
$$(nd(x) \cdot x^{r_1} + fcs_1(x)) \cdot x^{r_2} \bmod g_2(x) = fcs_2(x).$$

The sent telegram T = [ND, FCS_1, FCS_2] consists of the information bits and both FCS like in Twofold-CRC. Here the receiver checks if:

$$(nd'(x) \cdot x^{r1} + fcs_1'(x)) \bmod g_1(x) = 0\ ?$$
$$\left((nd'(x) \cdot x^{r1} + fcs_1'(x)) \cdot x^{r2} + fcs_2(x)\right) \bmod g_2(x) = 0\ ?$$

If and only if both checks hold, the telegram is regarded to be transmitted correctly.

This combination is analyzed because industrial communication is often executed through different layers. In each layer, a CRC with a specific generator polynomial may be applied. This interlocking has not yet been considered at the calculation of the residual error probability. Thus, usually unnecessary additional work has been done to achieve a residual error probability which was gained by the CRC in upper communication layer only.

4 Comparison of the Combinations

The introduced combinations are compared regarding their residual error probability.

4.1 Calculation of P_{re} of the Combinations

The residual error probability of Product-CRC is calculated by means of stochastic automata (see Fig. 1) because it is the most precise method. P_{re} of Twofold- and Cascaded-CRC are calculated by transformed code analyses where the undetectable error patterns were generated by matrix-vector-multiplication (see Section 2.1).

The matrix to generate telegrams for Twofold-CRC is given by $(I_m \mid A_1 \mid A_2)$, where A_1 is the characteristic matrix for polynomial $g_1(x)$ and A_2 the characteristic matrix for polynomial $g_2(x)$. All undetectable error patterns F (or their corresponding vector f, respectively, cf. section 2.1) of the dual code can be generated by:

$$f = k \cdot \left(\begin{array}{c|c} A_1^T & \\ A_2^T & I_r \end{array} \right)$$

where $k = (k_{r-1}\ k_{r-2}\ \ldots\ k_0)$ comprises all possible vectors element of $\{0;\ 1\}^r$. These undetectable error patterns where used to determine the dual weights as the basis for the calculation of the original weights. P_{re} is calculated by equation (6).

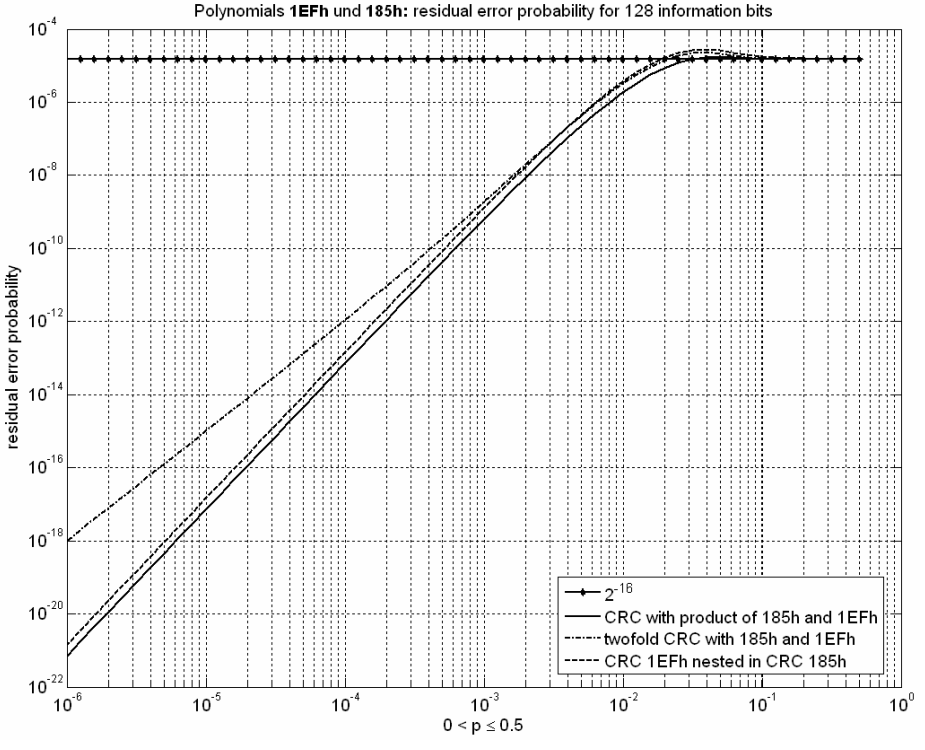


Fig. 7. Residual error probability for polynomials 1EFh and 185h and 128 information bits

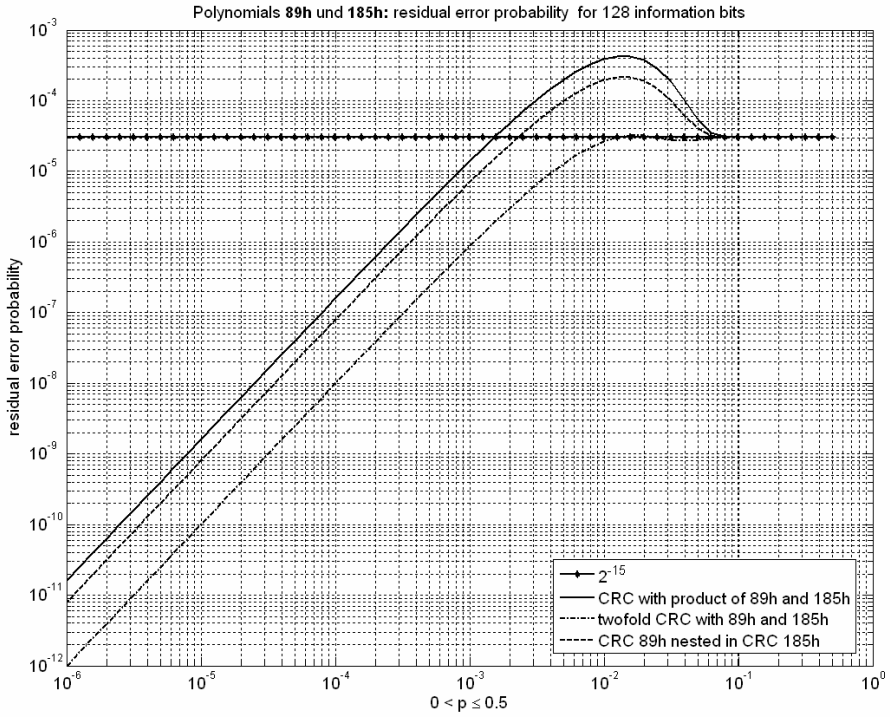


Fig. 8. Residual error probability for polynomials 89h and 185h and 128 information bits

The residual error probability of Cascaded-CRC is determined in the same way. There the matrix to generate telegrams can be derived as follows:

Let $(I_m \mid A_1)$ be the matrix to generate the temporary telegram T_t = [ND, FCS_1] based on the net data ND (i.e. compared to equation (4) it is $t_t = nd \cdot (I_m \mid A_1)$). Furthermore, let $(I_{m+r1} \mid A_2)$ be the matrix that is used to calculate the telegram T = [ND, FCS_1, FCS_2] out of T_t (i.e. $t = t_t \cdot (I_{m+r1} \mid A_2)$) in the second CRC, then the following equations holds:

$$\begin{aligned} t &= t_t \cdot (I_{m+r1} \mid A_2) \\ &= \left(nd \cdot (I_m \mid A_1)\right) \cdot (I_{m+r1} \mid A_2) \\ &= nd \cdot (I_m \mid A_1 \mid B) \end{aligned}$$

with $B = (I_m \mid A_1) \cdot A_2$. Thus, all undetectable error patterns F can be generated by:

$$f = k \cdot \left(\begin{array}{c|c} A_1^T & \\ B^T & I_r \end{array} \right)$$

where $k = (k_{r-1}\, k_{r-2} \ldots k_0)$ comprises all possible vectors element of $\{0; 1\}^r$.

Since matrix multiplication is not commutative, the derivation of the matrix to compute the telegram shows that the sequence of polynomials in Cascaded-CRC is of importance (unlike Product- and Twofold-CRC).

4.2 Results of the Comparison

The comparison of the combinations shows that no combination can be figured out as the best one for all applications. The best combination depends on the generator polynomials $g_1(x)$ and $g_2(x)$, the length m of ND, and the bit error probability p.

In Fig. 7 the residual error probabilities P_{re} of the investigated combinations over bit error probability p is given for polynomials 1EFh[2] and 185h. The number of information bits is 128.

The figure shows, that Twofold-CRC is the worst alternative up to a bit error probability of about 0.003, beyond that the Cascaded-CRC becomes even worse. Product-CRC is the best alternative in this specific case.

For orientation, the residual error probability for equal distribution (at $p = 0.5$) is drawn with line, that is marked with stars (cf. estimation (5)).

In contrast to Fig. 7, Product-CRC is the worst alternative in Fig. 8. There, P_{re} of polynomials 89h and 185h for 128 information bits is given. The product of these polynomials is polynomial C599h that is used in CAN bus [3].

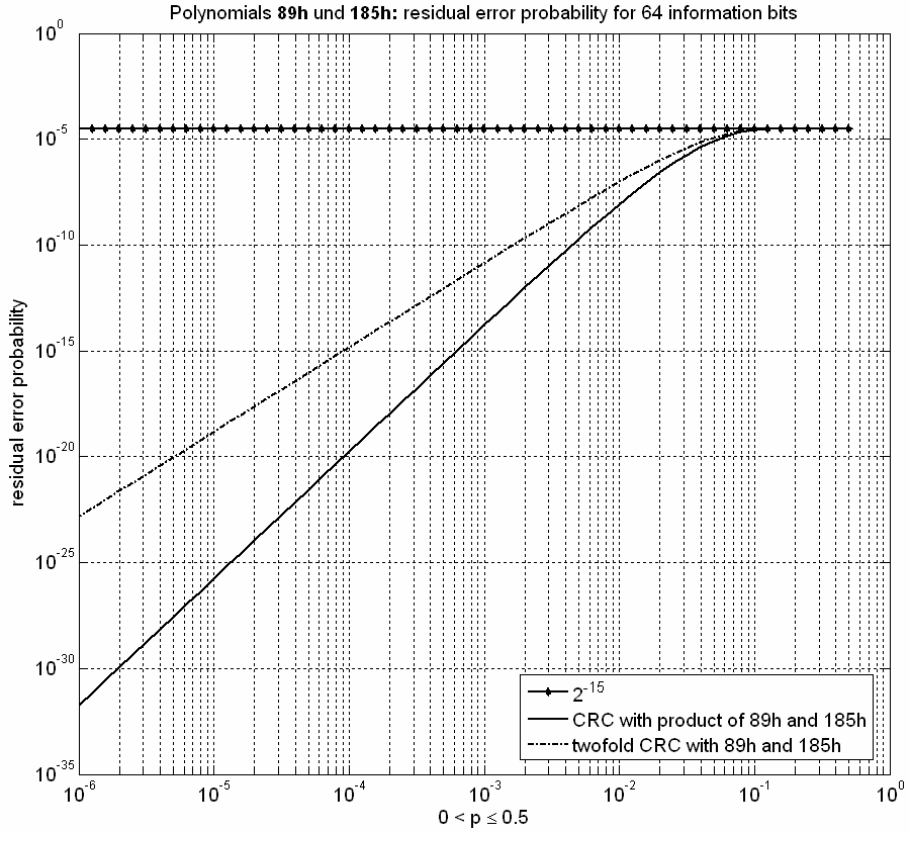


Fig. 9. Residual error probability for polynomials 89h and 185h and 64 information bits

[2] Polynomials are denoted hexadecimally, i.e. 185h = $(1\ 1000\ 0101)_2 = x^8+x^7+x^2+1$.

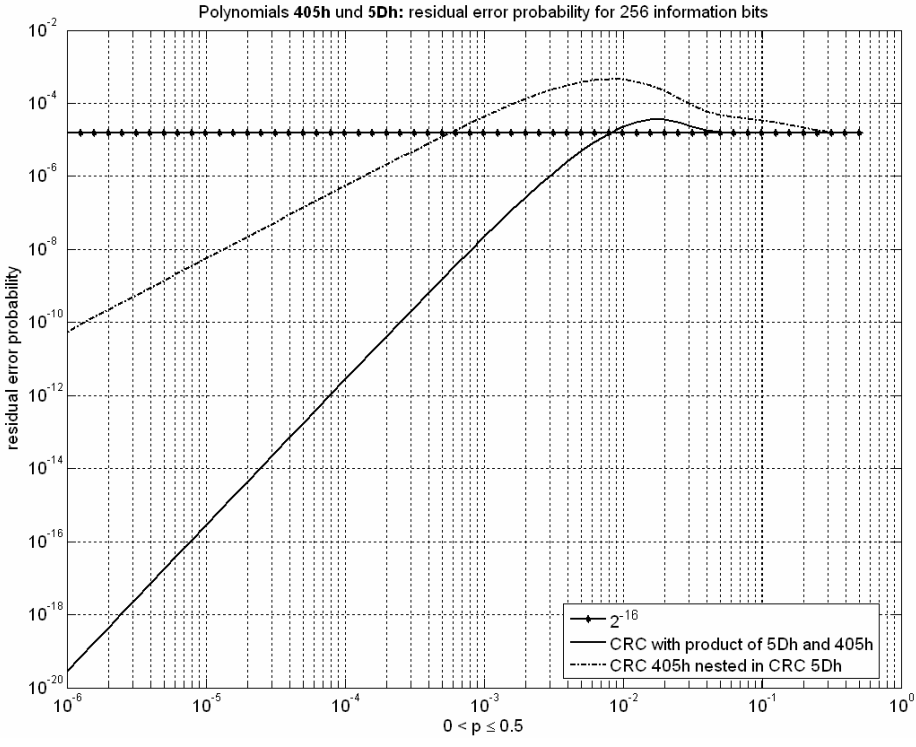


Fig. 10. Residual error probability for polynomials 405h and 5Dh with 256 information bits

Fig. 9 displays the residual error probability again for polynomials 89h and 185h but for 64 information bits, which is the maximal length of a CAN bus telegram. Only Product-CRC and Twofold-CRC are depicted. The figure shows that in this case the Product-CRC is much better than Twofold-CRC. This emphasizes that also the length of net data is very important for the choice of a combination.

In Fig. 10 it is shown, that the quality of the combinations differs in a wide range. There, P_{re} of polynomials 405h and 5Dh for 256 information bits is displayed for Product-CRC and Cascaded-CRC. At bit error probability p = 0.000001, Cascaded-CRC is nearly 10^9 times better in error detection than Product-CRC.

5 Conclusions and Future Work

It has been shown that various combinations of CRC are worth to be investigated for practical applications:

- The appropriate combination of existing implementations of CRC can reduce the residual error probability tremendously.

- Unfortunately no combination can be figured out as the best one for all applications. The choice of the combination depends on the number of information bits and the applicable generator polynomials.
- Existing cascades of CRC can be involved into the safety proof easily.

All necessary algorithms for the calculations of the residual error probability have been developed.

Future work will include additional information bits of the outer CRC in cascades of CRC. There an extension of the calculation methods is necessary. It is based on coding theory as well as on the stochastic automaton approach (cf. Fig. 1)

A second future task investigates the layer-oriented communication in detail. There usually several CRC are applied on different layers with the consequence that a detected erroneous telegram will not be forwarded to the upper layers. In many applications, the detection within the lower layers is even not known and cannot be used for the proof of the overall residual error probability of transmitted data. Thus, the worst case of identical checks has to be assumed with the effect that only the check of the upper known check is applied for the proof. A real problem arises if the determination of the rate of undetectable erroneous telegrams involves the measured rate of detectable erroneous telegrams. There the proof is almost impossible since the detectable erroneous telegrams are not forwarded and not known on the upper layers. The only way to overcome that problem is to increase the independency of the checks of different layers. The solution to that task will be published in a forthcoming paper [11].

References

1. Blahut, R.E.: Algebraic Codes for Data Transmission. Cambridge University Press, Cambridge (2003)
2. International Electrotechnical Comission: Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-related Systems (IEC 61508) (2005)
3. International Organization for Standardization: Road vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and physical signaling (ISO 61508) (2003)
4. Koopman, P., Chakravarty, T.: Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks. In: International Conference on Dependable Systems and Networks, DSN-2004, Florence, Italy, pp. 145–154 (2004)
5. Mac Williams, F.J., Sloane, N.J.A.: Theory of Error-Correcting Codes. North-Holland Mathematical Library (1991)
6. Mattes, T.: Untersuchungen zur effizienten Bestimmung der Güte von Polynomen für CRC-Codes. Univ. of Trier Siemens, AG Nuremberg (2004)
7. Peterson, W., Weldon, E.J.: Error Correcting Codes. MIT Press, Cambridge (1996)
8. Pfahler, J.: Analyse von Kombinationen von Fehleraufdeckungsverfahren in der industriellen Kommunikation. Tech. Univ. of Munich (2006)
9. Schiller, F., Mattes, T.: An Efficient Method to Evaluate CRC-Polynomials for Safety-Critical Communication. Journal of applied computer science 14, 57–80 (2006)

10. Schiller, F., Mattes, T.: Analysis of CRC-polynomials for Safety-critical Communication by Deterministic and Stochastic Automata. In: 6th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes, SAFEPROCESS 2006, Beijing, China, pp. 1003–1008 (2006)
11. Schiller, F., Mattes, T., Büttner, H., Sachs, J.: A New Method to Obtain Sufficient Independency of Nested Cyclic Redundancy Checks. In: 5th International Conference Safety of Industrial Automated Systems, SIAS 2007, November 12-13, 2007, Tokyo, Japan (2007) (accepted for publication)

A Comparison of Partitioning Operating Systems for Integrated Systems

Bernhard Leiner[1], Martin Schlager[1],
Roman Obermaisser[2], and Bernhard Huber[2]

[1] TTTech Computertechnik AG
Schoenbrunner Strasse 7, 1040 Vienna, Austria
phone: +43 1 5853434 0, fax: +43 1 5853434 90
{bernhard.leiner,martin.schlager}@tttech.com
[2] Vienna University of Technology
Treitlstrasse 3, 1040 Vienna, Austria
{ro,huberb}@vmars.tuwien.ac.at

Abstract. In present-day electronic systems, application subsystems from different vendors and with different criticality levels are integrated within the same hardware. Hence, encapsulation of these subsystems is required in the temporal as well as in the spatial domain. Partitioning Operating Systems (OSs) are employed to allow shared access of applications to critical resources within an integrated system.

In this paper we will discuss fundamental properties of partitioning OSs and compare features of existing solutions. Thereby, we will investigate on LynxOS which is a partitioning OS according to ARINC653, on Tresos, a partitioning OS in accordance with AUTomotive Open System ARchitecture (AUTOSAR), as well as on two prototypical partitioning OS realizations that have been implemented within the Dependable Embedded COmponents and Systems (DECOS) project, an integrated project within the Sixth Framework Programme of the European Commission.

Keyworks: Embedded Systems, Dependability, Partitioning OS.

1 Introduction

Dramatic advances within the last years have paved the way for the integration of application subsystems by different vendors into a single coherent embedded system architecture. Thereby, important initiatives (e. g., AUTOSAR [1], Integrated Modular Avionics (IMA) [2], DECOS [3]) in the automotive, avionic and related domains have been concerned with a systematic, domain oriented process to bundle different application subsystems within the same hardware. These approaches target at increased interoperability, a reduction of the number of Electronic Control Units (ECUs), cables and connectors, and an increase in reliability of the overall system.

A fundamental pre-requisite for the integration of different application subsystems is given by a reliable protection mechanism that partitions a system into execution spaces that prohibit unintended interference of different application

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 342–355, 2007.

subsystems. Reliable protection in both the spatial and the temporal domains is particularly relevant for systems where the co-existence of safety-critical and non safety-critical application subsystems shall be supported. Partitioning on node level enforces fault containment and thereby enables simplified replacement/update and increased reuse of SW components. A major commercial benefit of partitioning comes with significantly reduced certification effort for mixed criticality systems.

This paper investigates on different existing OSs that are designed to support the partitioning of hardware resources in order to enable the integration of different application subsystems. Such *partitioning OSs* can be found in the automotive (e. g., Tresos according to AUTOSAR), the avionic (e. g., LynxOS according to ARINC653 for IMA systems), or in cross-industry approaches (e. g., DECOS Encapsulated Execution Environment (EEE) [4], DECOS partitioning OS-based on Real-Time Application Interface (RTAI) [5]).

In this paper we identify fundamental features of partitioning OSs and compare these features based on the existing partitioning OSs: Tresos, LynxOS, DECOS EEE, and the RTAI based DECOS partitioning OS.

The remainder of the paper is structured as follows: Subsequent to this introduction, section 2 describes the concepts of an integrated architecture as tackled by AUTOSAR, IMA, and DECOS. Section 3 outlines the fundamentals of partitioning OSs that enable spatial and temporal partitioning. Section 4 outlines properties of TRESOS, LynxOS, and the DECOS OSs, whereas section 5 compares the features of these solutions. Section 6 concludes this paper.

2 Integrated Architecture

Characteristic of an integrated architecture is the sharing of computational resources (e. g., CPU time, memory) and communication resources (i. e., network bandwidth) among multiple software components. This strategy leads to a reduction of the number of deployed node computers and avoids unnecessary resource duplication. In the following we discuss our system model of an integrated architecture. In particular, we describe the model of an integrated node computer, which exploits a partitioning operating system to establish an execution environment for multiple software components.

2.1 System Model

Today large distributed computer systems are typically constructed out of node computers (e. g., denoted as ECUs in the automotive domain). However, there is a shift towards a component-based integration at a finer level of granularity. In integrated architectures, such as DECOS or AUTOSAR, vendors supply software components instead of node computers. In an additional step these software components are then allocated to the ECUs of the target platform [6,7]. This integration of software components on an integrated node is depicted in Figure 1 and will replace the “1 Function – 1 ECU” methodology of today’s federated architectures.

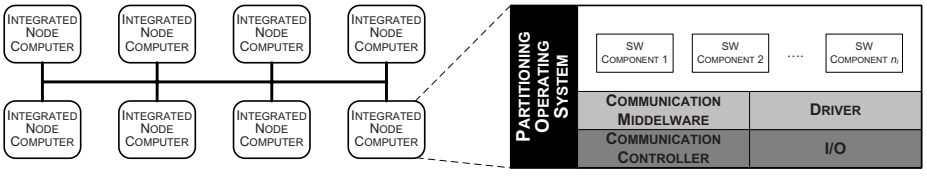


Fig. 1. Integrated Architecture

In order to provide an execution environment that allows the execution of software components without unintended interference, temporal and spatial partitioning for both computational and communication resources is required. For both communication and computational resources, one can distinguish two types of partitioning [8]:

- **Spatial Partitioning.** Spatial partitioning ensures that one software component cannot alter the code or private data of another software component. Spatial partitioning also prevents a software component from interfering with control of external devices (e. g., actuators) of other software components.
- **Temporal Partitioning.** Temporal partitioning ensures that a software component cannot affect the ability of other software components to access shared resources, such as the common network or a shared CPU. This includes the temporal behavior of the services provided by resources (latency, jitter, duration of availability during a scheduled access).

While partitioning of communication resources in an integrated architecture has been addressed in [9], this paper focuses on the partitioning of computational resources.

2.2 Model of an Integrated Node Computer

An integrated node computer provides an execution environment for multiple collocated software components of one or more application subsystems as shown in Figure 1. The model of an integrated node computer comprises:

- *Software components:* The software components implement the application functionality. A software component is part of an application subsystem and represents the unit of distribution. Each software component is the responsibility of a single organizational entity (e. g., a specific supplier). The interaction with other software components occurs through the communication services provided by the communication middleware.
- *Partitioning operating system:* The purpose of the partitioning operating system is the establishment of multiple encapsulated execution environments for combining multiple software components within a single node computer. The encapsulated execution environment provided for a software component is denoted as a *partition* and provides guaranteed computational resources

(CPU time, memory). The partitioning operating system implements mechanisms for spatial and temporal partitioning in order to protect the computational resources of the individual partitions. The scheduling of partitions needs to ensure that a timing failure of a software component, such as a worst-case execution time violation, does not affect the CPU time available to other partitions. In analogy, the spatial partitioning mechanisms of the partitioning operating system include memory protection between partitions (e. g., hardware-enforced with a Memory Management Unit (MMU)). Thereby, each partition emulates a virtual node computer that is dedicated to a single software component only.

- *Communication middleware:* The main purpose of the middleware is the management of the communication resources as previously described. The middleware provides a technology invariant interface to the software components that abstracts from any hardware-specific implementation details. For example, in AUTOSAR [7] the runtime environment (RTE) provides such a generic communication service for the applications. In DECOS the high-level virtual network services perform this task [10].
- *Communication controller:* The purpose of the communication controller is to provide access to the underlying communication system. By the use of hardware drivers and the provision of standardized Application Programming Interface (API) one typically abstracts from the used hardware and thus ensures reuse of existing code in future systems.
- *Input/Output (I/O) and drivers:* The software components hosted on a node computer exploits the input/output subsystem for interacting with the controlled object and the human operator. This interaction occurs either via a direct connection to sensors and actuators or via a fieldbus (e. g., Local Interconnect Network (LIN) [11]). The latter approach simplifies the installation – both from a logical and a physical point of view – at the expense of increased latency of sensory information and actuator control values.

3 Partitioning OS Fundamentals

Integrated node computers as those introduced in Section 2 raise the demand for an OS which is in charge of managing the available resources. Besides the usual feature set of an OS like process scheduling, memory management, inter-process communication and Input/Output, support for *partitioning* in the temporal and the spatial domains as mentioned in Section 2 is important for encapsulation as the basis for composability. A more detailed discussion is included in [12].

The goal of a partitioning OS as depicted in Figure 2 is to provide fault containment equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals, and all inter-partition communications are carried out on dedicated lines [8]. As mentioned above, partitioning can be splitted into *spatial* and *temporal* partitioning. Both of this partitioning dimensions have an influence on the implementation of the basic OS features listed below as well as on the general OS API. For instance, no system call shall corrupt other partitions or the OS itself.

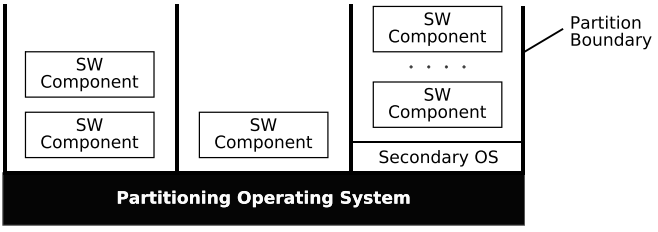


Fig. 2. Concept of a partitioning OS

In the following, we will investigate on the fundamental properties of an partitioning OS, namely scheduling, memory management, and communication.

3.1 Scheduling

Scheduling is concerned with the allocation of resources to software components including the instants of invocation, the assignment of memory regions and the right to access I/O. We can distinguish between static (i. e., offline) and dynamic (i. e., online) scheduling approaches. In [13] the following four classes of scheduling paradigms in the area of real-time systems are outlined:

Static table-driven scheduling: The resource allocation is based on a static schedulability analysis. At runtime, the invocation of tasks is triggered according to a pre-defined schedule, which is usually called table.

Static priority driven preemptive scheduling: A static schedulability analysis is performed but, in contrast to static table-driven scheduling, software components are scheduled at runtime according to a "highest priority first" strategy.

Dynamic planning-based scheduling: With dynamic planning-based scheduling, resource allocation to a software component is decided dynamically based on a feasibility check (that is also performed at runtime).

Dynamic best effort scheduling: The focus of dynamic best effort scheduling is to provide an efficient allocation of given resources to software components. No feasibility checks are performed. Hence, no guarantees with respect to the real-time behavior can be given and tasks may be aborted during their execution.

A partitioning OS typically supports a static table-driven scheduling approach that is very well suited for safety-critical, hard real-time systems since its static nature makes it possible to check the feasibility of the schedule in advance. Furthermore, the maximum time between two partition activations is known in advance.

In addition to partitioning OS services, it should be possible to host several software components within a single partition. In this case, the partitioning OS schedules the partitions and a *secondary OS* schedules the software components

within a partition (*multi-level scheduling*). This secondary OS can implement a simple table-driven scheduling or even a full featured OS or virtual machine.

3.2 Memory Management

Memory management deals with the allocation of memory to a partition (and to the software components residing in that partition). Hence, the OS shall ensure that no undesired interference between any two partitions might occur. Therefore, it must be avoided that a software component is able to write into or execute code from a different partition if not explicitly granted to do so. Spatial partitioning is only possible if the processor provides hardware support for memory protection, i. e., a dedicated protection unit that assigns memory access rights to a certain partition and that avoids errors by strictly blocking illegal requests of faulty partitions.

In existing implementations, two approaches can be found with respect to memory protection that depend on the available hardware support.

Virtual Address Space: Memory protection in modern OSs is typically organized by providing a virtual address space to each process (e. g., Windows, Linux). Virtual addresses are translated to physical addresses by an MMU. In case a virtual address cannot be translated into a physical address (due to an illegal request from a software component), a protection trap is raised.

Protection Blocks: When using protection blocks, multiple memory areas are assigned to software components with different access rights, e. g., read, write, execute. This approach is comparably simpler and thus favored for low-end embedded devices that offer a Memory Protection Unit (MPU).

3.3 Interaction

A partitioning OS must support the interaction of a software component with other software components and with its physical environment. We distinguish between: (a) communication that takes place between software components on the same physical node (i. e., *intra-node communication*), (b) communication between software components that are located on different nodes (i. e., *inter-node communication*), and (c) interaction of a software component with its physical environment (i. e., *I/O interaction*).

Intra-Node Communication. The interaction between software components that are located in different partitions on the same physical hardware node must be supported by a partitioning OS. The most simple mechanism for intra-node communication is to provide shared memory regions that can be accessed by more than one partition. More sophisticated approaches provide message channels / queues for intra-node communication.

Inter-Node Communication. Inter-node communication is typically supported by a middleware layer as discussed in Section 2 that provides a message-based interface to a software component. The inter-node communication must be

supported by a partitioning OS in the sense that the realization of a middleware layer is enabled that provides a message-based communication interface to the partitions and that accesses and instruments the communication controller of a node.

I/O Interaction. I/O interaction of a software component with its physical environment takes place either across a standardized fieldbus (e. g., LIN [11], Controller Area Network (CAN) [14]) or via direct I/O. Thus, I/O interaction is concerned with the protection of I/O hardware of a given microcontroller (e. g., a particular I/O block). In case a transducer or a fieldbus is instrumented only from a single partition, it is sufficient to grant this partition access to the transducer/fieldbus. If more than one partition needs to access I/O, typically a shared partition is implemented that offers I/O services to other partitions by inter and intra-node communication.

It should be mentioned that if a memory-mapped device is employed, controlled I/O interaction must be supported by the memory management mechanism as previously discussed. This means that a partition is granted access to the memory region to which the memory-mapped device is linked.

4 Overview of Partitioning OSs

In this section we discuss the capabilities of partitioning OSs that have been selected from the avionics (i. e., LynxOS), automotive (i. e., Tresos), and cross-industry domains (i. e., DECOS OSs). Our survey is based on data available from the respective vendors/research groups. No actual verification of the stated properties has been carried out for this survey.

4.1 Encapsulated Execution Environment – DECOS Core OS

The Encapsulated Execution Environment (EEE) was developed by TTTech within the DECOS project and consists of the DECOS Core Operating System (COS) as well as a set of graphical configuration tools to generate configuration files.

The COS has been written from scratch with strict temporal and spatial partitioning of all system resources in mind. Using a static configuration is the fundamental mechanism to implement the partitioning functionality. Resource ownership and scheduling is defined statically and can be checked for feasibility in advance. Besides partitioning mechanisms, the COS provides a system interface for intra-node communication as well as error handling and health checks.

Scheduling. The COS uses a two-level scheduling hierarchy. The top-level schedule table divides the schedule cycle into multiple time slots, each of them assigned to a single partition. A second schedule table is used to trigger events and their corresponding event handler function within those partition intervals. Both of these schedule tables are generated during system configuration. The

table responsible for scheduling the partitions is fixed whereas the secondary table can be reconfigured during runtime. The COS ensures that SW components can only reconfigure events belonging to their partition as well as that the new scheduled time for an event lies within a partition interval of the same partition.

For event handlers within partitions fixed priority-based preemptive scheduling is used. This is also necessary for hardware interrupts. An interrupt belonging to a partition is only enabled if this partition is currently scheduled. If this interrupt is triggered, it is mapped to a high-priority partition event with the ISR as event handler.

Temporal partitioning among different partitions is guaranteed since at the end of a partition time slot, the COS preempts all running event handlers, disables all interrupt sources belonging to this partition and finally sets up the environment for the next partition.

Memory Management. The memory protection mechanisms of the COS depends on an MMU that allows the concurrent assignment of permissions to at least five different memory regions (protection blocks). Each partition has execute rights for all its private code as well as for a memory region containing the system interface functions and libraries. Additionally, each partition has read and write permissions for its private data and a dedicated part of a node-wide shared memory. The last protection block is used to grant read access to the whole shared memory.

Interaction. Intra-node communication can be done by using the *message channel* service provided by the COS which can either be *sampled* or *queued* and support an arbitrary number of producers and consumers. A simpler way of intra-node communication can be achieved by using the node-wide shared memory. In the DECOS project this memory is also used for private inter-node communication lines (memory mapped I/O).

Another way to provide private access to I/O devices is to deny direct access to hardware for usual partitions[1]. A dedicated I/O partition with the necessary access rights is used which provides an API for the different hardware components.

4.2 RTAI-Based Partitioning OS

In the course of the DECOS project, a partitioning operating system has been developed, which is based on the real-time Linux variant RTAI and which exploits the Linux Real-Time (LXRT) extension of RTAI for realizing spatial and temporal partitioning. LXRT is an extension of RTAI that enables the development of hard real-time programs running in user space by utilizing the real-time scheduler provided by RTAI. The RTAI real-time scheduler executes the Linux kernel as an idle task, i. e., non real-time Linux applications are only executed when no RTAI/LXRT tasks are active.

[1] Only possible if the target platform provides support for such a restricted run level.

The central element of the partitioning OS is a time-triggered dispatcher, an RTAI kernel module that is responsible for the allocation of processor time to the individual partitions. The partitions are implemented as LXRT tasks, thus executed in user space while preserving the temporal benefits of RTAI.

Scheduling. The activation of partitions is performed by the time-triggered dispatcher that extends the built-in functionality of RTAI/LXRT operating system. The dispatching points are derived from a static dispatching table which is created during system configuration. The dispatching table consists of the activation of a particular partition along with its maximum time granted for execution.

For providing temporal partitioning, individual partitions must not be able to exceed their maximum assigned processor time as defined in the dispatching table, even in the case of a software fault within a partition. For this purpose, deadline monitoring is done by the time-triggered dispatcher: The dispatcher is periodically executed according to the dispatching table. Due to its realization as an RTAI kernel module with the system-wide highest priority, an eventually active partition would be preempted by the dispatcher. The dispatcher analyzes the processor state of the previously executed partition right after its activation and, eventually, removes the partition from the *ready-queue* of the RTAI scheduler. Due to this enforced preemption by RTAI, the partitions are not required to act cooperatively and release the processor on their own.

Memory Management. Memory protection of the RTAI-based partitioning operating system relies on the MMU functionality provided by the processor on which it is executed. Since a processor that is equipped with an MMU basically distinguishes between supervisor mode and user mode, where memory access is only protected in the latter one, spatial partitioning by exploiting the functionality of the MMU can only be provided for applications running in user mode.

Like in many real-time Linux variants, applications using the RTAI API are implemented as Linux kernel modules. Thus, they are executed in supervisor mode and could circumvent memory protection. However, due to the realization of the partitions as LXRT tasks which are executed in user mode, the memory protection mechanisms of Linux are preserved and thus spatial partitioning between individual partitions is provided.

Interaction. According to the system model of DECOS, a software component, denoted in DECOS as *job* [3], is the basic unit of work that is distributed among the nodes of a DECOS cluster. Usually, a mapping of one job per partition is established. Communication between jobs is realized via virtual network services [10]. Thus, from the point of view of the individual jobs it is transparent whether an interaction occurs via intra-node or inter-node communication.

The virtual network service is provided by the virtual network middleware, which is implemented in a dedicated partition on each DECOS node. The interaction of jobs with the virtual network middleware occurs via shared memory,

denoted as *ports*. The memory layout and access rights of these ports is determined during system configuration. The code sequences for allocating and initializing the corresponding memory areas are then statically linked to the application code in order to prevent an application developer from allocating forbidden areas. The same strategy is followed for protecting I/O regions, which are accessed by the use of *memory mapped I/O*, i. e., by mapping the physical address of the I/O in the virtual address space of a particular partition.

4.3 AUTOSAR OS – Tresos

The AUTomotive Open System ARchitecture (AUTOSAR) standard contains the operating system specification AUTOSAR OS which defines the main features of an AUTOSAR-compliant OS: real-time performance, static configuration combined with a priority-based scheduling strategy and protective functions for memory and timing during runtime. A further requirement is the capability to be hostable on low-end microcontrollers. In order to support this, the AUTOSAR OS specification introduces four scalability classes, each of them with different mandatory features. Those scalability classes also affect partitioning. E.g., memory protection is only required for class three and four, timing protection for class two and four. The example OS for this section is Tresos, developed by Elektrobit, which supports scalability classes one to four.

The AUTOSAR standard does not know the concept of partitions. Instead, multiple *OS-Applications*, which form a cohesive functional unit composed of tasks, Interrupt Service Routines (ISRs) and other resources, are used. OS-Applications can either be *trusted* or *non-trusted*. Since trusted OS-Applications are allowed to run with monitoring and protection features disabled at runtime, a non-trusted OS-Application is the best fit to the partition notion like defined in Section 3.

Scheduling. The AUTOSAR OS standard is based on OSEK/VDX which is widely used in the automotive industry. In order to be OSEK-compatible, AUTOSAR uses the same fixed priority-based preemptive scheduling strategy. The scheduling is event-triggered and a high-priority event is always able to reserve the CPU which prevents strict temporal partitioning. To support temporal partitioning to a certain extent, two mechanisms are available in AUTOSAR: (1) *Schedule tables* which allow defining a statically, periodic activation of events and (2) *time monitoring* which is used to limit the maximum execution time of tasks/ISRs, the maximum time they are allowed to hold a shared resource/disable interrupts and the arrival rate of tasks/interrupts.

Memory Management. The basic memory protection requirement that shall be fulfilled by the OS is to protect the data, code and stack section of tasks within an OS-Application from other non-trusted OS-Applications. Additionally, it should provide protection for private data and stack for tasks within the same OS-Application. This requires hardware support in form of an MMU or an MPU. Since this is highly platform specific, the AUTOSAR OS standard does not define

implementation details. Tresos uses static allocation of memory to tasks and OS-Applications in combination with an MPU to achieve spatial partitioning of memory.

Interaction. All AUTOSAR software components run on top of the AUTOSAR Runtime Environment (RTE), which acts as a communication abstraction layer. The same interface in form of ports is provided whether intra-node or inter-node information channels are used. The current version of the AUTOSAR RTE specification (Release 2.0) does *not* support memory protection mechanisms even if provided by the OS.

4.4 ARINC653-Compliant Partitioning OS – LynxOS-178

The operating system LynxOS-178 is a real-time operating system that has been developed by LynuxWorks for safety-critical avionic applications based on Integrated Modular Avionics (IMA) [2]. LynxOS-178 adheres to the ARINC standard 653 [15], which is known as APplication EXecutive (APEX) and defines the services of the avionic software environment. APEX provides services for partition management, process management, time management, memory management, interpartition communication, intrapartition communication, and diagnosis. In addition, LynxOS-178 distinguishes between a small partitioning kernel, which establishes the encapsulated partitions, and higher software layers (e.g., for POSIX support) that run within the partitions. LynxOS-178 supports certification to the highest criticality levels, namely DO-178B level A [16]. LynxOS-178 has already been deployed in safety-critical avionic military and aerospace systems.

Scheduling. For the scheduling of partitions, LynxOS-178 uses fixed cyclic scheduling. Each partition is statically assigned CPU time via a periodically recurring time slice. Thereby, interference between partitions is prevented in the temporal domain. Within a partition, on the other hand, LynxOS-178 offers a process-based execution environment with priority-based preemptive scheduling, priority inheritance, and priority ceilings according to the POSIX model.

Memory Management. In analogy to the allocation of the CPU time, LynxOS-178 statically performs the allocation of memory to the partitions. The memory allocation of a partition is fixed at design time and the configured memory size cannot be changed at runtime. An MMU is employed for isolating the partitions from each other. In contrast to the memory allocation at the partition level, dynamic memory management is supported within a partition. Therefore, LynxOS-178 offers an API with POSIX-compliant calls. The software layer for establishing this POSIX interface is not part of the LynxOS-178 partitioning kernel, but executed in the partitions.

Interaction. For the interaction between partitions, ARINC653 specifies communication channels that are accessible via two types of ports: sampling and

queuing ports. At *sampling ports*, successive messages contain identical but updated data. Received messages overwrite old information, thus requiring no message queuing. In *queueing ports*, messages are assumed to contain uniquely different data. Messages are buffered in queues, which are managed on a first-in/first-out (FIFO) basis.

Inner-partition communication services (e. g., message queues, black boards, semaphores, and events) are not part of the LynxOS-178 partitioning kernel, but can be provided by software layers within the partitions.

5 Feature Comparison

Table 1 gives a brief overview of the features of the four partitioning OSs discussed in this paper. LynxOS-178 has the highest maturity level and provides the

Table 1. Feature Comparison Overview

	DECOS COS	DECOS RTAI	Tresos	LynxOS-178
Vendor	TTTech	Vienna University of Technology	Electrobit	Lynuxworks
Maturity	prototype	prototype	commercial	certified
Standard	—	—	AUTOSAR	ARINC653
Footprint	$\leq$ 1 MB	1 – 10 MB (incl. Linux Kernel)	$\leq$ 1 MB	$\geq$ 100 MB (incl. secondary OSs)

Temporal Partitioning	
DECOS COS	Temporal partitioning ensured by static cyclic scheduling of partition time slots. Deadline monitoring to detect faulty SW components within a partition.
DECOS RTAI	Temporal partitioning ensured by static cyclic scheduling of partition time slots. Deadline monitoring to detect faulty SW components within a partition.
Tresos	No strict partitioning due to preemtive scheduling. Time monitoring to detect faulty SW components.
LynxOS-178	Temporal partitioning ensured by static cyclic scheduling of partition time slots.

Spatial Partitioning	
DECOS COS	Private memory statically assigned and protected by MPU. Private communication lines for I/O, inter-partition communication and cluster-wide communication.
DECOS RTAI	Private data protected by MMU. Private communication and I/O by mapping memory into the virtual address room of partitions.
Tresos	Private memory statically assigned and protected by MPU. Communication middleware (RTE) does not support memory protection.
LynxOS-178	Private data protected by MMU.

most features at the cost of higher hardware requirements. Tresos concentrates on the compliance with the AUTOSAR OS specification, which focuses more on backward compatibility and sophisticated communication middleware than on partitioning. Both OSs for the DECOS project have been developed with partitioning as core feature. The DECOS COS is a small OS written for low-end embedded hardware whereas DECOS RTAI profits from the many features provided by underlying Linux kernel.

6 Conclusion

The bundling of software components by different vendors and with different levels of criticality on an integrated node computer as currently undertaken in integrated architecture approaches (e. g., AUTOSAR, ARINC, DECOS) requires strict partitioning of these software components at the OS level. A number of partitioning OSs exist that aim at providing encapsulation of software components in the temporal and the spatial domains.

In this paper we presented four partitioning OSs, i. e., DECOS EEE Core OS, RTAI-based partitioning OS, Tresos, and LynxOS, and compared the features of these partitioning OSs. Thereby, we investigated on the temporal and spatial protection mechanisms as well as on the code size and the targeted area of application of these partitioning OSs. It turned out that although all presented partitioning OSs adhere to the same core principles, there are notable differences with respect to maturity, code size, and support for spatial protection of these OSs. In the future we expect significant improvements and further establishement of partitioning OSs on different markets (particularly in the automotive domain).

Acknowledgments

We would like to thank our colleague Bernhard Wenzl and the anonymous reviewers for their comments on earlier versions of this paper. This work has been supported by the European IST project DECOS under contract No. IST-511764.

References

1. GbR, A.U.T.O.S.A.R.: AUTOSAR – Technical Overview V2.0.1 (June 2006)
2. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. ARINC Specification 651: Design Guide for Integrated Modular Avionics (November 1991)
3. Obermaisser, R., Peti, P., Huber, B., Salloum, C.E.: DECOS: An integrated time-triggered architecture. e&i journal (Journal of the Austrian professional institution for electrical and information engineering) 3 (March 2006)
4. Schlager, M., Herzner, W., Wolf, A., Gründonner, O., Rosenblattl, M., Erkinger, E.: Encapsulating application subsystems using the DECOS core OS. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 386–397. Springer, Heidelberg (2006)

5. Huber, B., Peti, P., Obermaisser, R., El Salloum, C.: Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. In: Proc. of the Third Int. Workshop on Intelligent Solutions in Embedded Systems (May 2005)
6. Heinecke, H., Schnelle, K.-P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Maté, J.-L., Nishikawa, K., Scharnhorst, T.: AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In: Proceedings of the Convergence Int. Congress & Exposition On Transportation Electronics, Detroit, MI, USA, October 2004, SAE, 2004-21-0042 (2004)
7. Scharnhorst, T., Heinecke, H., Schnelle, K.-P., Fennel, H., Bortolazzi, J., Lundh, L., Heitkämper, P., Leflour, J., Mate, J.-L., Nishikawa, K.: AUTOSAR – challenges and achievements 2005. In: VDI Berichte 1907, Verein Deutscher Ingenieure (2005)
8. Rushby, J.: Partitioning for avionics architectures: Requirements, mechanisms and assurance. NASA contractor report CR-1999-209347, NASA Langley Research Center (October 1999)
9. Obermaisser, R., Peti, P.: Realization of virtual networks in the DECOS integrated architecture. In: Proc. of the Workshop on Parallel and Distributed Real-Time Systems 2006 (WPDRTS), April 2005, IEEE Computer Society Press, Los Alamitos (2005)
10. Obermaisser, R., Peti, P., Kopetz, H.: Virtual networks in an integrated time-triggered architecture. In: Proc. of the 10th IEEE Int. Workshop on Object-oriented Real-time Dependable Systems (WORDS2005), Sedona, Arizona, February 2005, pp. 241–253. IEEE Computer Society Press, Los Alamitos (2005)
11. LIN Consortium. LIN Specification Package Revision 2.0 (September 2003)
12. Conmy, P.M.: Safety Analysis of Computer Resource Management Software. PhD thesis, University of York (2005)
13. Ramamritham, K., Stankovic, J.A.: Scheduling algorithms and operating systems support for real-time systems. Proceedings of the IEEE, 55–67 (1994)
14. Robert Bosch Gmbh. CAN Specification, Version 2.0. Stuttgart, Germany (1991)
15. Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services (March 2006)
16. Radio Technical Commission for Aeronautics, Inc (RTCA), Washington, DC. DO-178B: Software Considerations in Airborne Systems and Equipment Certification (December 1992)

Software Encoded Processing: Building Dependable Systems with Commodity Hardware

Ute Wappler and Christof Fetzer

Technische Universtät Dresden
Department of Computer Science
Dresden, Germany
{ute.wappler,christof.fetzer}@inf.tu-dresden.de
http://wwwse.inf.tu-dresden.de

Abstract. In future, the decreasing feature size and the reduced power supply will make it much more difficult to built reliable microprocessors. Economic pressure will most likely result in the reliability of microprocessors being tuned for the commodity market. In the dependability domain we expect the continued spreading of mixed-mode computing systems, i.e., systems that execute both critical and non-critical functionality. To permit the efficient execution of non-critical applications and the correct execution of critical applications, we introduce the concept of Software Encoded Processing (SEP). SEP enforces a crash failure semantics of the underlying CPU. It does not require the source code of encoded programs and provides probabilistic guarantees. To achieve this, arithmetic codes and signatures are used to detect corrupted data and faulty executions of programs.

1 Introduction

Society depends more and more on critical computing systems such as financial systems or x-by-wire systems. Safety related systems are typically built using special purpose hardware. However, such hardware is expensive because the number of units is much smaller than that of commodity systems. Also, such hardware is usually an order of magnitude slower than commodity hardware. We expect that in future there will be economic pressure to use commodity hardware for dependable computing. Systems will need to facilitate the execution of both critical and non-critical applications on the same computer system. Such mixed-mode commodity systems will require new dependability mechanisms which make it possible to cope with the restrictive failure detection capabilities of commodity hardware. One crucial step in providing such mechanisms is the ability to transform value failures of commodity hardware into crash failures: the aim is that the probability that a value failure of the CPU is not transformed into a crash failure is negligible. The concept of Software Encoded Processing (SEP) presented in this paper guarantees such a *failure virtualization*, i.e., the transformation of a (more difficult to handle) failure model into another (easier to handle) failure model.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 356–369, 2007.

A valid question is, of course, if such a SEP is indeed needed. Historically, hardware reliability has been increasing with every new generation. In future, the decreasing feature size of hardware will however not lead to more reliable but to less reliable hardware—logic and memory—as [17] states. [4] impressively describes the effects of reduced feature sizes. Today's CPUs have a variation in operating frequency of about 30% which is dealt with by using die binning, i.e., testing the resulting chips to find their operating frequency. This variability will increase further with decreasing feature sizes because of the following reasons [4]:

- **dopant variation.** The threshold voltage of transistors is controlled by dopants inserted into the transistor channels. The smaller the transistors become, the less dopants are inserted. That results in a greater impact of variations (in this amount of dopants) onto the electrical properties of the transistors.
- **subwavelength lithography.** Nowadays, the wavelength of the light used in lithography is bigger than the produced structures—resulting in rough structures. That again results in variations in the electrical properties of the produced transistors.
- **varying heat flux.** How much heat is produced highly depends on the functionality of a building block and thus varies across the die. Since the transistor's electrical properties are influenced by heat, transistors will have varying properties depending on their location on the die.

So, the uncontrollable variety of the production process will make processor designs, at least as done today, more and more unpredictable.

Apart from that, smaller features are less reliable because smaller transistors age faster and thus become faster unreliable. Furthermore, smaller features are more susceptible to soft errors since supply voltages decrease with decreasing feature size. It is expected that the amount of failures caused by soft errors increases exponentially with every new technology generation [4].

While the number of transistors per chip is expected to increase exponentially for at least another decade, we expect that the increase in CPU cycles needed to provide the functionality of safety-critical components will be much less. Therefore, we are faced with the prospect that in future we will have plenty of CPU cycles to spare but we cannot assume that all instructions are executed correctly. The objective of SEP is to provide probabilistic guarantees that can be tuned for the demanded dependability requirements and the expected maximum failure rate of the underlying hardware. SEP will support commodity hardware while executing standard binaries without requiring their recompilation. Programs with low or no safety requirements will be executed at full speed concurrently to software encoded programs.

2 Related Work

Detection of transient and permanent hardware errors is a widely researched topic. A widespread technique to detect errors in memory are error correcting

codes (ECC) and parities. [16] and [5] demonstrate their usage to protect Itanium and Power4 processors. But soft errors can also influence the logic building blocks of a computing system. On hardware level this problem is usually tackled by replication. [2] describes lock-stepped and loosely lock-stepped processors, that is redundant processors executing the same instruction stream. Their results are compared to detect erroneous executions. These approaches are completed by redundant memory blocks, redundant communication links and redundant disks. This redundancy can also provides means to fail-over.

Another, approach especially used in the avionics and aeronautics area is radiation hardening. This obviously only protects from soft errors. Permanent errors are not handled.

These hardware solutions which are able to handle errors in logic building blocks are very expensive. That typically rules out the execution of non-safety-critical software on such hardware. Furthermore, they are not designed for detection of permanent design faults.

Control flow checking which can be implemented in hardware, e.g., [10] and [11], or software, e.g., [1], provides means to recognize invalid control flow for the executed program, that is execution of instructions which are not expected for the executed binary. If hardware errors do only influence processed data, these errors will not be recognized. Even if an error is detected, it is not possible to identify erroneous data as it is possible with SEP. Furthermore, most control flow checking techniques require the source or assembler code of the executed program to determine a model for the expected control flow. They do not provide adjustable guarantees.

Algorithm based fault tolerance [8,18] and self-checking software [22,3] use invariants contained in the executed program to check the validity of the generated results. This requires that appropriate invariants exist. These invariants have to be designed to provide a good failure detection capability.

On software level, redundant execution too is used to detect hardware errors. [13] duplicates instruction and compares their outcomes. A similar approach is taken in ED4I [14], but the duplicated instructions do not process the original data but a k-multiple of it. Thus all results of duplicate instructions have to be k-multiples of the original results. So, many hardware errors are recognizable. But faults in program loading, changed control flow and operand errors are not recognized if these occur in both program versions. Also not recognized are errors in the code comparing the two results.

Currently, much research is done on how to efficiently exploit multicore processors for fault detection [19] using them for efficient redundant execution of the same program and result comparison. Here too, design faults will influence both copies, and the comparison is critical wrt. soft errors.

The Vital Coded Processor (VCP) presented in [7] uses time, space and data redundancy to recognize transient and permanent errors. Programs executed by the VCP process data which is encoded using:

- an AN-code, that is multiplication of every data item with a constant A, to detect *data modifications* and faulty CPU operations (*operation errors*),

- variable dependent signatures to detect execution of wrong operations (*operator errors*) or usage of wrong operands (*operand errors*), and
- a time stamp d to detect the usage of out-dated operands (*lost updates*).

Thus, instead of using the functional value x_f of the variable x the value is transformed to $x_c = A * x_f + B_x + d$. A is a variable independent constant, B_x is the signature of the variable x and d is the time stamp which identifies the current iteration of the executed program. Signatures are associated with variables after transforming a program into single-assignment form, i.e., every variable is assigned exactly once. Signatures for all input variables are chosen during the program development process. The signatures which are to be expected for any dependent variable including output variables can be derived using the signatures of input variables and the source code of the executed program.

In the following listing a_c, b_c, and c_c are encoded variables. The signature of the result r_c can be precomputed to be $B_a + B_b + B_c$. The code has to maintain the timestamp d. That requires correctional steps such as subtraction of the current d after an addition of two encoded variables.

```
// encoded computation of a+b+c
int f(int a_c, int b_c, int c_c, int d){
    int t_c=a_c+b_c-d;     // t_c = (A*a+Ba+d)+(A*b+Bb+d)-d
                           // t_c = A(a+b)+Ba+Bb+d
    int r_c=t_c+c_c-d;     // r_c = (A*t+Bt+d)+(A*c+Bc+d)-d
                           // r_c = A(t+c)+Bt+Bc+d
                           // r_c = A(a+b+c)+Ba+Bb+Bc+d
    return r_c;
}
```

An encoded program only processes encoded values. It has no direct access to their functional values. The encoding of input values is done by a special hardware encoder before the input is given to the main CPU. The signatures for input variables and precomputed signatures for output variables are stored in another part of special hardware. The precomputed signatures are used by the hardware-implemented checker to test the validity of generated output r_c by evaluating the following condition: $(r_c - d) \mod A == B_a + B_b + B_c$. The functional value r_f of r_c is obtained by integer division: $(r_c - d)/A$. Code checking and decoding require that $B_r = B_a + B_b + B_c < A$ is valid. Special hardware provides the d which is used for encoding, decoding, and code correction during execution. The VCP executes programs in a loop where every run uses the same signatures but a different time stamp d.

It can easily be seen that r_c will not be a valid code word if *a)* any of the operations executed is faulty, e.g., an addition which delivers for some input values incorrect results, *b)* if an unintended operation, e.g., a subtraction instead of an addition, is executed, *c)* an operand is modified or replaced with another operand encoded using a different signature or *d)* an out-dated operand, i.e., an operand from the previous iteration which is encoded using a wrong d, is used. Errors remain undetected if they result in a multiple of A plus the correct signature for the modified variable and the current time stamp d. With high probability, an

introduced error destroys the code of any variable which depends on a variable containing an invalid code word. Further faults might rectify the code. But that is very improbable because these faults would have to create a correctly encoded word for the modified variable, i.e., a multiple of A plus the appropriate signature and timestamp d. In section 4 we compute the probability for undetected errors.

Control structures such as branches or loops are implemented in such a way that the signatures of all variables whose values depend on the control structure are defined independently of the chosen branch or the number of executed iterations. However, the signature will be incorrect if the wrong branch is taken or a wrong number of iterations are executed (see [7]).

The following pseudo code demonstrates the encoding of the unencoded if-statement `if(x>=0){y=z+x}else{y=x-y}`.

```
sigCond = sigGEZ(x_c); //if(x_f<0) sigGEZ(<0) else sigGEZ(>=0)
if( x_c >= 0 ){
   y_c = z_c+x_c-d;          // y_c=A*(z_f+x_f)+Bz+Bx+d
}else{
   y_c = x_c-y_c+d;          // y_c=A*(x_f-y_f)+Bx-By+d
   y_c+= (Bz+Bx);            // y_c=A*(x_f-y_f)+Bx-By+Bz+Bx+d
   y_c+= -(Bx-By)            // y_c=A*(x_f-y_f)+Bz+Bx+d
   y_c+= -sigGEZ(<0)+sigGEZ(>=0); // y_c=A*(x_f-y_f)+Bz+Bx+d
                                  //    -sigGEZ(<0)+sigGEZ(>=0)
}
y_c += sigCond;              // By=Bz+Bx+sigGEZ(>=0)
```

`sigGEZ` computes the signature for the comparison of $x_f >= 0$. `sigGEZ` evaluates to two different values: one if x_f is greater-equal zero and another if x_f is less zero. The signature of y_c after executing the if-statement is defined to be $B_z + B_x + sigGEZ(>= 0)$—independent of the chosen branch. If any of the computations or any of the used operands is faulty, the signature of y_c will be destroyed, i.e., not be equal to $B_z + B_x + sigGEZ(>= 0)$. The same is the case if a branch is chosen which does not match x_f's size in relation to zero.

VCP has the following disadvantages that restrict its use: *a)* The complete data flow of the encoded program has to be known before the execution to be able to precompute the signatures of all output variables. *b)* The source code of all software components is needed. *c)* The signatures chosen for the input variables have to be selected in such a way that the signatures of all variables are smaller than A or corrective actions during execution have to be taken. *d)* Special hardware is required to encode input variables, to store signatures, and to check the signatures of output variables.

3 Software Encoded Processor (SEP)

The goal of SEP is to provide the same safety as the VCP but without its disadvantages. SEP will execute arbitrary programs given as binaries on commodity hardware and turn hardware failures into fail-stop runs.

The main idea is to develop an interpreter which itself is encoded using the principles of VCP [7], that is every variable which is crucial to the correct

execution of a program is encoded. This includes the program executed by the interpreter and data processed by that program, but also data used by the interpreter to manage program execution such as the program counter. The interpreter executes the program in an encoded fashion and thereby generates encoded outputs which are checked by another (standard) hardware unit to determine if they are valid code words (see section 3.4 on Code Checking).

In this paper we focus on the problem how to cope with unpredictable dataflow of programs. For example, for a server program that can process a diverse set of requests, it is in general not possible to know the exact dataflow beforehand because it depends on the exact sequence of requests sent to the server and also the arguments provided to these requests. Hence, in general we cannot precompute the signatures of variables. Furthermore, our interpreter approach introduces new fault types which we have to deal with: the interpreter might load wrong instruction (*loading error*) from the process image, that is the instruction does not match the current program counter (PC) or the PC itself is incorrect. Second, the interpreter might execute a wrong instruction (*selection error*) which does not match the loaded one.

We will not discuss how to encode instructions such as addition, multiplication or jumps. These instructions are assumed to be atomic with high probability, i.e., either executed completely or result in invalid code words. Furthermore, different instructions have to result in different signatures for the data items they produce. See [21] for information on encoding a CPU's instruction set.

3.1 Encoding of the Process Image and Management Data

We abstract from the dataflow of the executed program by associating a signature with each memory address of the virtual address space provided to the executed program. This signature only depends on the memory address and the number of executed instructions (version). Thus, it can be computed dynamically without knowing the executed program. Only knowledge of the number of so far executed instruction and the checked address is required.

The whole process image, that is processed data and program code, are encoded using the following code: $x_c = A * x_f + h(address, version)$ where $address$ is the address of the encoded memory cell and $version$ the number of instructions executed since program start. A memory cell can contain a data item to be processed by the executed program or an instruction to be executed by the interpreter. This code implies that every memory cell has to be updated after each instruction to match the current instruction count. In section 3.5 we shortly introduce a more sophisticated approach which generates less overhead. Obviously, encoding will result in a higher memory consumption depending on the size of A.

Multiplication with A protects from unrecognized *data modifications* and *operation errors*. A has to be chosen in such a way that it is very unlikely that bitflips result in valid code words. That excludes all powers and multiples of two. Forin [7] suggests the usage of prime numbers.

$h(address, version)$ maps $address$ and $version$ to a number smaller than A. Its addition protects from *operand errors*, *operator errors* and *lost updates* as the

signatures introduced by Forin [7] do. $h(address, version)$ is called the signature of *address* or signature of x.

To protect from *loading* and *selection errors* we have to encode the program counter (PC) too. The PC is part of the interpreter and points to the next instruction to be executed. The following code is used: $PC_c = A * PC_f + h(PC_f, version)$. The next section shows how the encoding of the PC is used to detect *loading errors*.

3.2 Execution of Encoded Programs

The following pseudocode shows the interpreter main loop which executes one instruction of the binary and changes the PC to point to the next one.

```
1  // decode PC and load instruction from memory
2  PC_f=PC_c/A;     Instruction_c=M[PC_f];
3
4  // extract execution information from instruction_c
5  OpCode = getOpcode(Instruction_c - h(PC_f, version));
6  O1 = getOperand1Addr(Instruction_c - h(PC_f, version));
7  O2 = getOperand2Addr(Instruction_c - h(PC_f, version));
8  Result = getResultAddr(Instruction_c - h(PC_f, version));
9
10 // Did a load error occur?
11 S_li = Instruction_c % A - h(PC_f, version);
12 S_pc = PC_c % A - h(PC_f, version);
13
14 // execute instruction
15 switch(OpCode){
16     case ADD:
17         *Result = *O1+*O2;
18         *Result -= h(O1,version)+h(O2,version);
19         S_op=formOp(ADD, Result, O1, O2);
20         *Result += Instruction_c/A-S_op;
21         *Result += S_li+S_pc;
22         *Result += h(Result,version+1);
23     case SUB:
24         *Result =*O1-*O2;
25         *Result -= h(O1,version)-h(O2,version);
26         S_op=formOp(SUB, Result, O1, O2);
27         *Result  += Instruction_c/A-S_op;
28         *Result  += S_li+S_pc;
29         *Result  += h(Result,version+1)
30     case MULT:
31         ...
32 }
33 version++;
34 updateSignatures(Instruction_c);
35 incrementPC();
```

Line 2 loads the instruction to be executed from memory. Lines 5 to 8 extract which operation is to be executed and operand and result addresses. These addresses identify registers or memory locations. Immediates are handled similarly. This information can be extracted from the encoded instruction $instruction_c$ after subtracing its signature, because then the contained information is an A-multiple of the unencoded information. Lines 11 to 12 calculate checkvalues which are later on (lines 21 and 28) added to the generated result. `S_li` checks if the loaded instruction matches the current PC, i.e., the instruction signature has to equal $h(PC_f, version)$. Thus, if no error happenend `S_li` should be zero. Next, it is checked if the PC used to load the instruction was a valid codeword. In that case `S_pc` should be zero.

The switch statement uses the `OpCode` to select the code implementing the instruction matching the opcode. Its selection is checked by the lines 19/20 and 26/27. If the correct branch was chosen `Instruction_c/A-S_op` should evaluate to zero. If any of the check values (`S_li, S_pc` and `Instruction_c/A-S_op`) does not, their addition will destroy the code of the result.

The remaining errors: *operand*, *operation* and *operator errors* are handled in the same way as in the VCP. Lines 17 and 24 use encoded numbers for the computations and thus are protected from *operation errors*. Lines 18/22 and 25/29 correct the signature of the result to match the signature which is expected for the result address. If an *operand*, *operator* or *lost update error* for the operands had occurred this correctional step would destroy the code of the result, since the precomputed signature used for correction would not match the actual existing one.

During computation of the result of an instruction its version is updated (see lines 22 and 29). Additionally, the versions of all other encoded values have to be updated (line 34), because otherwise they would not match the current version information `version` which is incremented in each round to protect from *lost updates* (line 33) and counts the number of executed instructions. Last, the PC is incremented by an encoded addition so that it points to the next instruction to be executed. Its encoding ensures that errors on the PC and instruction loading errors will be detected.

3.3 Program Loading

On load time, binary programs have to be transformed into the encoded process image described in section 3.1. This has to be done in a safe way without unrecognized modifications. Therefore, we expect that binaries are equipped with a cryptographic hash covering all parts which are used by the program loader to install the process image in memory. After loading and encoding the binary the loader starts a check procedure which is encoded after the principles of VCP [7]. The check procedure computes the hash of the encoded process image and compares it to the stored hash. If they are equal, the execution can be started. Otherwise, the program has to be loaded again or an error has to be reported.

3.4 Code Checking

Encoding alone will not result in the recognition of errors. Therefor, the code has to be checked. A code word is valid if the condition $x_c \mod A == h(address, version)$ holds. *address* is the address where x_c is stored and *version* either equals the global version counter `version` or is determined using `version` and an additional data structure as described in section 3.5. In addition to checking if a memory address contains a valid code word, the validity of the PC_c has to be checked because control flow errors might only manifest in destroying PC_c's signature.

In the VCP code checking is done by a trusted hardware part. For SEP that could be an FPGA which we expect processors will have on chip in future, the graphics processors, the data receiving party or other software encoded processors—in short anything made fault tolerant by other means. FPGAs for example can be made fault tolerant using triple modular redundancy [6]. Naturally, the code can be checked in parallel by multiple independent checkers. Each of the checkers can independently interrupt the execution of the main CPU.

When and how often code checking is performed, influences performance and the latency of error detection. Code checking has to be done at least before data becomes externally visible. This generates the smallest overhead but results in the biggest latency. On the other hand, a special variable can be used which collects all the errors which occurred. The easiest way to realize such a variable is to sum up (encoded) all results generated by executed instructions. We also have to sum up the modified PC_c to be able to detect control flow errors. The resulting value of the error collecting variable is either valid encoded if no error occurred or invalid otherwise. This variable can be checked periodically. The check will introduce an insignificant overhead, but of course after each instruction two additional encoded additions are required. But an encoded addition is one of the fastest encoded operations (see section 5).

For detecting if the interpreter has crashed or hangs, the checker has to implement a watchdog functionality. The interpreter periodically has to send an alive-sign to the checker to reset the watchdog. Therefor, the error collecting variable can be used which also ensures early detection of execution errors. Additionally, the *version* variable has to be sent to the watchdog. Thus, the watchdog can check if the interpreter does make progress.

If an invalid code word is detected or the watchdog is not reset by the interpreter, the execution can be stopped (fail-stop) or otherwise dealt with, e.g., by going back to a checkpoint with data values with correct signatures or by recomputing invalid data.

3.5 Updating the Versions

Obviously, updating the whole memory image with the new version number after each instruction would generate way too much overhead. Thus, we decided to use additional data structures to store the version of its last change for each address instead of updating the whole process image after each instruction. The

used data structures have to be encoded manually. If any error happened, such as lost updates to the memory and these structures, this has to result in the return of a wrong version number for the affected memory cell. This in return will destroy the code of any memory cell updated using this memory cell.

Note, that using an unencoded data structure like a hashmap would increase the probability for lost updates: a lost update and a not updated hashmap would remain unrecognized. The version of an address has to depend on the current instruction counter `version` **and** the data structure. The simplest way to achieve this is a list. After each instruction, the updated address is inserted before the first list element. Subtracting the number of list items in front of an address from the current `version` results in the version information for that address. If both the update of `version` and the data structure are lost, `version` will not match PC_c. If additionally, PC_c was not updated, the affected instruction will be executed again and if that re-execution is fault-free everything is fine.

When updating an address, the list item describing the last update of that address has to be removed and the surrounding items have to be updated accordingly. Thus, each item has to store an increment which is used when computing version information. The performance of the list approach highly depends on the data locality of the executed program. In worst case, it is $O(n)$ for finding a version information and updating the list. The list can be pruned by updating the versions stored in the memory. Another less locality dependent approach are tree structures which independent of data locality provide a complexity of $O(\log(n))$.

3.6 Implementation

Our proof-of-concept implementation executes DLX-binaries. DLX is an academic RISC instruction set developed by Hennessy and Patterson [15]. For compiling, we use the DLX compiler based on gcc which is provided by UC Santa Cruz for a student project [12].

Currently, we are able to execute simple C-programs which can make use of nearly the whole DLX instruction set, apart from floating point instructions. The binaries can use bitwise logical instructions but these are not completely encoded at the moment: the actual operation is done on the functional values and not on the encoded ones. Encoding of these operations is possible but our implementation is not yet feature complete. We also provide a simple set of system calls which implement basic I/O-functionality such as `printf` and file operations.

4 Guarantees

Code words consist of n functional bits and k redundant bits with $k = sizeOf(A) + 1$. Assumed, that errors are leading to a random modification of a valid code word, such a modification will result in another valid code word and thus in an undetected error with the probability

$$p_{undetected} = \frac{\text{number of valid code words-1}}{\text{number of possible words}} \approx \frac{2^n}{2^{n+k}} = 2^{-k}.$$

To determine the probability of unrecognized errors during program execution, we have to assume a failure model for the executing infrastructure. The least restrictive assumption we can make, is that with every executed instruction an error occurs which modifies the result or used operands. Modifications can change a value to any number. The error is assumed to be uniformly distributed. This results in the following probability distribution $p(x)$ for x failures (i.e., errors which are not detected because codes are still valid), within $noOfInst$ executed instructions:

$$p(x) = \left(1 - \frac{1}{2^k}\right)^{noOfInst-x} * \left(\frac{1}{2^k}\right)^{x} * \binom{noOfInst}{x}$$

The actuarial expectation of this binomial distribution gives the failure rate FIT (failures in $10^9 hours$) for SEP under the assumed model:

$$FIT(k) = \frac{noOfInst \text{ in } 10^9 \text{ hours}}{2^k}$$

The required redundancy k for a decreased failure rate increases logarithmically: To achieve a failure rate of at most 1 undetected error in 10^9 hours 72 bits redundancy are required. With a 128-bit architecture—as used in IBM's Cell processor—32 bit numbers can be encoded using 96 bits of redundancy which can ensure a failure rate of 0 FIT even for an execution environment which injects at least one error into every result or operand.

To test the fault detection capabilities of SEP, we used our simulation-based fault injection tool FITgrind [20]. We compared the three execution variants:

encoded interpreter the used interpreter was encoded as described in section 3 using an A which added 16 bits of redundancy.

unencoded interpreter the same interpreter but no encoding (of the interpreted code, data or interpreter) was used.

native the executed program was compiled to the native architecture and executed.

FITgrind was used to inject probabilistically errors of the following types into the running interpreter or native execution: bitflips in memory and operation results and execution of different instructions to simulate address line errors. How many bits were flipped was chosen according to an exponential probability distribution. That is, mostly one bit was flipped and the probability of n-bit flips decreased exponentially with n. For one bit flips one would expect 100% coverage since one bit flip corresponds to addition or subtration of a power of two which cannot result in another multiple of A with the same signature.

The results of an injection run were compared with an error free (golden) run. It was checked if any erroneous output, i.e., output differing from the golden run, was generated and if an error was recognized either by the OS or the interpreter. For testing, we used a program for computing the MD5 hash of a string consisting of 10,000 characters. That we executed around 8000 times for each variant.

For each run, FITgrind was initialized with another random number and thus generated different error patterns.

While the encoded interpreter produced no incorrect output at all, 4% and 9% respectively of the unencoded interpreted and native executions produced incorrect output. On the other hand, 31% of all native runs and 15% of the unencoded interpreted runs produced complete and correct output despite the injected errors. For the encoded interpreter, only 0.06% of the runs under fault injection produced completely correct results. That shows that the interpreter detects errors before they can propagate and become visible.

5 Performance

For our performance measurements we encoded 32-bit programs using a 31-bit A which doubles the memory consumption and executed the following programs:

- Computation of prime numbers up to 5,000 using the Sieve of Eratosthenes which is very computation intensive and uses a lot of multiplications whose encoding is quite expensive.
- Computing the MD5 hash of a string which has a length of 10,000. This represents a real world problem. It is a mixture of loops, branches and computations.
- A Quicksort, sorting 1,000 numbers which is not computation intensive.

The comparison of unencoded interpreted version and encoded version shows the slow down induced by encoding. The measurements were done on an AMD Athlon 64 running with a clock rate of 2200 MHz under SuSE Linux. For the prime number computation the encoded version is 25 times slower than the unencoded version. But for MD5 the slowdown is 3.4 and for Quicksort 2.2 [9].

Table 1 shows the slowdowns for some encoded 32-bit operations (31-bit A) as we measured them using the processor's time stamp counter. Additions and Subtractions are quite fast, while divisions, multiplications and comparisons induce higher overheads. That confirms slowdowns measured for our example programs: A higher slowdown is to be expected for programs which are using more multiplications, divisions and comparisons.

An encoded multiplication (using encoded 32-bit numbers) requires 128-bit arithmetic which is realized using the processor's SSE extension. Hand-crafted assembler routines proved to be much slower. Comparisons are relatively slow because the signature computation requires additional encoded subtractions and unencoded modulo operations.

Table 1. Runtime comparison of encoded vs. unencoded operations

operation	add	unsigned sub	div	unsigned div	mult	greater
slowdown	3	2	15	7	38	30

The slowdown generated by our straight-forward interpreter implementation are very high and definitely not practicable. We expect these can be reduced to the same levels other virtual machine based approaches provide.

6 Conclusion and Future Work

SEP provides failure virtualization, i.e., it turns value failures of the underlying infrastructure into crash failures. Neither expensive, e.g., radiation hardened, hardware nor recompilation of executed programs are required. Programs for which failure virtualization is required, just have to be executed using the SEP interpreter introduced in Section 3. The interpreter itself is encoded similar to the approach of [7] and encodes the executed program at load time with a similar arithmetic code as used by Forin [7]. But in contrast to Forin's approach, this code can be checked without the requirement to know the complete data flow of the executed program beforehand.

SEP opens up the possibility to run critical applications (in particular, fail-stop but to a more limited extend also fail-operational applications) in parallel to non-critical applications on commodity hardware which is tuned for performance instead of reliability and is in particular much cheaper than hardware designed for reliability.

SEP's failure rate, i.e., the rate of undetected value failures, is bounded. That bound depends on the amount of redundancy used for encoding and is independent of the failure rate of the underlying hardware. As long as encoded numbers do not exceed the processor's native word width an increase in redundancy does not increase the overhead.

Our next steps will be to switch to a more widespread architecture and to replace interpretation as far as possible with precompiled and preencoded parts to reduce the overhead generated by interpretation. Than either recompilation or preprocessing of some bytecode format, e.g., Java Bytecode, will be required. Another option would be to integrate our approach into an existing virtualization framework.

Fault injection experiments have shown that in contrast to native execution or unencoded interpretation, SEP produced no unsafe (i.e., only fail-stop) runs. But a more thorough research on the error detection capabilities of the used code is required. Especially, our knowledge about the influence of the choice of A on the Hamming distance of code words is shallow.

Another interesting question we will look at is the applicability of SEP for code safety, e.g., protection from buffer overflow attacks. As long as an attacker does not know the code parameters, it is nearly impossible for him to inject valid code.

References

1. Bagchi, S., Kalbarczyk, Z., Iyer, R., Levendel, Y.: Design and evaluation of preemptive control signature (PECOS) checking. IEEE Trans. on Computers (2003)
2. Bernick, D., Bruckert, B., Vigna, P.D., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: NonStop advanced architecture. DSN (2005)

3. Blum, M., Luby, M., Rubinfeld, R.: Self-testing/correcting with applications to numerical problems. In: Proceedings of STOC '90, United States (1990)
4. Borkar, S.: Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. IEEE Micro (2005)
5. Bossen, D.C., Tendler, J.M., Reick, K.: Power4 system design for high reliability. IEEE Micro (2002)
6. Carmichael, C.: Triple module redundancy design techniques for virtex series FPGA. Xilinx Application Notes 197 (March 2001)
7. Forin, P.: Vital coded microprocessor principles and application for various transit systems. In: IFA-GCCT (September 1989)
8. Huang, K.-H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Computers (1984)
9. Knauth, T.: Performance improvements of the vital encoded interpreter. Großer Beleg, Technische Universität Dresden (2006)
10. Li, X., Gaudiot, J.-L.: A compiler-assisted on-chip assigned-signature control flow checking. In: Asia-Pacific Computer Systems Architecture Conference. LNCS, Springer, Heidelberg (2004)
11. Mahmood, A., McCluskey, E.J.: Concurrent error detection using watchdog processors—a survey. IEEE Trans. Comput. (1988)
12. Miller, E.L.: UC Santa Cruz, School of Engineering, http://www2.ucsc.edu/courses/cmps111-elm/dlx/install.shtml
13. Nicolescu, B., Velazco, R.: Detecting soft errors by a purely software approach: Method, tools and experimental results. In: DATE 2003 (2003)
14. Oh, N., Mitra, S., McCluskey, E.J.: ED4I: Error detection by diverse data and duplicated instructions. IEEE Trans. Comput. (2002)
15. Patterson, D.A., Hennessy, J.L.: Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1990)
16. Quach, N.: High availability and reliability in the Itanium processor. IEEE Micro (2000)
17. Spainhower, L., Gregg, T.A.: IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. IBM Journal of Research (1999)
18. Stefanidis, V.K., Margaritis, K.G.: Algorithm based fault tolerance: Review and experimental study. In: International Conference of Numerical Analysis and Applied Mathematics (2004)
19. Wang, C., Kim, H.s., Wu, Y., Ying, V.: Compiler-managed software-based redundant multi-threading for transient fault detection. In: Proceedings of CGO 2007 (2007)
20. Wappler, U., Fetzer, C.: Hardware fault injection using dynamic binary instrumentation: FITgrind. In: Proceedings Supplemental, vol. EDCC-6 (October 2006)
21. Wappler, U., Fetzer, C.: Hardware failure virtualization via software encoded processing. In: INDIN 2007 (2007)
22. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. J. ACM (1997)

Reliability Modeling for the Advanced Electric Power Grid*

Ayman Z. Faza, Sahra Sedigh, and Bruce M. McMillin

University of Missouri-Rolla, Rolla, MO, 65409-0040, USA
Phone: +1 (573) 341-7505, Fax: +1 (573) 341-4532
{azfdmb,sedighs,ff}@umr.edu

Abstract. The advanced electric power grid promises a self-healing infrastructure using distributed, coordinated, power electronics control. One promising power electronics device, the Flexible AC Transmission System (FACTS), can modify power flow locally within a grid. Embedded computers within the FACTS devices, along with the links connecting them, form a communication and control network that can dynamically change the power grid to achieve higher dependability. The goal is to reroute power in the event of transmission line failure. Such a system, over a widespread area, is a cyber-physical system. The overall reliability of the grid is a function of the respective reliabilities of its two major subsystems, namely, the FACTS network and the physical components that comprise the infrastructure. This paper presents a mathematical model, based on the Markov chain imbeddable structure, for the overall reliability of the grid. The model utilizes a priori knowledge of reliability estimates for the FACTS devices and the communications links among them to predict the overall reliability of the power grid.

Keywords: Reliability, cyber-physical, embedded, FACTS, power grid.

1 Introduction

Providing reliable power delivery has always been an essential requirement in the design and maintenance of the power generation and distribution system. In its simplest form, the power grid consists of generators, transmission lines, and corresponding loads. Increased load and a greater number of power transfers, caused by deregulation, stress the power grid to an unprecedented extent. Continued provision of reliable power delivery necessitates distributed, intelligent control of the grid.

The advanced electric power grid, as proposed by the US Department of Energy [1], promises a self-healing infrastructure. To achieve this vision requires a mechanism for maintaining functionality of the grid and ensuring its adaptability to changes in load. Coordinated power electronics can be used to this end,

* Supported in part by NSF MRI award CNS-0420869, NSF CSR award CCF-0614633, and the UMR Intelligent Systems Center.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 370–383, 2007.

by controlling power flow to prevent failures, or to mitigate the effect of failures when they occur.

A Flexible AC Transmission System (FACTS) controller is a system based on power electronics that enable control of one or more AC transmission system parameters [2]. FACTS devices modify specific parameters of the transmission line, such as the line impedance, to control power flow. Correct modification of these parameters necessitates communication among the FACTS devices, to maintain balance in the grid and to harden it against contingencies.

This need for communication and control creates a cyber-physical system with two parallel networks, a cyber network comprised of FACTS devices and the communication links among them, and a physical network comprised of generators, loads, and transmission lines. These two networks interact; however, there is no one-to-one correspondence between their respective components.

Cascading failures have particular significance in our reliability analysis of the grid, as component failures can no longer be assumed independent. One such scenario occurred in North America on August 14, 2003. A power plant failure in Ohio, combined with the tripping of a transmission line, caused a series of cascading failures that eventually led to a complete blackout. Eight states in the Northeastern United States and the province of Ontario were affected, and over 50 million people were left without power [3]. In order to prevent similar blackouts in the future, the power grid and the network of FACTS devices need to be carefully designed and maintained to provide reliable delivery of power.

In this paper, we present a system-level reliability model for the power grid, which includes FACTS devices and the interactions among them. Our effort towards producing a mathematical representation is guided by actual failure scenarios from the field. Exhaustive analysis of all possible failures in a cyber-physical system is infeasible; hence, our model takes into consideration the most common failure scenarios for the grid. This information can aid in determination of the expected frequency of failures, as well as identification of areas of the system where adding redundancy will have the greatest impact on fault tolerance.

2 Related Literature

Reliability modeling has been the subject of numerous studies. In this section, we discuss a select few that are most relevant to this paper. One such study is [4], which describes an early effort in hierarchical modeling of system reliability. The method proposed is useful in cases where the system is too large to be analyzed as a whole, and is instead studied as a sum of its parts. The importance of individual components in determining system reliability was first examined in [5]. The studies presented in [6], [7] and [8] further explore the topic of component importance, introducing metrics and indices that can be used in assessing the importance of components in a system. None of those studies, however, utilizes this knowledge of importance in implementing redundancy or otherwise improving the reliability of the system.

In our research, the specific system being analyzed for dependability is the power grid. Relevant studies include [9], which numerically estimates the reliability of the power grid based on component attributes such as failure rate, outage time, and unavailability. Other research investigates methods for increasing fault tolerance of the grid, including [10], which proposes the use of hybrid cars as local generators of electric power.

As described in Sect. 1, FACTS devices are used to increase the reliability of power grids and facilitate reliable transmission of power, even in the presence of failures in the grid. Use of the max flow algorithm and prudent positioning of FACTS devices are discussed in [11]. This work is further extended in [12], where a distributed version of the max flow algorithm is introduced. FACTS devices are particularly important in reducing the risk of cascading failures. A number of related case studies appear in [13], and [14], which demonstrates the use of FACTS devices in mitigating the risk of cascading failures.

In subsequent sections of this paper, we will use the results of [13] and [14] to develop a model for system reliability. Our work is different from other studies presented in this section, as we are modeling the reliability of the power grid as a whole, including both the physical network, i.e., the grid itself, and the cyber network of FACTS devices.

3 Reliability Modeling of Complex Networks

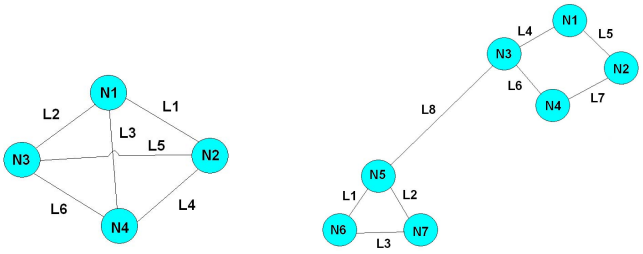
The power grid can be represented as a network comprised of two sets of components; nodes and links. Nodes can represent buses or transformers, while links represent the transmission lines among them. The two main models that we use to represent power networks are the mesh network and the bipartite network. This section provides descriptions of both models.

3.1 The Mesh Network Model

Figure 1(a) depicts the mesh network model for a system of 4 nodes and 6 links. A direct link between every pair of nodes provides redundancy. For a fully functional system, all 4 nodes and all 6 links should be operational; however, the system can still function well despite the failure of one or two of the links, provided that a path still exists between any two nodes. Node failure is more harmful than link failure, but it is possible for the system to remain functional, albeit in a degraded mode, despite the failure of one or more nodes. The number of component failures that the system can tolerate depends on the application for which it is being used.

3.2 The Bipartite Network Model

Figure 1(b) depicts the bipartite network model. Similar to the case of a mesh network, both the links and the nodes of a bipartite network are prone to failure. However, a distinction between this model and the previous one is the presence



(a) Mesh network (b) Bipartite network

Fig. 1. The two network models used to represent the power grid

of link L8, which connects two separate subsystems. When this link fails, the two subsystems are disconnected from each other, and even if each subsystem can function separately, the overall system will operate in a degraded mode. We acknowledge the critical location of L8 by assigning a higher importance value to this link, and therefore, a greater contribution to the overall reliability.

3.3 The Markov Chain Imbeddable Structure

Our model for the reliability of a network of components is based on a Markov chain Imbeddable Structure (MIS). In this section, we provide a brief introduction to the MIS, and illustrate its application to the two network models described in Sects. 3.1 and 3.2.

The state of a system composed of n components can be represented by an n-dimensional binary vector, $\boldsymbol{S}$. Each of the 2^n possible states of this vector represents one combination of component failures for the system.

Let $\boldsymbol{\Pi_0}$ denote a vector of probabilities, where $\Pr(Y_0 = \boldsymbol{S}_i)$ is the probability of the system initially being in state $\boldsymbol{S}_i$. In a normal system, the initial state would be $\boldsymbol{S}_0$, which represents a fully functional network with no component failures.

$$\boldsymbol{\Pi}_0 = [Pr(Y_0 = \boldsymbol{S}_0), Pr(Y_0) = \boldsymbol{S}_1, ..., Pr(Y_0) = \boldsymbol{S}_N]^T \quad . \tag{1}$$

Also, for a given component, l, we define a matrix $\boldsymbol{\Lambda}_l$ that represents the state transition probabilities of the system as a function of l. In other words, each element $p_{ij}(l)$ in the matrix $\boldsymbol{\Lambda}_l$ represents the probability that the system would switch from state $\boldsymbol{S}_i$ to another state $\boldsymbol{S}_j$ due to the failure of component l.

Finally we define $\boldsymbol{u}$, which is a vector of length equal to the number of states, where each element has a value of 1 if the corresponding state is considered a "good" state for the system, and 0 otherwise. In this sense, the system is in a "good" (i.e., acceptable) state if it exceeds the minimum threshold for a given quality of service parameter.

The overall reliability of the n-component system can now be expressed as:

$$R_n = (\boldsymbol{\Pi}_0)^T (\prod_{l=1}^{n} \boldsymbol{\Lambda}_l)\boldsymbol{u} \ . \tag{2}$$

The MIS technique can now be applied to the mesh and bipartite networks. For brevity, we illustrate the technique only for the more general bipartite network.

3.4 Application of MIS to the Bipartite Network Model

To further refine the state of the system, we now define five levels of operation and explain each in detail. This refinement allows the system to endure a finite number of component failures; as long as it delivers some fraction of its expected functionality, it is still considered operational. The system is assumed to initially be in the fully functional state, but transitions to an increasingly degraded operational level with each additional component failure. The five levels are described below.

Level 1: Fully functional system. All components are functional, i.e., the system is in state $\boldsymbol{S}_0$.

Level 2: Degraded mode. A number of links have failed, but all system nodes are still able to communicate with each other, i.e., no node is isolated from the others.

Level 3: Barely acceptable. A single node is isolated from the rest of the system. This could happen as a result of the node failing, or when the failure of a sufficient number of links isolates the node from the remainder of the network.

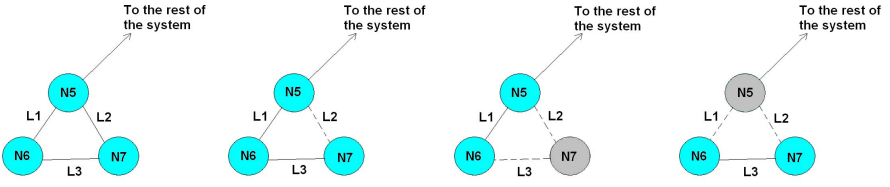
Level 4: Separated subsystems level. The link connecting the two subsystems (L8) has failed. The two subsystems are still functioning separately, but communication between them is lost.

Level 5: Unacceptable operation. Any failures beyond those described in Levels 1-4 bring the system to this state. This level of functionality is considered unacceptable.

Mathematically, a different reliability function corresponds to each level of functionality. Recall that the state vector $\boldsymbol{u}$ represents each "acceptable" state of the system with a 1. Any state where the system is not in Level 5 is considered an acceptable state.

The bipartite network representing a power grid is typically quite large, and as the system grows in size, the reliability analysis becomes increasingly complex. The matrix $\boldsymbol{\Lambda}$, which represents the state transition probabilities, doubles in size with the addition of one component. To simplify the analysis, we divide the system into two subsystems, each of which is analyzed separately with the MIS technique. The results are then combined to produce the system-level reliability equations.

For the example of Fig. 1(b), the portion depicted in Fig. 2(a) is labeled "Subsystem 1," and analyzed first. Here, the subsystem is considered fully functional.



(a) Level 1 (b) Level 2 (c) Level 3 (d) Level 4

Fig. 2. Operational states for Subsystem 1 of the bipartite network of Fig. 1(b)

Level 2 results from the failure of a single link, e.g., L2 in Fig. 2(b). The failure of two links, or equivalently, a single node, further degrades the subsystem operation to Level 3. The failed components depicted in Fig. 2(c) are links L2 and L3, or node N7. Level 4, which causes the separated subsystems state, appears in Fig. 2(d), where links L1 and L2, or node N5 have failed.

3.5 Derivation of Reliability Expressions

In the bipartite network, for each of the two subsystems, equations were calculated using the MIS technique for the different operational levels defined. The overall system reliability can then be calculated as:

$$R_{sys} = R_{sub1} * R_{sub2} * R_{L8} \ , \tag{3}$$

where R_{sys} is the system reliability, R_{sub1} and R_{sub2} are the reliabilities of subsystems 1 and 2, respectively, and R_{L8} is the reliability of link L8, which connects the two subsystems.

In order to derive an expression for each of the operational levels, different combinations of R_{sub1} and R_{sub2} are used to reflect the overall operational level of the system.

Let R_{subi_j} denote the reliability expression corresponding to the j^{th} level of operation in Subsystem i. The equations for overall system reliability can be calculated as follows:

Operational Level 1. For this level, none of the components in either subsystem can fail; therefore, the corresponding expressions for each subsystem are R_{sub1_1} and R_{sub2_1}, respectively, and the overall system reliability becomes:

$$R_1 = R_{sys} = R_{sub1_1} * R_{sub2_1} * R_{L8} \ . \tag{4}$$

Operational Level 2. For this level, only a single link is allowed to fail; this link could be in either subsystem. This raises two cases, if the failed link is in Subsystem 1, then:

$$R_{sys_1} = R_{sub1_2} * R_{sub2_1} * R_{L8} \ . \tag{5}$$

If the failed link is in Subsystem 2, then:

$$R_{sys_2} = R_{sub1_1} * R_{sub2_2} * R_{L8} \ . \tag{6}$$

Assuming that the two cases are equally likely, the mean of (5) and (6) represents the overall system reliability:

$$R_2 = R_{sys} = 0.5 * (R_{sys_1} + R_{sys_2}) \ . \tag{7}$$

Operational Level 3. In this level, the failure of a single node can be tolerated in one of the subsystems, and the failure of a single link in the other subsystem. This again leads to two cases, based on the respective subsystems of the failed node and link. If a node fails in Subsystem 1 and a link in Subsystem 2, then:

$$R_{sys_1} = R_{sub1_3} * R_{sub2_2} * R_{L8} \ . \tag{8}$$

If it is the other way around, then:

$$R_{sys_2} = R_{sub1_2} * R_{sub2_3} * R_{L8} \ . \tag{9}$$

Symmetry, and the assumption that all states are equally likely, leads to the following expression for system reliability at Level 3.

$$R_3 = R_{sys} = 0.5 * (R_{sys_1} + R_{sys_2}) \ . \tag{10}$$

Operational Level 4. This level of operation can occur in any of three cases; if link L8 between the subsystems fails, or if one of nodes N3 or node N5 fails. This can be translated into the following expressions:

$$R_{sys_1} = R_{sub1_1} * R_{sub2_4} * R_{L8} \ . \tag{11}$$

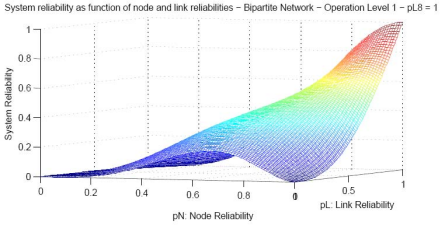
$$R_{sys_2} = R_{sub1_4} * R_{sub2_1} * R_{L8} \ . \tag{12}$$

$$R_{sys_3} = R_{sub1_1} * R_{sub2_1} * (1 - R_{L8}) \ . \tag{13}$$

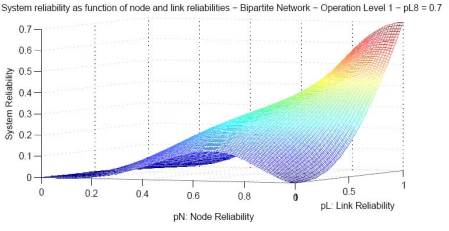
Again, assuming that all cases are equally likely:

$$R_4 = R_{sys} = (1/3) * (R_{sys_1} + R_{sys_2} + R_{sys_3}) \ . \tag{14}$$

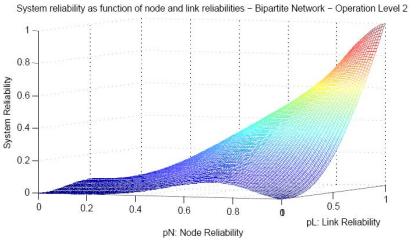
Figures. 3(a), 3(b), and 3(c) depict the overall system reliability as a function of node and link reliability. In Fig. 3(a), the value of R_{L8} is assumed to be 1, corresponding to no chance of failure for the critical link L8. Figure 3(b) assumes R_{L8} to be 0.7. It is clear that this link has a direct effect on system reliability, as the decrease of R_{L8} drastically affects the rest of the system. Finally, Fig. 3(c) represents the reliability of the bipartite network in Operational Level 2. In the next section, we will apply the generalized model for system reliability derived thus far to the specific case of the power grid.



(a) Bipartite network, Level 1, $R_{L8} = 1$



(b) Bipartite network, Level 1, $R_{L8} = 0.7$



(c) Bipartite network, Level 2

Fig. 3. Reliability of networks in Fig. 1 as a function of node and link reliability

4 Reliability of Power Transmission Lines

Several components constitute the power grid, including power generators, transmission lines, and transformers. In addition, we also consider the presence of FACTS devices, as they significantly affect the overall system reliability.

In general, all devices are prone to failure; however, our main interest will be in the failures of transmission lines. Even though generators and transformers can fail, there are usually sufficient backup units installed in the system to compensate for their failures, leaving transmission lines as the main contributors to system failures. Analysis of the reliability of transmission lines is the subject of this section.

4.1 Known Failure Scenarios, No FACTS Devices Installed

Several cascading failure scenarios in the IEEE 118 bus system have been studied and described in detail in [13]. Cascading failures show that links are not independent, negating a common assumption in reliability modeling. Our model does not assume independence of the links.

The following definitions and notations will be used for the remainder of this paper.

F_A: The event that link A remains functional for a specified period of time.
OV_A: The event that link A is overloaded.

NOV_A: The event that link A is not overloaded.
$R(A)$: Reliability of link A
$Q(A)$: Unreliability of link A, which is equal to $1 - R(A)$
Pr(.): Probability that the specified event will happen.

The reliability of link A is defined as the probability that link A remains functional for a specified period of time. Using conditional probability notation,

$$R(A) = Pr(F_A) = Pr(F_A|OV_A) * Pr(OV_A) + Pr(F_A|NOV_A) * Pr(NOV_A) \ . \tag{15}$$

In theory, lines that are not overloaded are expected to function properly for an indefinite period of time. Under normal operating conditions, the failure of a line can only be attributed to accidents, such as inclement weather or physical disconnection. Therefore, the probability denoted as $Pr(F_A|NOV_A)$ is the probability that no such accidents occur for the transmission line during normal non-overloaded conditions.

However, when a line is overloaded, a different situation occurs. Depending on the amount of overload, the duration of time before the line fails can range between 0.15 seconds for an overload of 2000% of the line capacity, and 4.2 seconds for an overload of 150% [15].

In our analysis, we assume that no repair will be carried out for overloaded lines. According to this assumption, if the line becomes overloaded, it will eventually fail. In that case, the term $Pr(F_A|OV_A)$ will be reduced to 0, and (15) becomes:

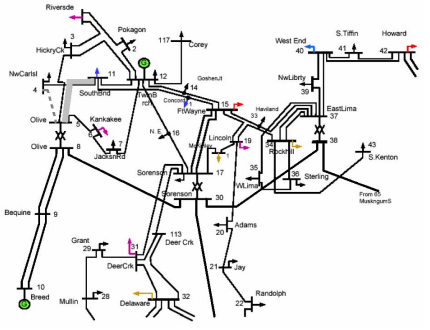
$$R(A) = Pr(F_A) = Pr(F_A|NOV_A) * Pr(NOV_A) \ . \tag{16}$$

Outage of Line 4-5. As described in [13], the outage of line (4-5) leads to several other outages that lead to a failure in the system. The problem starts when the line fails, as line (5-11) becomes overloaded in the process. Taking out line (5-11) will force power to be diverted through lines (5-6), (6-7), and (7-12). Line (7-12) becomes overloaded, and its failure causes overload in lines (3-5) and (16-17). As line (3-5) fails, power is diverted through line (8-30). This causes overload in lines (15-17), (14-15), (15-19), (15-33), and (33-37). Eventually, line (14-15) breaks and causes the system to fail. This sequence of events is depicted in Figs. 4(a)-4(e) From the information available about this scenario, we can now construct reliability expressions for the links involved. For example, line (5-11) becomes overloaded if line (4-5) fails. Thus:

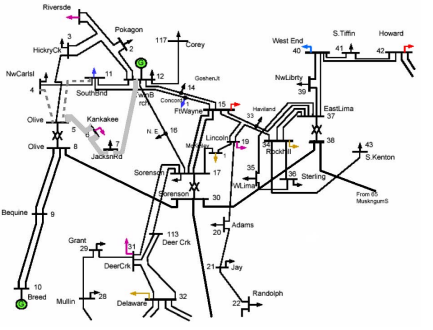
$$R(5-11) = Pr(F_{(5-11)}) = Pr(F_{(5-11)}|NOV_{(5-11)}) * Pr(NOV_{(5-11)}) \ . \tag{17}$$

The probability $Pr(F_{(5-11)}|NOV_{(5-11)})$ is the probability of link (5-11) staying functional in non-overloading conditions, as described above. It is readily apparent that the probability of link (5-11) is the same as the probability of link (4-5) failing, or link (4-5)'s unreliability, Q(4-5), which is equal to 1 - R(4-5). Therefore, $Pr(NOV_{(5-11)}) = 1 - Q(4-5) = R(4-5)$, and the equation for R(5-11) becomes:

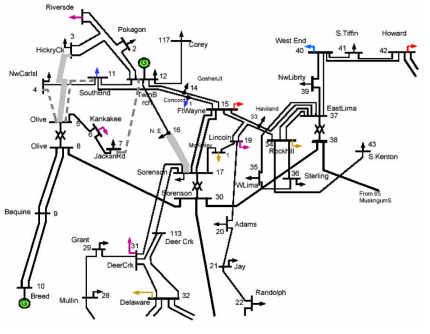
$$R(5-11) = Pr(F_{(5-11)}) = Pr(F_{(5-11)}|NOV_{(5-11)}) * R(4-5) \ . \tag{18}$$



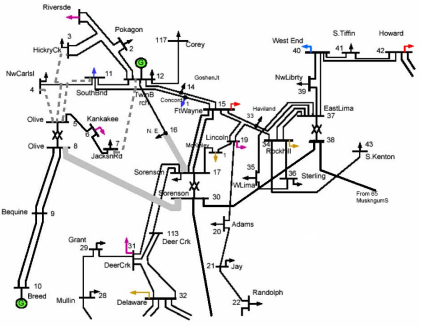
(a) Cascade stage 1



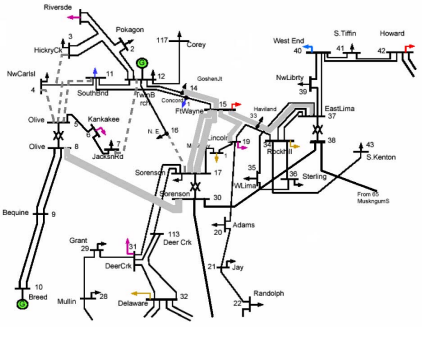
(b) Cascade stage 2



(c) Cascade stage 3



(d) Cascade stage 4



(e) Cascade stage 5

Fig. 4. Cascading failure resulting from the outage of line (4-5)

Taking another example from the same cascading scenario, the reliability of line (7-12) can be found using the expression:

$$R(7-12) = Pr(F_{(7-12)}) = Pr(F_{(7-12)}|NOV_{(7-12)}) * Pr(NOV_{(7-12)}) \ . \quad (19)$$

Again, $Pr(NOV_{(7-12)})$ is a function of the reliability of line (5-11), whose failure causes line (7-12) to overload. Hence, (19) becomes:

$$R(7-12) = Pr(F_{(7-12)}) = Pr(F_{(7-12)}|NOV_{(7-12)}) * R(5-11) \ . \quad (20)$$

Substituting (18) into (20) yields:

$$R(7-12) = Pr(F_{(7-12)}|NOV_{(7-12)}) * Pr(F_{(5-11)}|NOV_{(5-11)}) * R(4-5) \ . \quad (21)$$

We can further simplify the notation, by defining $R_{NOV}(A)$ as the non-overloading, or nominal, reliability of link A to be equal to $Pr(F_A|NOV_A)$. Equations (18) and (21) will therefore become, respectively:

$$R(5-11) = R_{NOV}(5-11) * R(4-5) \ . \quad (22)$$

$$R(7-12) = R_{NOV}(7-12) * R_{NOV}(5-11) * R(4-5) \ . \quad (23)$$

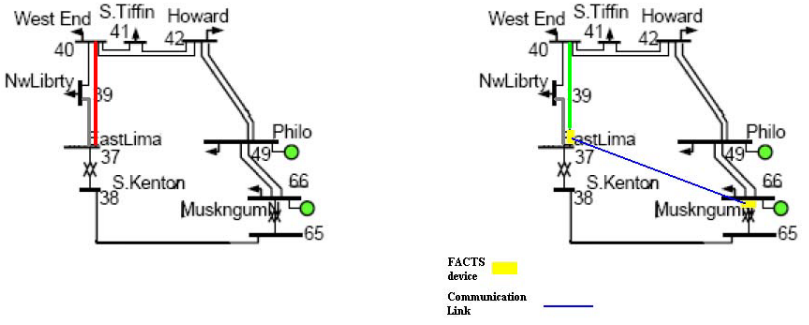
In a similar fashion, reliability equations can be derived for every other link.

4.2 Known Failure Scenarios, FACTS Devices Installed

As an improvement to the situation described in Sect. 4.1, FACTS devices are installed in the system in order to prevent cascading failures from happening. However, if these devices fail or lose communication with each other, failures can still occur and cascades can happen.

Figs. 5(a) and 5(b) show a scenario where the outage of line (37-39) can lead to a cascading failure. After this line fails, line (37-40) becomes overloaded and eventually fails. Installing FACTS devices on lines (37-40) and (66-65) can alleviate the problem, by rerouting enough power through lines (37-38), (38-65), and through transformer (65-66) to prevent line (37-40) from becoming overloaded. A problem arises if one or more of the installed FACTS devices, or a the communication links among them fail. If the FACTS device on line (37-40) fails, line (37-39) will fail, causing (37-40) to become overloaded and eventually fail. In the case of a failure in the communication line, the FACTS devices can still function on their own; however, the effect on line (37-40) will depend on how they choose to operate. A number of possible alternatives are summarized below.

- Bypass the FACTS devices and leave the system to function according to the laws of physics. This choice will obviously cause the system to fail whenever an outage in line (37-39) occurs.
- Set the power flow value to the capacity of the line. This could cause a trouble only if nearby lines are of lower capacity, but in that case the FACTS device would probably have been installed on the lower capacity line. This choice should normally keep the system stable.



(a) Potential cascading failure

(b) Mitigation performed by FACTS devices

Fig. 5. Outage of line (37-39)

- Maintain the power flow values as they were the last time the max flow algorithm was run before the outage of the communication link occurred. Depending on the system, this may or may not be a good choice. In our case, this choice leaves overloads on lines (24-72), (34-43) and (70-75), so it is not the best choice.
- Allow the power flow to change in the system according to the laws of physics, as long as they stay within ±20% of the line capacity, otherwise stop the power flow from increasing any further. Again, depending on the system, this may or may not be a good choice.

In general, one of two cases arises, depending on our choice of action when the communication link fails.

Case 1. The reliability of line (37-40) depends on the reliability of both the FACTS devices at (37-39) and (65-66) and the communication link between them.

Case 2. The reliability of line (37-40) depends only on the reliability of the FACTS device at (37-40), and does not depend on the communication link or the other FACTS device.

In Case 1, the reliability equation for line (37-40) will be:

$$R(37-40) = R_{NOV}(37-40) * Pr(NOV_{(37-40)}) \ . \tag{24}$$

In order to compute the probability of overload for line (37-40), we define the following events.

F1: The event that the FACTS device at line (37-40) remains functional.
F2: The event that the FACTS device at transformer (65-66) remains functional.
CL1: The event that the communication line between the FACTS devices stays functional.
L(37-39): The event that line (37-39) remains functional.

The probability of no overloading in line (37-40) can now be defined as:

$$Pr(NOV_{(37-40)}) = Pr\left[L(37-39) \cup [\overline{L(37-39)}] \cap [F1 \cap F2 \cap CL1]\right] \quad . \quad (25)$$

This can be translated into:

$$Pr(NOV_{(37-40)}) = R(37-39) + Q(37-39) * R(F1) * R(F2) * R(CL1) \ , \ (26)$$

and the equation for the reliability of link (37-40) becomes:

$$R(37-40) = R_{NOV}(37-40) * (R(37-39) + Q(37-39) * R(F1) * R(F2) * R(CL1)) \ . \quad (27)$$

For Case 2, we are only concerned with the reliability of the FACTS device installed on line (37-40). The equation then reduces to:

$$R(37-40) = R_{NOV}(37-40) * (R(37-39) + Q(37-39) * R(F1)) \ . \quad (28)$$

5 Conclusions and Future Work

The objective of the research described in this paper is the development of a model for the reliability of the advanced electric power grid. The model takes into consideration the FACTS network used to regulate power flow and lessen the likelihood of cascading failures. We used the Markov chain imbeddable structure to find expressions for system reliability at predefined levels of degraded operation. On the link level, we demonstrated the sequence of events leading to a cascading failure. The occurrence of cascading failures illustrates that links do not fail independently, meaning that the reliability of each link is dependent on the reliabilities of surrounding links. Using actual failure scenarios, we developed a model for link-level reliability.

The limited amount of information available on failure scenarios is the main challenge in accurate modeling of any aspect of dependability for the electric power grid. In future extensions to this research, we plan to use statistical techniques to investigate the confidence levels achievable for reliability models based on this limited sample set. We also plan to validate the proposed reliability model through large-scale simulation of the power grid, which will undoubtedly lead to refinements to the model.

References

1. Campbell, S.A.: "Grid 2030": A National Vision for Electricity's Second 100 Years (2006), http://www.electricity.doe.gov/documents/Electric_Vision_Document.pdf
2. Edris, A.A., Adapa, R., Baker, M.H., Clark, L.B.K., Habashi, K., Gyugyi, L., Lemay, J., Mehraban, A., Myers, A., Reeve, J., Sener, F., Torgerson, D., Wood, R.R.: Proposed terms and definitions for Flexible AC Transmission System (FACTS). IEEE Transactions on Power Delivery 12(4), 1848–1853 (1997)

3. U.S. Canada Power System Outage Task Force: Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations. Technical report (April 2004)
4. Sahner, R.A., Trivedi, K.S.: A hierarchial, combinatorial-Markov model of solving complex reliability models. In: Proceedings of the 1986 ACM Fall Joint Computer Conference (ACM '86), pp. 817–825. IEEE Computer Society Press, Los Alamitos (1986)
5. Birnbaum, Z.W.: On the importance of different components in a multicomponent system, pp. 581–592. Academic Press, New York (1969)
6. Fricks, R.M., Trivedi, K.: Importance analysis with Markov chains. In: Annual Reliability and Maintainability Symposium, pp. 89–95 (January 2003)
7. Wang, W., Loman, J., Vassiliou, P.: Reliability importance of components in a complex system. In: Annual Reliability and Maintainability Symposium, pp. 6–11 (2004)
8. Coit, D.W., Jin, T.: Prioritizing system-reliability prediction improvements. IEEE Transactions on Reliability 50(1), 17–25 (2001)
9. Billinton, R., Jonnavithula, S.: A test system for teaching overall power system reliability assessment. IEEE Transactions on Power Systems 11(4), 1670–1676 (1996)
10. Parker, R.: On public electrical power grid reliability and terrorism (December 2004), http://www.futurepundit.com/archives/002499.html
11. Armbruster, A., McMillin, B., Crow, M.: Controlling power flow using FACTS devices and the maximum flow algorithm. In: Proceedings of the 5th International Conference on Power Systems Operation and Planning (ICPSOP '02) (December 2002)
12. Armbruster, A., Gosnell, M., McMillin, B., Crow, M.L.: Power transmission control using distributed max flow. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), pp. 256–263. IEEE Computer Society Press, Washington (2005)
13. Chowdhury, B.H., Baravc, S.: Creating cascading failure scenarios in interconnected power systems. In: IEEE Power Engineering Society General Meeting, IEEE Computer Society Press, Los Alamitos (2006)
14. Lininger, A., McMillin, B., Crow, M., Chowdhury, B.: Analysis of max-flow values for setting FACTS devices (2007) (In preparation)
15. Blackburn, J.L.: Protective relaying, 2nd edn. Marcel Dekker, Inc., New York (1998)

Case Study on Bayesian Reliability Estimation of Software Design of Motor Protection Relay

Atte Helminen

TVO, FI-27160 Olkiluoto, Finland
Atte.Helminen@tvo.fi

Abstract. A case study on the reliability estimation of software design of a motor protection relay is presented. The case study is part of a long-term research effort to develop methodology and support for the reliability estimation of computer-based systems to be used in the safety-critical applications of nuclear industry. The estimation method is based on Bayesian inference and the case study is a follow-up to previous case study presented in SAFECOMP 2003.

In the case study reliability estimate of the protection functions of the relay is built in a sophisticated expert judgement process. The expert judgement process consists of two phases including several sessions where the relay designers from different development stages participated. The sessions are named according to the phases as qualitative and quantitative sessions. The qualitative sessions are used to identify and record possible uncertainty and unpunctuality in the planning and documentation of the software design. The quantitative sessions are used to analyse the recordings and to generate a prior reliability estimate. Finally, the prior estimate is updated to a posterior estimate using the operating data of the relay.

The estimation demonstrates the excellence of Bayesian modelling in the reliability estimation of computer-based systems. The reliability estimation typically involves evidence of different kind and with Bayesian modelling the evidence can be combined coherently and transparently together. The estimation method is particularly attractive for probabilistic safety assessment (PSA) of nuclear industry. The method provides informative posterior probability distributions on the failure rates of the protection functions to be used in the PSA models.

1 Introduction

Digital systems are introduced to nuclear power plants in accelerated pace. The instrumentation and control (I&C) systems based on old analog technology are coming to the end of their lifespan and can be considered obsolete in many ways. The old I&C systems will be replaced with digital systems applying the benefits of programmable technology.

Digital systems are generally more tolerant against wear and tear of ageing and environmental factors than the old systems. On the other hand the risk of design faults in digital systems is greater since the implementation of diverse and complex

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 384–396, 2007.

functions is easier and the functionality of a system can be altered significantly just by making minor changes in the software. With increasing influence to the safety of nuclear power plants it is important to be able to accurately and plausibly estimate the reliability of the digital systems.

Considerable effort is generally required for making a reliability assessment of a digital system. The effort is even increased if the system is complex and an estimate of high reliability is aspired. Generating an estimate of high reliability for a non-trivial digital system at least one of the following assumptions must be valid:

- The scope of the system, possible failure modes and/or possible inputs of the system must be strongly reduced or simplified in the assessment.
- Strong influence of expert judgement must be accepted in the assessment.

A practical reliability assessment presumes a combination of both assumptions. Expert judgements are typically applied on the evaluation of testing and operational data. Relying only on this so called hard evidence in the reliability estimation of digital systems is not rational. For example, information from the design features and the development process can provide qualitative evidence on the reliability characteristics of the system and discarding this evidence is not wise. Proper methods for supporting more extensive use of evidence of qualitative nature in the reliability assessment of digital systems should therefore be developed.

The paper presents a case study of reliability estimation of a motor protection relay. The estimation is based on Bayesian inference and Bayesian modelling framework is used to create a plausible and transparent reliability estimate for the software design of the relay. The case study is part of a long-term research effort to develop methodology and support for the reliability estimation of computer-based systems. The study is a follow-up to a similar case study presented in SAFECOMP 2003 [1]. The emphasis of the case study is in the development of the expert judgement process and in the feasibility evaluation of the method for the use of probabilistic safety assessments (PSAs) of nuclear power plants.

The aim of the case study has been in the methodological development of the assessment approach. The failure rate distributions in the end of the paper are given only as a demonstration of the informative results the assessment method can provide. The details related to the technical features of the case study system and to the evidence used have only been reviewed on a level necessary to test the functionality of the assessment method.

The paper starts with a description of the reliability assessment procedure and model in section 2. In section 3 the case study system, the system software and the development process of the software is described. The expert judgement process is explained in detail in section 4. The numerical results of the assessment are given in section 4. Finally, the conclusions of the assessment are stated in section 5.

2 Reliability Assessment Procedure and Model

Overview of the reliability assessment procedure is shown in Fig. 1. The assessment can be formulated as a pyramid where the bottom part is created by the information available for the system and the upper part by the reliability target of the software design.

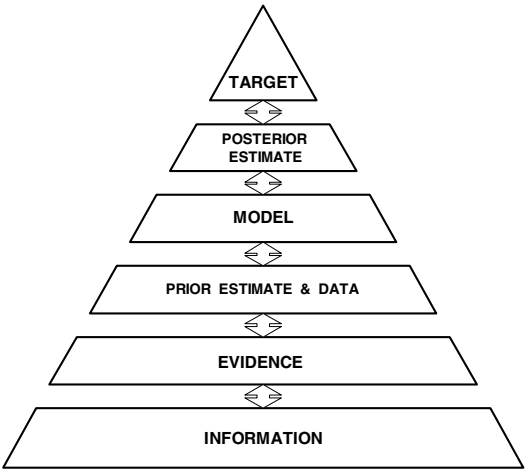


Fig. 1. Reliability assessment procedure

In the case study the system information is analysed and relevant parts of the information are nominated as evidence. The evidence may include, for example, artefacts of software development process, descriptions of software design features, analysis of software design, results of software testing and feedback from the operational use of the software. The reliability estimate of the software design is based on the evidence. Part of the evidence can be easily interpreted as quantitative figures and implemented as data to the assessment. Part of the evidence is more of qualitative characterisations of the software and to use the evidence in the reliability estimation extensive expert judgement is required.

The qualitative evidence is carefully analysed, discussed and finally transformed into quantitative reliability measure using a special expert judgement process. The reliability measure applied in the case study is the distribution of the failure rate of selected protection functions. The failure rate distribution estimated in the expert judgement process functions as the prior reliability estimate for the software design.

The prior estimate and data are implemented to a reliability model. The reliability model used in the case study is a homogenous Poisson model. Therefore, the appropriate conditional distribution of the number of software faults of the protection functions given the failure rate and the number of operating years is considered Poisson distributed as follows:

$$f(x|\lambda,T) \sim Poisson(\lambda T)\ , \tag{1}$$

where x is the number of software faults, λ the failure rate and T the estimated operating years of the selected protection functions in the assessment.

Bayesian inference is used to update the prior estimate to a posterior estimate. The posterior estimate represents the best reliability estimate of the software design with the given evidence. In Bayesian estimation it is important to keep the data and other evidence separate so that the data does not corrupt the prior estimation. The posterior estimate can be updated as additional evidence is collected for the software. The

posterior estimate can be compared with the reliability target set for the software design.

3 Case Study System

3.1 System Overview

The assessment object is motor protection relay REM 610. REM 610 is a versatile multifunctional protection relay mainly designed for protection of standard medium and large medium voltage asynchronous motors in a wide range of motor applications. The typical area of usage for REM 610 is motor applications from 500 kW to 2 MW. REM 610 handles fault conditions during motor start-up, normal operation, idling and cooling down at standstill. The relay can be used in both circuit-breaker controlled and contactor controlled drives. Different uses of the relay can be, for example, motors in pump, fan, mill and crusher applications. REM 610 was released to the market in the beginning of 2004 and a picture of the relay is shown in Fig. 2.

Fig. 2. REM 610

REM 610 contains ten different protection functions. From the ten protection functions four were selected to the actual reliability assessment:

- Thermal overload protection
- Start-up supervision
- Short-circuit protection
- Earth-fault protection

Detailed descriptions of the four selected protection functions can be found from the technical reference [2].

3.2 Software Overview

The size of the binary code of REM 610 is around 150-160 kilobytes and code executed on the main CPU is approximately 100-120 kilobytes. The software is programmed using C and assembly languages. Approximately half of the software is reused from the previous products and most of the code of the protection functions is legacy from the previous products. The software is implemented as firmware on the program memory.

3.3 Software Development Process

The main design phases of REM 610 and the main documents produced are following:

1. Market requirement phase => Market requirement specification (MRS)
2. Product requirement phase => Product requirement specification (PRS)
3. Functional design phase => Functional design specification (FDS)
4. Functional test phase => Functional test specification (FTS)

In the market requirement phase the hopes and needs of different customers are collected and documented to MRS. In the product requirement phase the requirement of MRS are analysed and discussed between the developers. Some of the requirements of MRS may be removed, some are accepted and some are accepted as modified. All accepted requirements are finally documented in PRS.

Based on the requirement of PRS the functions of the device are designed. The exact design of each function is documented in FDS. FDS contains all the required information for the implementation of the functions. FDS is actually a collection of documents each document describing a specific functional entity to be implemented. Each entity is programmed as an own programme segment and the programming is carried out mainly within the same design group that generated the design specification. In the assessment the programming of the protection functions is not considered as an independent development phase but is seen as an integrated part of the functional design phase.

The functional test phase involves the design of the test cases, running the tests and analysis of the results. The testing phase is carried out independent of the software designers of functional design phase. FTS exists as an electronic database containing the test cases of the different programme segments and the final programme. Results of the functional testing phase are recorded to the database.

4 Expert Judgement Process

Ten designers of the systems participated in the assessment. The experts formed four assessment groups representing the different development stages of REM 610. The experts were assigned to different assessment groups according to their actual position

and involvement in the development process. The expert judgement process of REM 610 contained the following sessions:

1. Expert training
2. Qualitative session
3. Quantitative session

In the expert training the experts received an introduction to the assessment, expert judgement process and applied methods. The expert training was followed by the qualitative and quantitative sessions. In the qualitative session the uncertainty and unpunctuality related to the design and documentation of the relay was identify and recorded. The recordings of the qualitative sessions were gathered and distributed to the assessment groups in the quantitative sessions. The objective of the quantitative session was to generate the actual reliability estimate for the four protection functions of REM 610. The training was given to all experts simultaneously. The qualitative and quantitative sessions were carried out for all assessment groups separately. The total number of sessions in the assessment was 13 and the duration of a single session was from two to three hours. The qualitative and quantitative sessions are described below in more detail.

4.1 Qualitative Session

A special form was used in the qualitative session. In the beginning the experts described simplified logical models for the protection functions with the required inputs and outputs. The general idea of a logical model is to illustrate the principle functionality of a function without anchoring the actual implementation. The manner of representation and the level of detail in the logical models were left for the assessment groups to decide, but it was guided that an independent assessor with a technical background should easily understand the models.

From the logical models different dependencies of the protection functions were identified and written down. In the identification the dependency of the protection functions on the other protection functions, hardware and new functionality implemented first time on REM 610 were particularly considered.

After completing the forms for the four protection functions the recordings were reviewed and compared to the descriptions of technical reference manual [2]. If any uncertainty or inconsistency was found in the recordings or descriptions, they were listed to the bottom part of the form for further analysis and discussion in the quantitative session.

4.2 Quantitative Session

The quantitative sessions were carried out after all qualitative sessions had been held. Failure rates of two different failure modes were estimated:

- Failure of trip – system does not launch a protection when needed
- Spurious trip – system launches an unnecessary protection

First, a target failure rate was considered for both failure modes. The target failure rate should be seen as a target design value for the software reliability of a new motor

protection relay. In other words, the target failure rate represents a value the software of the critical protection functions of a new relay should in average meet so that the end customer would be satisfied with the reliability of the new product. It should; however, be emphasised that at this point of the session the failure rate of REM 610 was not considered in any ways. The target failure rates were discussed only in a general level and the goal of the discussion was to create some kind of consensus among the assessment group members on the target reliability of software design of new motor protection relays in general.

Next, the recordings of the qualitative sessions of all assessment groups were distributed to the assessment group. The recordings were examined and discussed. In the examination the logical models of design phases of different protection functions were reviewed so that potential inconsistencies between the design phases could be notified. As guidance to the experts it was stated that a significant deviation in the logical models may imply misapprehension in the relay design, and therefore should decrease the expert's belief on the relay reliability. The dependencies, potential factors of uncertainty and unpunctuality listed by the other assessment groups were also reviewed to see if some important points had been forgotten by the group in the previous session. Finally, the own recordings of the assessment group were studied once more in detail. Key elements for and against the reliability of the protection functions of REM 610 were pointed out and written down on the form.

After listing the key elements the session proceeded to the actual quantification phase. Two numerical prior estimates were generated by each assessment group to represent the reliability of the four protection functions. The first prior estimate was generated for the failure of trip –failure mode and the second for the spurious trip – failure mode. Description of the quantification is following:

1. Based on the information given in the recordings and the remarks pointed out in the discussions each expert gave an individual and independent median value for the failure rate of the first failure mode. The expectation value of the target failure rate discussed in the beginning of the session could be used as a reference point, but this was left for the expert to decide.
2. Similarly 10 and 90 percentiles were given by the expert to reflect the uncertainty of his estimation.
3. The combined median, 10 percentile and 90 percentile values for the assessment group were generated simply by taking an average of the median, 10 percentile and 90 percentile values given by the individual experts.
4. A lognormal distribution based on the least squares method was fitted to the combined values.
5. The fitted lognormal distribution was presented for the assessment group for discussion.
6. Requested modifications to the lognormal distribution were made and the group accepted the estimate.
7. Similar procedure was carried out for the second failure mode.

In the generation of the group estimate the average values were used in point 3 to give equal weights to the estimates given by the individual experts. In point 4 the lognormal distribution and the method of least square were used so that it was possible to give an immediate response to the assessment group.

After creating the prior estimates for the four assessment groups the estimates were merged to form joint prior estimates. One joint prior estimate was created for each failure mode. Equal weights were given to the assessment groups when creating the joint prior estimates. The actual construction of the joint prior estimates was carried out in the calculation program WinBUGS [3].

5 Numerical Results

5.1 Prior Distributions

The recordings of different sessions are described in detail in [4]. Table 1 lists the key elements increasing and decreasing the confidence of the experts to the reliability of software design of REM 610.

The lognormal prior distributions of the assessment groups for two different failure modes are illustrated in Fig. 3-4. Fig. 3 presents the probability density functions of the failure of trip failure mode. Fig. 4 presents the corresponding probability density functions for the spurious trip failure mode.

5.2 Operational Data

The operational data used in the numerical calculations contains the estimated operating years and the reported software faults of REM 610 in the first year of operation. The operating years were estimated based on the sales figures of REM 610 using the following assumptions:

- After an order the relay was delivered to the customer in one week
- The time between delivery and commissioning phase is on average several months
- After commissioning the relay was used full-time for the rest of the year

The estimated number of operational years of REM 610 by the end of 2004 was 140 years. This number was used in the calculations as the operating years of REM 610. In the calculations all the devices were assumed to function in a single and similar operational profile.

Since start of the deliveries, following software faults of REM 610 have been found and reported by the customers by the end of 2004:

1. In case of external open command for circuit breaker, the circuit breaker failure protection function operated without current condition indicating circuit breaker status.
2. Trip circuit supervision function gave only local alarm indication to HMI, not to alarm relay output.

These were of limited influence and they were not related to the four protection functions under analysis in the case study. Therefore, the number of software faults used for the calculations is 0.

Table 1. Key elements increasing and decreasing confidence on the reliability of REM 610

MRS	PRS	FDS	FTS
Increasing confidence: – Logical models are consistent even to the level of detail – Designers are experts with strong experience on similar previous projects – Customers are provided with a support program to help them find the proper relay settings	*Increasing confidence:* – Thermal overload protection has been surprisingly well understood in the different phases	*Increasing confidence:* – The technology used and the functionality of the relay is well known to the designer – Thoroughly tested legacy code was used in the implementation of the protection functions	*Increasing confidence:* – Descriptions of the protection functions of the different design groups are consistent
Decreasing confidence: – Uncertainty related to the dependencies of the load testing – Uncertainty related to the increased number of serial communication protocols	*Decreasing confidence:* – The level of abstraction in the descriptions of the MRS group deviates from the other groups – The limitations of the testing equipment increases uncertainty	*Decreasing confidence:* – The calculation accuracy in the motor start-up supervision – RTD ambient temperature – Special PU-scale settings may cause unwanted behaviour from customer point of view	*Decreasing confidence:* – Sufficiency of performance testing – Uncertainty related to the testing of multiple simultaneous protection function

5.3 Posterior Distributions

Based on the prior estimates and the data posterior reliability estimates were calculated. The results are shown in Fig. 5.

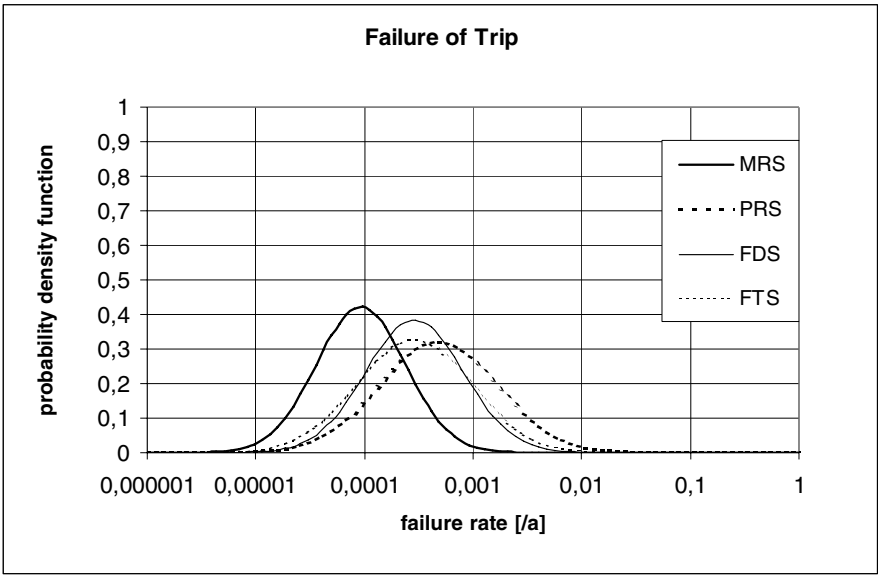


Fig. 3. Prior distributions (pdf) of the failure of trip failure mode

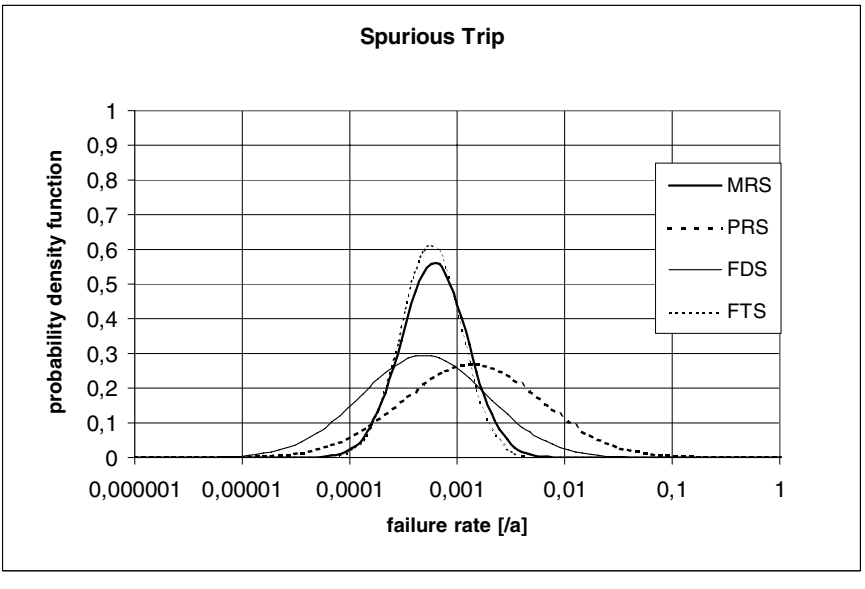


Fig. 4. Prior distributions (pdf) of the spurious trip failure mode

In the graphs 2,5-50-97,5 percentiles of the failure rate distributions are presented. Starting from the top, the first graph is the joint prior estimate generated from the prior estimates of the assessment groups given for the failure of trip failure mode. The

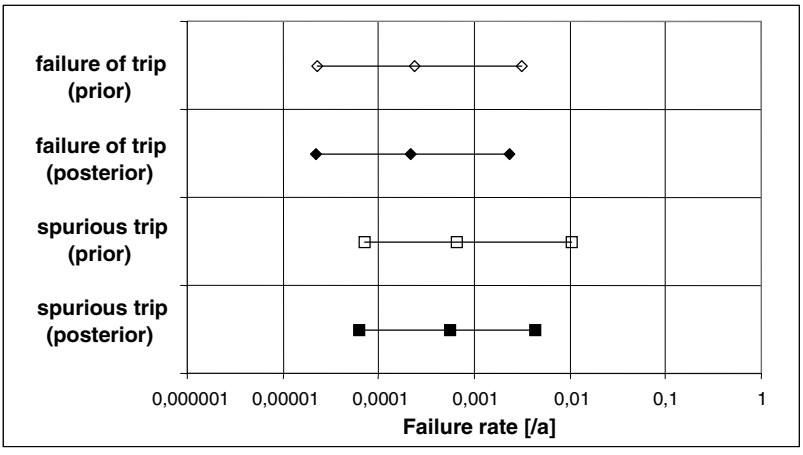


Fig. 5. Prior and posterior estimates of two failure modes

second graph is the calculated posterior estimate of the failure of trip failure mode. The third and fourth graphs present the corresponding prior and posterior estimates of the spurious trip failure mode.

From Fig. 5 it is easy to see that the prior estimates have a strong influence to the posterior estimates. The reliability estimates do not change very much from the prior estimates. The median values are only slightly shifted to the left and the variance of the posterior estimates is only modestly narrower than the variance of the prior estimates. Reason to the minor change in the estimates is the relatively small amount of operational data available in the estimation. The operational data was even diminished due to some conservative assumptions stated in the previous subsection. In the future, it will be interesting to see how the estimation changes while it is updated with new operational data collected for the relay.

6 Conclusions

In the paper a case study on the reliability estimation of the software design of motor protection relay REM 610 is presented. The purpose of the case study is to develop the expert judgement process of the assessment method and to evaluate the feasibility of the method for the use of PSA.

Based on the experiences the expert judgement process applied in the case study was considered functional. In the feedback the phased structure of the expert judgement process and the use of assessment groups instead of interviewing individual experts was appreciated by the assessment experts. With clear phasing it was easier for the experts to analyse the important reliability related factors of a specific protection function. Making the assessment as a teamwork created discussion on topics that might have been overlooked in pure personal interviews.

Giving quantitative estimates on the reliability of software design is not; however, an everyday task for typical software engineers. Therefore, better support should be

provided for the estimation task. This could be done for example by developing special reliability estimation tools that could be integrated as a part of the normal software development process requiring no extra effort from the engineers. Other way of supporting the task is to make the engineers more experienced and familiar with the reliability estimation by making the assessment a normal part of the software development process.

The feasibility of Bayesian method in the reliability assessment of computer-based systems has been addressed for example in [5] and [6]. The experience from the case study supports this. Different evidence is typically involved in the reliability assessment and with Bayesian method the evidence can be discussed and flexibly combined together.

Using the method for the purposes of PSA is particularly attractive. The method provides informative posterior probability distributions for the failure rates of the protection functions. These can be applied in the PSA models. The assessment is carried out from bottom-up (see Fig. 1) trying to create as plausible reliability estimate of the software design as possible. For practical reasons it is useful to discuss about the target failure rates, or reliability goals, in general in the assessment. This sets the experts at least to some extent to the same magnitude in their estimations making it easier to generate the joint prior estimates for the assessment groups.

In PSA the main interest is in generating an objective reliability estimate of the target system. Discussion on the reliability goals may influence the outcome of the assessment increasing the top-down (see Fig. 1) influence to the estimates. Therefore, it is highly important to maintain the qualitative statements of the assessment to find the reasoning behind the posterior estimates. This makes it easier for independent assessors to critically evaluate the validity of the results as demanded in [7] and [8].

Acknowledgement

The work presented in the paper was carried out in co-operation with ABB Substation Automation. The authors would like to thank the staff of ABB for their time and the expertise in the work.

References

1. Helminen, A., Pulkkinen, P.: Quantitative Reliability Estimation of a Computer-based Motor Protection Relay Using Bayesian Networks Using Bayesian Networks. In: Anderson, S., Felici, M., Littlewood, B. (eds.) SAFECOMP 2003. LNCS, vol. 2788, pp. 92–102. Springer, Heidelberg (2003)
2. REM 610 Motor Protection Relay - Technical Reference Manual, ABB Oy
3. Spiegelhalter, D., Thomas, A., Best, N., Gilks, W.: BUGS 0.5 Bayesian Inference Using Gibbs Sampling Manual (version ii), MRC Biostatistic Unit, Cambridge, pp. 1–59 (1996)
4. Helminen, A.: Case Study on Reliability Estimation of Computer-Based Device for Probabilistic Safety Assessment, VTT Research Report BTUO-051375, Espoo, pp. 1–29 (2005)

5. Littlewood, B., Popov, P., Strigini, L.: Assessment of the Reliability of Fault Tolerant Software: A Bayesian Approach. In: Proceedings of 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000), pp. 294–308. Springer, Berlin (2000)
6. Gran, B., Helminen, A.: A Bayesian Belief Network for Reliability Assessment., OECD Halden Reactor Project, HWR-649, Halden, pp. 1–26 (2001)
7. Pulkkinen, U.: Programmable automation systems in PSA. In: Radiation and Nuclear Safety Authority, Helsinki, pp. 1–19 (1996)
8. Littlewood, B., Strigini, L.: Software Reliability and Dependability: a Roadmap. In: State of the Art Reports given at the 22nd International Conference on Software Engineering, pp. 177–188. ACM Press, New York (2000)

A Reliability Evaluation of a Group Membership Protocol

Valério Rosset, Pedro F. Souto, Paulo Portugal, and Francisco Vasques

Faculdade de Engenharia, Universidade do Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
vrosset@fe.up.pt, pfs@fe.up.pt, pportugal@fe.up.pt, vasques@fe.up.pt

Abstract. We present a reliability evaluation of a group membership protocol (GMP), by computing the probability of violating the fault assumptions made in its proof. The evaluation of the reliability of a GMP is of paramount importance because group membership services are often used as building blocks in the design of fault-tolerant applications. The GMP that we consider here has been proposed for dual scheduled TDMA networks such as FlexRay, a protocol that is likely to become the de-facto standard for next generation automotive networks. Our study is carried out by modeling the GMP with discrete-time Markov chains. The models consider different fault scenarios, including permanent, transient and common-mode faults, affecting both channels and nodes. Furthermore we perform a sensitivity analysis to assess the influence of different parameters on the protocol's reliability. The results show that the GMP can achieve reliability levels in the range required for safety critical applications.

1 Introduction

It is widely accepted [1] that services such as group membership and distributed agreement facilitate the systematic development of safety-critical applications.

In [2] we presented a novel group membership protocol (GMP) for a new class of time division multiple access (TDMA) control protocols, which we call dual scheduled TDMA (DuST). This type of TDMA protocol is likely to become widely deployed, because FlexRay [3], a protocol being designed by a consortium of some of the main car manufacturers and automotive electronics companies, uses a DuST protocol and it is expected that FlexRay will become the *de facto* standard of the next generation network for automotive applications.

Because the GMP is intended to be used as a general service by safety-critical applications, it is important to ensure its correctness, ideally through a proof. However, any proof relies on fault assumptions and it ensures that the protocol behaves correctly only as long as the fault assumptions hold true. Therefore the reliability evaluation of the protocol is of paramount importance.

In this paper we evaluate the reliability of the GMP by equating it to the reliability of the assumptions made in its proof, i.e. the probability of these

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 397–410, 2007.

assumptions being true, an approach that, to our knowledge, was first proposed by Latronico, Miner and Koopman in [4]. The fault model used in this study includes both permanent and transient faults affecting both nodes and channels. In addition, we consider two classes of common-mode transient communication faults, faults that partition the network for the duration of the one message and error bursts.

In order to carry out this study we have developed several discrete-time Markov chain (DTMC) models, which were evaluated using PRISM [5], a probabilistic symbolic model checker. The results show that the GMP protocol can achieve reliability levels required by safety-critical applications, even for configurations of a small number of nodes and common-mode faults, by introducing a small modification to the original protocol.

The remaining of this paper is structured as follows. In Section 2, we describe the GMP and its maximum fault assumption (MFA). The fault model and the use of discrete-time Markov chains with PRISM to evaluate the reliability under this fault model are shortly described in Section 3. Then in Section 4 we present the experiments we carried out in this study, and analyze their results. Related work is covered in Section 5. In Section 6 we conclude with a summary of the main results.

All the models (or the scripts used for their generation) and the results of all the experiments are available at `http://web.fe.up.pt/~pfs/research/safecomp2007/experiments`.

2 Group Membership Protocol

In safety-critical applications, group membership is used mainly to keep track of the operating components (we will use the term nodes) of the system. Therefore, a group membership protocol comprises essentially two tasks: failure detection, by which a node determines the operational state of the other nodes in the system, and set agreement, by which non-faulty nodes agree on the state of all the nodes in the system.

The Group Membership Protocol (GMP) [2] under study comprises two phases: a failure detection phase (FD-phase) and a set agreement phase (SA-phase), as shown in Figure 1. During the FD-phase, each group member broadcasts a heartbeat message, and uses the heartbeats received to update a membership set. During the SA-phase, each group member broadcasts its own membership set, we call this message a *vote*, and applies a majority set function to its own membership set and to all membership sets it received in that phase, thus performing a majority voting.

The GMP was designed to take advantage of a new class of time division multiple access (TDMA) control protocols. In this class of protocols, rounds are split in two segments: one whose slots are statically scheduled like in conventional TDMA protocols, and another in which slots are allocated dynamically to nodes. Thus, the GMP executes the FD-phase during the statically scheduled segment of the communication round, and the SA-phase during the dynamically scheduled segment of the round, only when events that may lead to group membership change are detected. This approach allows for a more efficient use of the network

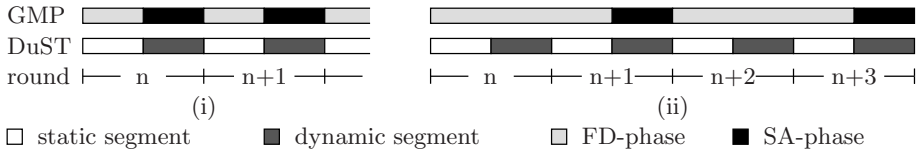


Fig. 1. GMP phases *vs.* DuST phases. Single-round (i) and multiple-round diagnosis periods (ii).

bandwidth than if set agreement was executed in every round, and therefore used the statically allocated segment. Figure 1(i) illustrates the relationship between GMP phases and the segments of DuST protocols. Figure 1(ii) shows a proposed extension of the GMP, in which the FD-phase rather than comprising the static segment of one communication round, comprises several static segments in consecutive communication rounds. We call this extension multiple-round diagnosis period GMP whereas we use the expression single-round diagnosis period GMP for the original protocol.

2.1 Maximum Fault Assumption

The majority set function is the core of the GMP. To mask faults and to ensure agreement among non-faulty group members, no more than half of the group members may fail between consecutive executions of the GMP. This condition is the maximum fault assumption (MFA). If it is violated, there is no guarantee that the GMP will satisfy its specifications.

2.2 Goals of the Reliability Evaluation

The reliability evaluation reported here started out with the general goal of quantifying the reliability of the GMP. The methodology we follow is to determine the reliability of the assumptions made in its proof, i.e. the reliability of the MFA.

From this general goal, more specific questions arose as the study progressed. The following, is a summary of the questions to which we try to answer:

1. What are the main factors that affect the GMP reliability?
2. What is the effect of using diagnosis period multiple of the communication round?
3. How do common-mode faults affect the GMP reliability?

3 Model

3.1 Fault Model

In arguing the correctness of the GMP in [2] we assumed that a node might fail by crashing or by omitting either to send or to receive messages. However,

because of well-known results on the impossibility of agreement in the presence of communication faults [6], we had to assume that the communication channels were reliable. For reliability evaluation studies, like this one, these results are not important. Thus, we now consider a more realistic fault model, by assuming that communication channels may also fail by crashing or by omitting to deliver messages, e.g. as a result of noise. Furthermore, we assume that these faults may be either permanent or transient. E.g., a node might fail to receive a message because it has not enough buffer space, but recover later.

Finally, in addition to single message faults, we consider two other types of transient fault in channels: common-mode faults and error bursts. Common-mode faults partition the network for the duration of one message, leading to what some authors [7] call strictly omission asymmetric faults, i.e. a scenario in which some nodes receive a message correctly, whereas other nodes receive no messages. (We assume that the error detection codes used by communication protocols are strong enough to detect virtually all communication errors.) Communication error bursts are caused by long duration electromagnetic interference (EMI) bursts that make communication all but impossible while it lasts.

3.2 Model

Because the GMP is round-based, we chose to use discrete-time Markov chains (DTMC) to model its behavior. Indeed, the use of DTMCs allows us to associate the state transitions with the passing of rounds, facilitating the evaluation of the MFA, which specifies the maximum number of faults in each protocol execution.

The GMP reliability models we have developed are parametric models, allowing to analyze the sensitivity of the protocol's reliability to different factors, and are comprised of several DTMCs: one for each node, and one for the communication network. Because of space limitations, we are unable discuss the models in this paper. Interested readers can find the models used in this study in `http://web.fe.up.pt/~pfs/research/safecomp2007/experiments`.

3.3 Reliability Evaluation

We have implemented the reliability models mentioned using PRISM [5], a probabilistic model checker, and determined the reliability of the GMP MFA, which can be expressed as a conjunction of two predicates:

1. The number of good channels, i.e. channels without permanent faults, which we denote n_c must be larger than 0, otherwise no communication will be possible.
2. In every protocol execution, the number of members that may fail, either transiently or permanently, which we denote n_t and n_p respectively, must not exceed the number of good members, i.e. that do not fail, which we denote n_g. This condition is necessary so that the majority function can mask faults.

Thus, the reliability of the GMP is given by the probability of the predicate

$$(n_c > 0) \wedge (n_g > n_p + n_t)$$

holding true in a time interval of a given duration.

4 Experiments Design

In order to answer each of the questions stated in Section 2, we have designed one set of experiments except for the case of common-mode faults, for which we considered two sets of experiments: one for faults that affect a single message and another for communication error bursts.

For each experiment, all frames have the same size. Furthermore, we assume that each node transmits 2 frames per communication round, and that the communication round is as short as possible. I.e. we assume that as soon as the last node in a round finishes the transmission of its second frame the first node starts the transmission of its first frame of the next round. Therefore, the duration of a round depends on the number of nodes, the frame size and the bit-rate. An alternative is to assume that a communication round has a fixed duration. We decided to model the protocol as described, because most automotive applications are real-time requiring high responsiveness.

Also, in all these experiments we consider that the fault inter-arrival times have an exponential distribution. This is a common assumption for faults with a physical cause such as electromagnetic interference (EMI). Furthermore, for the model parameters we use values from the automotive domain, the main application domain of DuST networks in the near future. These values are based on figures provided both in [4] and in [8]. Table 1 shows the values of the relevant parameters that we have used in virtually all experiments. Other parameters used in specific sets of experiments are presented when we describe those experiments in more detail.

Table 1. Parameter values used in virtually all experiments

Parameter	Values (units)
Number of nodes (N)	3, 4, 5, 6, 7, 8, 9, 10
Number of channels	1, 2
Frame Size (FS)	32, 128 (bytes)
Bit Rate	1, 10 (Mbps)
Bit-error rate (BER)	1E-6, 1E-7, 1E-8
Node Permanent Fault Rate (PHw)	1E-5 (faults/hour)
Channel Perm. Fault Rate (PCh)	1E-6 (faults/hour)

The number of nodes range from 3 to 10. The reason for this is threefold. First, to limit the number of configurations. Second, because the time required to evaluate the models increases almost exponentially with the number of nodes. This

is especially true for some models whose symmetry is limited. Third, and most importantly, the reliability of the protocol improves with the number of nodes, as we shall see, leveling off at the probability of failure of the communication channels.

We consider both single and duplicated channels. Indeed, whereas the use of duplicated channels affords higher reliability, its cost is higher. In most applications, a single channel will be used if its reliability is acceptable. Thus, it is important to evaluate the reliability of single channel configurations.

The frame size, the bit error rate (BER) and the bit-rate all affect the probability of transient communication faults. We consider frames of two sizes: 32 and 128 bytes. This is likely to cover most automotive applications. E.g. the header, trailer and synchronization bits in FlexRay lead to an overhead of around 16 bytes. Frames with size of 128 bytes are probably very rare in automotive applications, however we have chosen this value because longer frames have a higher probability of being affected by errors. For the BER we consider values in the range typical of copper medium. Although, optical fiber has a much lower BER, it is more expensive and therefore seldom used in the automotive domain. With respect to the bit-rate we consider 2 values: 1 Mbps and 10 Mbps. Again, these are typical of the automotive domain. E.g., the timing parameters specified in FlexRay were determined for a 10 Mbps. However, it mentions the possibility of specifying values for lower bit-rates.

We assume that the only cause for transient faults in the GMP are communication errors. This is because for the parameter values considered, cf. Table 1, the probability of transient faults in nodes ([8] mentions a fault rate between 1E-3/hour and 1E-5/hour) is at least two orders of magnitude lower than the probability of transient communication faults and does not affect the reliability of the GMP.

For each of the permanent fault rates, hardware and channels, we used a single value, 1E-5/hour and 1E-6/hour, commonly used in the literature. However, we have performed some additional experiments to evaluate the effect of these parameters on the reliability of the protocol.

All the results presented in this section are the result of the evaluation of the probability of violation of the maximum fault assumption after one hour. This is a standard time interval used for reliability evaluation. Also, all the results were obtained starting from one initial state in which all components are working properly and all nodes are members of the group.

4.1 Single-Round Diagnosis Period

The goal of these experiments was to determine the main factors that affect the reliability of the GMP proposed in [2].

First, we carried out an experiment for all combinations of the values of the parameters shown in Table 1.

In addition, in order to evaluate the effect of the permanent fault rates, of both channels and nodes, and to keep the number of configurations evaluated manageable, we run two additional sets of experiments. In one of them we varied

the channel permanent fault rate one order of magnitude above and below the values shown in Table 1 while maintaining the node permanent fault rate constant (and equal to the value shown in Table 1). In the second one, we varied the node permanent fault rate while maintaining the channel permanent fault rate constant. In each of these experiments we considered all possible combinations for the remaining parameters.

Data Analysis. Figure 2 shows the unreliability of the GMP for a communication network with two channels for bit-rates of both 1Mbps and 10 Mbps. The permanent hardware fault rate is 1E-5 per hour and the permanent channel fault rate is 1E-6 per hour.

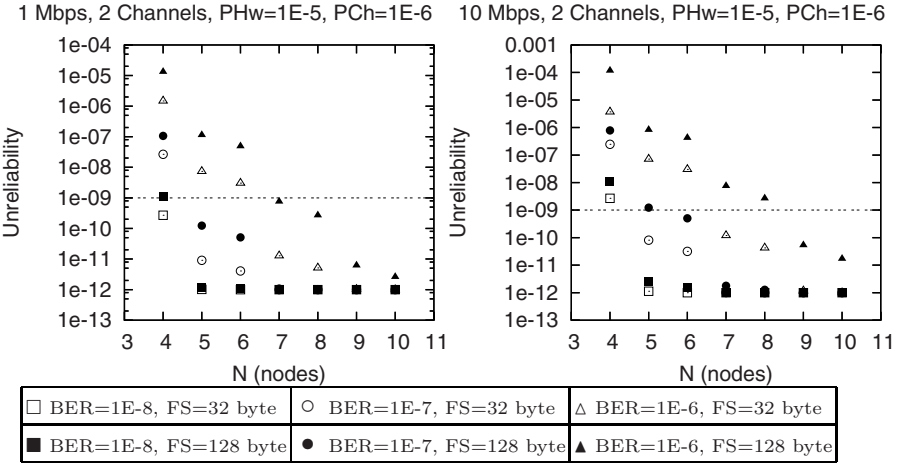


Fig. 2. Assumption violation probability in the first hour for the single-round diagnosis period GMP [2]

The figure shows that the GMP reliability is highly sensitive to transient communication faults. In particular, it depends heavily on the BER. The effects of the frame size are also visible but not as large. Indeed, larger frames lead to a higher transient communication fault, but this is partially compensated in our models by a decrease in the number of executions in the reliability evaluation interval as a consequence of the increase in the duration of the communication round. The effect of the number of protocols executions can be observed comparing the results for 1 Mbps and 10 Mbps channels. All other parameters being equal, the difference between configurations with 1 Mbps and 10 Mbps channels is on the number of protocol executions, and this leads to a reliability almost one order of magnitude lower for 10 Mbps channels, especially for configurations with a small number of nodes.

The GMP sensitiveness to transient communication faults is especially acute for configurations with a small number of nodes. This is because the GMP perceives transient communication faults as faults in nodes, and the smaller the

number of nodes more likely is that the GMP will be unable to gather a majority. As we increase the number of nodes, the reliability improves fast. Note however that when we increase the number of nodes by one, the reliability improvement is larger if the number of nodes before the increment is even. This is because in that case the increment will allow for the failure (real or perceived) of one additional node per protocol execution without violating the MFA, whereas if the number of nodes before the increment is odd it will not. Ultimately, for the factor values considered, the system reliability is bounded by the probability of both communication channels failing permanently. For example, for a BER of 1E-6 and frames with the size of 32 bytes, this limit is reached for configurations of 9 nodes.

From this discussion, it is clear that the best we can hope with a single channel configuration is an unreliability of 1E-6 – this bound is determined by the probability of the single channel failing permanently. (This is confirmed by the experiments for single channel configurations whose results we do not show.) It is obvious then that single channel configurations are not appropriate for safety-critical applications, and we do not consider them further in this paper.

These results hint that the probability of permanent channel faults affects significantly the reliability of the GMP. This is because, if a channel fails permanently, the protocol will operate with a single channel and therefore, the probability of a transient fault will be much higher. Indeed, the results we obtained from our experiments show that for networks with duplicated channels and the factor values shown in Table 1, a one order of magnitude variation in the probability of the permanent channel fault may lead to a variation up to two orders of magnitude in the GMP unreliability, for configurations with a large number of nodes. Conversely, for configurations with a small number of nodes and a high transient communication fault rate, e.g. 128 byte frames, the communication channel permanent fault rate has virtually no effect on the protocol reliability. On the other hand, our experiments have shown that for the values of all the other factors that we have considered, one order of magnitude change above or below the permanent hardware fault probability has no effect on the reliability of the GMP.

4.2 Multiple-Round Diagnosis Periods

One approach to improve the resiliency of the GMP to transient communication faults is to make the diagnosis period a multiple of the communication round and to diagnose a node as faulty only if it is affected by transient communication faults in more than some number, that we call *diagnosis threshold*, of these rounds.

In order to assess the efficacy of this approach we carried out an experiment in which we varied both the diagnosis period from 2 to 5 communication rounds, and considered thresholds of 1 and 2 messages.

We considered configurations with 3 to 6 nodes, only. This is because most configurations for single round diagnosis periods and a larger number of nodes already present an acceptable reliability for safety-critical applications.

Furthermore, as shown in Figure 3, below, the reliability increases with the number of nodes and configurations with 4 nodes already exhibit an unreliability below 1E-9.

For the BER we used only one value: 1E-6. This corresponds to the worst case for the values considered in Table 1.

Data Analysis. Figure 3 shows the reliability of the GMP for diagnosis periods multiple of the communication round, for communication network with two channels of 1 Mbps and 10 Mbps. The threshold used by the fault diagnosis algorithm is 1 message.

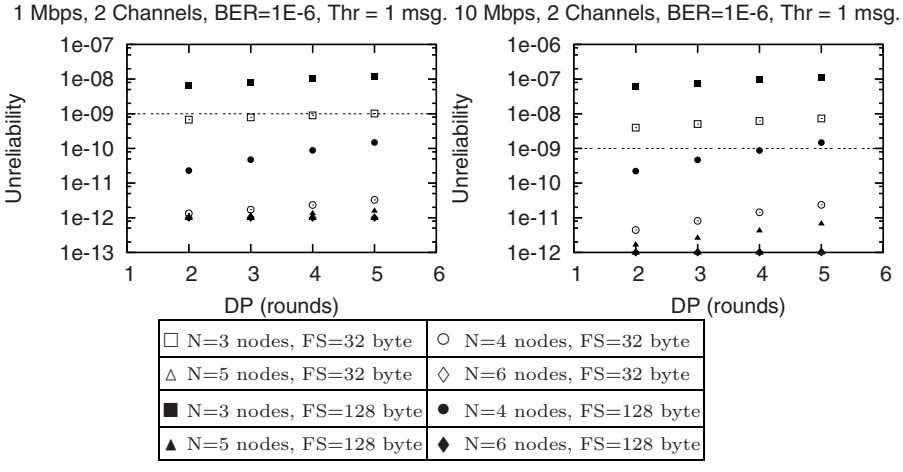


Fig. 3. Effect of the length of the diagnosis period on the MFA violation probability

These results show clearly the efficacy in terms of reliability improvement of using longer diagnosis periods, even with a diagnosis threshold of only 1 message. Note that this approach is especially advantageous for configurations with a low number of nodes, which have a relatively low reliability for single round diagnosis periods. Nevertheless, configurations with 3 nodes are still too unreliable to be used in safety-critical applications, except for configurations of 1 Mbps and frame sizes smaller than 32 bytes.

We have also evaluated the reliability of the GMP for a diagnosis threshold of 2 messages. As expected, the reliability improved compared with that obtained with 1 message thresholds, but 10 Mbps configurations with 3 nodes and 128 bytes still exhibit insufficient reliability for safety-critical applications.

4.3 Common-Mode Faults

In all previous experiments we considered only independent faults, i.e. faults that affect only one node. However, virtually all transient communication faults

are caused by EMI, which may be localized and therefore affect a subset of the nodes. Essentially, this corresponds to a partition of the network and may split the group members almost evenly.

In order to assess how this kind of fault affects the reliability of the GMP, we designed an experiment in which we modeled common-mode faults as a fault in every node. To keep the model tractable we made two simplifications. First, we considered that at most two common-mode faults may occur per diagnosis period. Second, we handled the loss of votes just like that of heart-beat messages: they are used to diagnose a node as faulty but are not otherwise taken into account in the gathering of a majority. This leads to a slight overestimation of the reliability.

Given that we did not find data on the rate of this kind of fault, we assumed that it is a fraction of the transient communication faults, that we call the *common-mode to independent fault ratio (cir)*. We considered a range between 0.1% and 10% for this parameter.

Furthermore, we have considered diagnosis periods of both 2 and 5 communication rounds, and thresholds of 1 and 2 messages, respectively.

Because of the execution time of the model is large, we run this experiment only for channels with a 1 Mpbs bit-rate, but nevertheless considered configurations with 3 to 10 nodes.

Data Analysis. Figure 4 shows the results of these experiments.

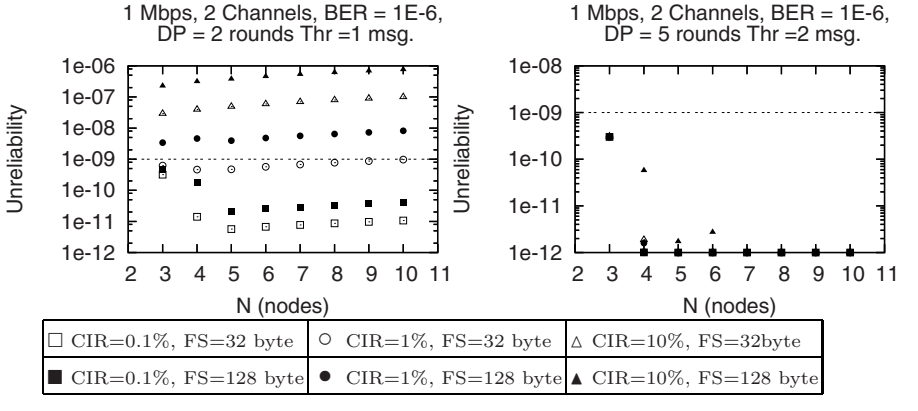


Fig. 4. Effect of the common-mode to independent faults ration in the MFA violation probability

As we might expect the reliability of the GMP is badly affected by the common-mode faults, especially for common-mode to independent fault ratios (CIR) above 1%, and for diagnosis thresholds of 1.

In contrast to the results of the previous experiments, for the values used, the unreliability increases with the number of nodes, especially for thresholds of one. This is partially an artifact of the way we model common-mode faults: for a

threshold of one, in diagnosis periods with common mode faults, one additional independent fault causes the violation of the MFA, and the probability of such a fault occurring increases with the number of nodes. For configurations with a threshold of two messages, the occurrence of one additional independent fault is not enough to cause a violation of the MFA. This partially explains the much better results obtained in experiments with that threshold, in spite of using a higher diagnosis period.

Another factor that accounts for the much better results of configurations with a threshold of two is that we limit the number of common-mode faults per diagnosis period to two. For configurations with a threshold of one, this does not affect the results because the occurrence of two common-mode faults leads automatically to the violation of the MFA. However, for configurations with a threshold of two this is not the case, leading to an overestimation of the reliability.

4.4 Communication Error Bursts

The goal of this experiment was to determine whether it is possible to make the GMP resilient to long duration EMI bursts by using diagnosis periods multiple of the communication round.

The conventional approach to deal with this kind of fault is to switch to a special operating mode such as the black-out mode in the TTP/C architecture. The main problem with this approach is that it requires that the nodes resynchronize. In the case of the GMP it would require the creation of a new group, usually a lengthy procedure.

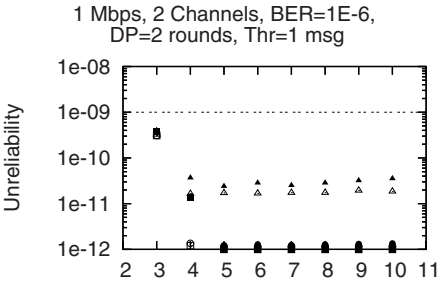
In order to tolerate faults that long, we propose to change the GMP so that it allows failure of the protocol execution in as many consecutive diagnosis periods as can be affected by the error burst. An alternative that we considered was to lengthen the diagnosis period to a duration longer than the error burst and to define a threshold that would tolerate the loss of as many messages as can be affected by the error burst. However, this alternative makes the GMP less responsive to permanent faults.

We characterize error burst faults with two parameters: the burst duration and the burst rate. In our experiment we used values of 50 ms for the burst duration, and varied the error burst rate (BR) one order of magnitude below and above 1E-4 per hour. These values are based on the figures provided in [8] for transient hardware faults in nodes caused by EMI.

To keep the model tractable we handle the loss of votes just like that of heart-beat messages, as described in the previous subsection.

We run this experiment for configurations of both 1 Mbps and 10 Mbps channels, diagnosis periods of 2 communication rounds and thresholds of 1 message. We chose these values because the results obtained in Section 4.2 show that they provide adequate reliability for almost all configurations.

Data Analysis. Figure 5 shows the results of these experiments.



□ BR=1E-5/hour, FS=32 byte	○ BR=1E-4/hour, FS=32 byte	△ BR=1E-3/hour, FS=32 byte
■ BR=1E-5/hour, FS=128 byte	● BR=1E-4/hour, FS=128 byte	▲ BR=1E-3/hour, FS=128 byte

Fig. 5. Effect of the error burst rate on the MFA violation probability

These results indicate that the approach proposed is effective to tolerate error bursts. Except for burst rates of 1E-3 per hour, the reliability is comparable to that for the corresponding configurations of the models without burst errors reported in Subsection 4.2. However, the reliability for configurations of 3 nodes is larger than that shown in Figure 3. This is because the way we model the effect of lost votes leads to an overestimation of the reliability.

Although at the time of writing we do not have values for 10 Mbps channels, we expect them to be in agreement with those obtained for 1 Mbps channels.

5 Related Work

In this paper we evaluate the reliability of a GMP through the computation of the reliability of its fault assumptions. To the best of our knowledge, this methodology was first proposed by Latronico, Miner and Koopman in [4]. Incidentally, [4] also uses a Group Membership Protocol (of the SPIDER architecture) as a case study. Although the methodology used is generally the same, there are some major differences between both works.

First, whereas [4] uses a continuous-time Markov chain (CTMC) model for modeling the GMP protocol we have chosen to use a discrete-time model. The use of a discrete-time model is more appropriate for round-based protocols that are executed repeatedly, such as group membership protocols. In particular, it allows to model the rounds themselves, facilitating the evaluation of assumptions on the number of faults in each round. This is important, because the fault tolerance of most round-based protocols relies on this kind of assumption. On the contrary, CTMC models make it virtually impossible to accurately model the rounds. This requires the analyst to use rough approximations in the evaluation of properties that depend on rounds. E.g. in [9] Latronico and Koopman also use CTMC models to evaluate the reliability of the group membership protocol of TTP/C. However, because of the limitation mentioned above, rather than

evaluating the reliability of TTP/C's fault assumption – that exactly one fault may occur within two-rounds, – they had to state a different fault assumption. Such an approach may raise some doubts regarding the outcome of the evaluation. Second, whereas in [4] the authors consider only independent fault models, in our work we consider also common-mode transient communication faults such as those caused by localized EMI or EMI bursts that affect several communication rounds. Finally, whereas [4] considers arbitrary, i.e. Byzantine, faults we do not. The reason for this is that whereas the SPIDER protocol tolerates these faults, the GMP we analyzed does not. Therefore, a single arbitrary fault will lead to violation of its fault assumptions. As pointed out in [4], faults outside the MFA are addressed by the concept of assumption coverage, as defined by Powell in [10].

6 Conclusions

We have evaluated the reliability of the assumptions made in the proof of a group membership protocol (GMP) especially designed to take advantage of a new TDMA protocol that is likely to become the *de facto* standard for next generation networks in the automotive domain.

In our study we considered several fault scenarios, including permanent, transient and common-mode faults, affecting both channels and nodes. Furthermore we performed a sensitivity analysis to assess the influence of different parameters on the protocol's reliability, using value ranges typical of the automotive domain.

The results obtained show that the protocol as originally proposed is highly sensitive to transient faults especially for configurations with a few nodes. However, reliability levels close to those required for safety critical applications can be achieved, through the use of duplicated channel and a diagnosis period multiple of the communication round. This same approach works also for common-mode transient communication faults, both those that partition the network for a frame duration and error-bursts that may affect several frames.

An interesting investigation direction that we plan to take is to extend this work to the class of protocols that use majority voting to mask faults. Indeed, many other fault-tolerant protocols rely on majority voting, and all models developed in this work focus on how faults affect the ability of gathering a majority. We believe that such an investigation may produce very general and very useful results.

Acknowledgments. The authors would like to thank the PRISM people in general and David Parker in particular for making available the PRISM prototype that supports symmetry reduction.

Valério Rosset would like to acknowledge the financial support of the Portuguese Fundação para a Ciência e a Tecnologia through the Scolarship FCT - BD 19302/2004.

References

1. Rushby, J.M.: Bus Architectures for Safety-Critical Embedded Systems. In: Proceedings of the 1st International Workshop on Embedded Software, pp. 306–323 (2001)
2. Rosset, V., Souto, P.F., Vasques, F.: A Group Membership Protocol for Communication Systems with both Static and Dynamic Scheduling. In: 6th IEEE International Workshop on Factory Communication Systems (WFCS06), IEEE Computer Society Press, Los Alamitos (2006)
3. Makowitz, R., Temple, C.: FlexRay: A Communication Network for Automotive Control Systems. In: 6th IEEE International Workshop on Factory Communication Systems (WFCS06), IEEE Computer Society Press, Los Alamitos (2006)
4. Latronico, E., Miner, P., Koopman, P.: Quantifying the Reliability of Proven SPIDER Group Membership Service Guarantees. In: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), IEEE Computer Society, Los Alamitos (2004)
5. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative Analysis with the Probabilistic Model Checker PRISM. Electronic Notes in Theoretical Computer Science 153(2), 5–31 (2005)
6. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
7. Azadmanesh, M., Kieckhafer, R.: Exploiting Omissive Faults in Synchronous Approximate Agreement. IEEE Transactions on Computers 49(10), 1031–1042 (2000)
8. Peti, P., Obermaisser, R., Ademaj, A., Kopetz, H.: A Maintenance-Oriented Fault Model for the DECOS Integrated Diagnostic Architecture. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), IEEE Computer Society Press, Los Alamitos (2005)
9. Latronico, E., Koopman, P.: Design Time Reliability Analysis of Distributed Fault Tolerance Algorithms. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05), pp. 486–495. IEEE Computer Society Press, Los Alamitos (2005)
10. Powell, D.: Failure Mode Assumptions and Assumption Coverage. In: 22nd Annual Intl. Symposium on Fault-Tolerant Computing (FTCS '92), pp. 386–395. IEEE Computer Society Press, Los Alamitos (1992)

Bounds on the Reliability of Fault-Tolerant Software Built by Forcing Diversity

Kizito Salako

The Center for Software reliability
City University
Northampton Square EC1V 0HB, The United Kingdom
kizito@soi.city.ac.uk

Abstract. Fault tolerance via diversity has been advocated as a viable defence against common-mode failure in safety critical systems. The consequences of using diverse, redundant software components in fault-tolerant, software-based systems have been the subject of much research. In particular, Littlewood and Miller showed analytically how "*forcing*" diversity between redundant software components might achieve higher expected system reliability than if these components failed independently. But their theorems concerned very special scenarios. This paper examines various lower and upper bounds on the expected reliability of systems built by "forcing diversity" and specify conditions for forced diversity to guarantee improved upper bounds on the system's expected probability of failure on demand (*pfd*).

1 Fault Tolerance Via Failure Diversity

Fault tolerance, the ability of a system to continue to operate in the presence of faults, can be achieved by building systems composed of multiple, functionally equivalent software components. These components may be combined in a "*1-out-of-N*" or a "*voted*" configuration[1]. At the heart of these approaches for achieving fault tolerance is the notion of failure diversity: the notion that redundant components may fail on different system demands. For instance, consider a 1-out-of-2 system. If its component software versions fail on mutually exclusive sets of demands then the 1-out-of-2 system does not fail on any demand. Diversity is possible because the process of software development is random; running the process multiple times may not result in the same software being produced each time. The resulting software components may fail on different demands.

In developing a 1-out-of-2 system, diversity between the failures of the component software versions can occur naturally. This can be achieved by software

[1] A parallel redundant (1-out-of-N) system is one in which correct system functioning is assured provided at least one channel functions correctly; in a voted system, correct system functioning is assured if a majority of channels function correctly. Many other architectures are possible, but here we are interested in the simplest practical scenario where evaluation problems arise.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 411–416, 2007.

development teams, one for each software component, developing their respective software components while being perfectly isolated from each other. The teams, as a consequence of being isolated, develop their respective software versions independently. Diversity that arises in this way is *unforced*/*natural*:it arises naturally from the randomness that is inherent in the software development processes. In a seminal paper by Eckhardt and Lee, [1], it was shown that *unforced* diversity results in average system reliability that is no better than if the component software versions were expected to fail independently. This is partly summarised by the statement "For a 1-out-of-2 system, independently developed software components cannot be expected to fail independently". This result can be understood by appreciating that though the teams (as a result of being isolated) are independent in how they develop their respective versions, the teams are likely to find similar tasks in the development process difficult. So, if a software version made by one team fails on a given demand, this indicates that the demand might be a difficult demand and so we may expect the other team's version to fail on the same demand.

Eckhardt and Lee [1] demonstrated, via their model (EL model, for short), why we might expect coincident failure between independently developed, software versions. However, Littlewood and Miller [2] demonstrated that *forcing* diversity can result in 1-out-of-2 systems with average system reliability that is better than if the software components failed independently. *Forcing* diversity is the deliberate requirement that the functionally equivalent software components be developed in different ways. Littlewood and Miller [2] argue that this is achievable because there is usually a choice of alternative *methodologies* that can be employed in developing software. For instance, there are various methods of fault removal, various equivalent algorithms, various programming languages and development environments that may be utilised in developing software components. By using *different* or *diverse methodologies* the development teams may have different view points of the issues in software development and consequently be less likely to make the same mistakes. So, some tasks during development may be easy for one team and difficult for another team. We can formalise this notion of varying difficulties between the teams. Making mistakes during the development of software can impact on whether the software successfully handles or fails to handle some subset of it's *input* or *demand* space[2]. So, as a consequence of mistakes that may be made by a team in the development process, the software developed by the team fails, deterministically, on some subset of the system's input space. Since we do not know which mistakes will be made by the team in development, we are also uncertain about the failure behaviour of the developed software on any given demand is also random. This means that we can associate with each demand a probability that the team develops software that fails on this demand. The *difficulty function*[3] for a team is then a mapping, defined on

[2] The input space is the set of all inputs that the system is required to operate on. This may be determined from the system's functional requirements.

[3] *Difficulty function* is a term coined by Littlewood and Miller [2] to name a concept introduced by Eckhardt and Lee [1].

the set of system demands, that assigns to each system demand a probability that the development team creates software that fails on that demand.

In this paper, by reasoning in terms of the LM model, we shall discuss bounds on the reliability of 1-out-of-2 systems and give rather general theorems about the effects, on system reliability, of forcing diversity. The layout of the paper is as follows. Section 2 introduces terminology and parameters necessary for our analysis. Section 3 details various bounds on the the reliabilities of systems built by forcing diversity. Finally, Sect. 4 outlines conclusions and relevant ongoing work.

2 Forced *and* Unforced Diversity: Relevant Parameters and Terminology

The following measures and terminology will be useful.

Probability of failure on demand (pfd): The uncertainty about which demand will be submitted next to the system in operation allows us to define the following reliability measure. The ***probability of failure on demand*** (*pfd*) of a system is the probability that the system fails on a demand submitted to it, selected at random, from its operational environment. In addition, the expected *pfd*, q_A, of a system developed using methodology A, is the probability that a system, developed by a team employing methodology A, fails on a demand submitted to it at random from its operational environment;

Homogeneous and Heterogeneous Systems: Recall, from Sect. 1, that in the development of a 1-out-of-2 system diversity may be forced or unforced (allowed to occur naturally). A development process of a 1-out-of-2 system that employs "Forced diversity" is one in which each of the redundant channels is built by employing a unique development methodology. For instance, given two methodologies called 'A' and 'B' say, 'A' may be used to develop one of the channels and 'B' may be used to develop the other channel. The resulting system is a "*Heterogeneous*" AB system with expected *pfd*, q_{AB}. Alternatively, if the component software channels are built by employing the same development methodology, methodology 'A' say, then diversity was not forced or occurred naturally and we say the resulting system is a "*Homogeneous*" AA system with expected *pfd*, q_{AA}.

Indifference: There are at least 2 contexts in which a system developer may be *indifferent*. The alternative methodologies, when used in development processes, may be such that they have equal likelihood of being used by the system developer and we say that the developer is "***indifferent between the methodologies***".

Diversity Parameter (γ_A): Given a methodology 'A' the quantity $\gamma_A := q_{AA}/q_A^2$ is an indicator of how effective unforced diversity, arising by employing methodology 'A' in building a 1-out-of-2, homogeneous system, is. To see this consider

the tightest bounds on γ_A. These bounds represent the extreme cases of methodology A's usefulness when used in building homogeneous systems.

- ***Maximum benefit from diversity with Methodology A*** : At one extreme, $\gamma_A = 1$, say. That is, $q_{AA} = q_A^2$. *This is the maximum benefit possible and we say that values of γ_A close to 1 are "good"*. Recall, from Sect. 1, that both teams use the same methodology and, as a consequence, have very similar difficulty functions.;
- ***No benefit from diversity with Methodology A*** : At the other extreme,the development teams always build software with identical failure behaviour and $q_{AA} = q_A$. Therefore,$\gamma_A = 1/q_A$. The software developed always have the same failure set. This means that the difficulty function only takes the values 0 and 1. So, *unforced diversity is expected to bring no improvement and we say that γ_A values close to $1/q_A$ are "bad"*.

The gamma factors are related to *beta factors*, used in common-cause failure models, since $\gamma_A \times q_A = \beta_A$.

3 Bounds on the Reliability of Systems Built by Forcing Diversity

The following, fairly general, scenario is necessary for all of the theorems that follow. Consider an assessor of some 1-out-of-2 systems who

1. has a choice of 2 software development methodologies, A and B say;
2. has estimates for the expected *pfds*, q_A and q_B, of single version systems;
3. may have evidence to support beliefs about whether, or not, $\gamma_A \leq \gamma_B$. Knowledge of the values of q_{AA} and q_{BB} is not necessary. Instead, knowledge of the beta factors β_A and β_B will suffice.[4];

Under these conditions all of the following theorems hold. Proofs for these theorems are available at [3].

Theorem 1. *Under **indifference between methodologies** A and B, building a 1-out-of-2 system by forcing diversity results in a system with expected pfd that is no greater than the expected pfd obtained if a homogeneous system were built, instead. That is,*

$$q_{AB} \leq \frac{q_{AA} + q_{BB}}{2} \ .$$

The following theorem can be stated as a corollary of Theorem 1. A similar theorem, derived in [2], requires a system assessor to know that $q_{AA} = q_{BB}$. This is not required here.

Corollary 2. *If the expected pfds of homogeneous systems are equal (that is, $q_{AA} = q_{BB}$) and a system developer is **indifferent between the methodologies** then*

[4] Beta factor estimates are used in industry and, in particular, are recommended in industry standards for safety-critical systems. For instance, EN 61508 (2008).

building a 1-out-of-2 system by forcing diversity results in a system with expected pfd that is no greater than the expected pfd obtained if a homogeneous system were built, instead. That is, $q_{AB} \leq q_{AA}$ *or* q_{BB}.

The following theorem states upper and lower bounds on q_{AB}.

Theorem 3

$$\begin{aligned} \max\left(0, q_A q_B\left(1-\sqrt{(\gamma_A-1)(\gamma_B-1)}\right)\right) &\leq q_{AB} \\ &\leq q_A q_B\left(1+\sqrt{(\gamma_A-1)(\gamma_B-1)}\right) \\ &\leq q_A q_B \sqrt{\gamma_A \gamma_B}\ . \end{aligned}$$

The following theorem gives a necessary and sufficient condition for $q_A q_B \sqrt{\gamma_A \gamma_B}$ to be the least upper bound on q_{AB}.

Theorem 4. *For the existence of a pair of difficulty functions such that*

$$q_{AB} = q_A q_B \sqrt{\gamma_A \gamma_B}$$

it is necessary and sufficient that $\gamma_A = \gamma_B$.

From Theorem 3 the following special cases can be identified in which *not* forcing diversity is preferred, from the view point of the upper bound on q_{AB}.

1. If $\gamma_A = 1$ and $\gamma_B = 1$ and $q_A \leq q_B$ then $q_{AA} \leq q_{AB} \leq q_{BB}$. Therefore, a homogeneous 'AA' system should be built.
2. If $\gamma_A = 1$ and $\gamma_B = 1/q_B$ and $q_A \leq q_B$ then $q_{AA} \leq q_{AB} \leq q_{BB}$. Therefore, a homogeneous 'AA' system should be built.
3. If $\gamma_A = 1/q_A$ and $\gamma_B = 1/q_B$ and $q_A \leq q_B$ then $q_{AA} \leq q_{AB} \leq q_{BB}$. Therefore, a homogeneous 'AA' system should be built.

4 Conclusions

The expected *pfd* of a 1-out-of-2 system is always no greater than the expected *pfd*s of either one of the system's redundant channels. In this sense "*Diversity is always a good thing*". However, when trying to build a more reliable, 1-out-of-2 system instead of a single version system, a choice may be made between building either a heterogeneous or homogeneous 1-out-of-2 system. That is, a choice has to be made between whether, or not, to force diversity in the development process of the 1-out-of-2 system. The focus of this paper has been to state bounds on the expected *pfd*, q_{AB}, of a 1-out-of-2 system built by forcing diversity. In particular, under rather general conditions, some theorems state preferences, in terms of upper bounds on q_{AB}, between forced and unforced (natural) diversity during the development of 1-out-of-2 systems. The preferences are obtained by deriving relevant bounds, which are functions of γ_A, γ_B, q_A and q_B, on the value of q_{AB}. In practice, some of these results depend on estimates of the values of

γ_A, γ_B, q_A and q_B will be known. It is not unusual to have estimates for component software (single version systems) that are to be integrated into a larger system. Indeed, for COTS-based systems it is sometimes possible to have empirical estimates of the reliabilities of the software components. Also, the use of beta factors in estimating system reliability is commonplace.

Theorem 1, in particular, states a development process scenario, for a 1-out-of-2 system, for which diversity should be forced. Suppose a system developer has a choice of 2 development process methodologies. Upon employing the methodologies to build homogeneous 1-out-of-2 systems the developer does not know which methodology results in a system with better expected *pfd*. As a result, methodologies are equally likely to be chosen by the developer. That is, the developer has no preference or is indifferent between the methodologies. In practice, it is common for system developers to be indifferent between methodologies especially when there is no evidence to support a clear choice between them. Theorem 1 states that under these conditions a heterogeneous 1-out-of-2 system should be built. Diversity should be forced in the system's development process.

Practical judgments about whether to force diversity or not should also take into account economic issues, such as the cost of implementing a regime of forcing diversity. However, this paper has focused on making the judgment about whether to force diversity or not purely in terms of the expected *pfd*s of the systems built. As an extension of the current work an economic/utility model could take into account our theorems and results, in a bid to aid system developers in decision making.

The parameters in this paper are averages so care should be taken in their use. In particular, a system's actual *pfd*, when built, will normally differ from the expected *pfd*, given the system's development methodologies.

Acknowledgment

This work was supported by the DIRC project ('Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems') funded by UK Engineering and Physical Sciences Research Council (EPSRC). Also, this work benefited from useful discussions with Lorenzo Strigini.

References

[1] Eckhardt, D.E., Lee, L.D.: A theoretical basis for the analysis of multiversion software subject to coincident errors. IEEE Transactions on Software Engineering SE-11, 1511–1517 (1985)

[2] Littlewood, B., Miller, D.R.: Conceptual modelling of coincident failures in multiversion software. IEEE Transactions on Software Engineering SE-15, 1596–1614 (1989)

[3] Salako, K.: Appendices (2007), http://www.csr.city.ac.uk/staff/salako/papers/Safecomp2007/

A Tool for Network Reliability Analysis

A. Bobbio[1], R. Terruggia[1], A. Boellis[2], E. Ciancamerla[2], and M. Minichino[2]

[1] Dip. di Informatica, Università del Piemonte Orientale, 15100 Alessandria (Italy)
[2] ENEA, CR Casaccia, sp Anguillarese 301, 00060 Roma (Italy)
bobbio@mfn.unipmn.it, terruggia@di.unito.it,
{boellis,ciancamerla,minichino}@casaccia.enea.it

Abstract. A network is a structure where any couple of nodes is normally connected by different independent paths, thus making the structure intrinsically reliable. For this reason, many natural, social and technological systems organize in the form of networks. A tool for network reliability analysis, where different approaches are tested and compared, is in progress. The paper presents some preliminary results of a scalable benchmark, that includes different network structures.

1 Introduction

A peculiar feature of the modern organization of the human life is the increasing level of interconnectivity. Critical Infrastructures (CI) form a framework of interconnected and interdependent networks and systems [6]. One relevant property of networks, that make them a preferential structure both in natural and technological systems, is that the connection between any two nodes of the network can be usually achieved through a number of redundant paths, thus making the connection intrinsically reliable.

The increased complexity of real networked systems requires new analytic approaches to afford their new scale and predict their behavior. There is a wide literature concerning complex networked system analysis [1,4,2] that looks at the structural properties of the graph and the rules governing the aggregation of its nodes, with the aim to predict networked system behavior. Two topological classes are relevant for representing CIs: Random Graph (RG) networks and Scale Free (SF) networks [1,4].

In the last decades, Binary Decision Diagrams (BDD) [3] have provided an extraordinarily efficient method to represent and manipulate Boolean functions, and we present a tool that makes use of the BDD's to encode the network connectivity function, and to compute the network reliability.

2 Network Definitions and Characterization

A network can be represented in the form of a graph $G = (V, E)$, where V is the set of vertices (or nodes) and E the set of edges (or arcs). If the network elements are binary entities (up or down) the network connectivity can be expressed as a Boolean function. By assigning a probability measure to each element of

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 417–422, 2007.

the graph to be up or down, we define as *two-terminal* reliability (or (s,t)-reliability) the probability that two nodes communicate with each other by at least one path of working edges.

Regular, Random and Scale Free networks - Networks can be classified according to different topological structures. Regular networks are represented by graphs that can be described by means of defined geometrical properties and can be very well suited for scalable benchmarks. An example of symmetric regular network with $N = 5$ nodes and connectivity degree $k = 2$ is shown in Figure 1 (the arcs are oriented counterclockwise).

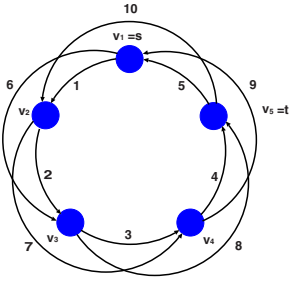


Fig. 1. A regular network with $N = 5$ and $k = 2$

The most relevant topological classes of networks for representing CI structures, are RG and SF networks. In RG networks, the degree distribution $P(k)$ follows a Poisson distribution; the network degree is characterized by an average value with a given standard deviation. In SF networks, the degree distribution $P(k)$ follows a power-law [5], so that a small number of very highly connected nodes (the hubs) are linked to a large number of poorly connected nodes.

3 Tool Implementation

A software tool for network reliability analysis has been developed. The tool accepts in input a graph with various formats, and evaluates its reliability by means of two algorithms based on BDD.

Binary Decision Diagrams - A BDD represents a Boolean expression by means of the Shannon's decomposition principle. If F is a Boolean function on the variables $x_1, x_2, \ldots, x_n$ the following (Shannon) decomposition holds:

$$F = x_1 \wedge F_{x_1=1} \vee \overline{x}_1 \wedge F_{x_1=0} \tag{1}$$

Furthermore, if we assign to every variable x_i a probability p_i of being true $(1-p_i$ false), we can compute the probability $P\{F\}$ of the function F by applying recursively Equation 2.

$$P\{F\} = p_1 P\{F_{x_1=1}\} + (1-p_1) P\{F_{x_1=0}\} = P\{F_{x_1=0}\} + p_1 (P\{F_{x_1=1}\} - P\{F_{x_1=0}\}) \tag{2}$$

In particular, two different algorithms will be discussed, the first based on the search of the minpaths, the second based on a recursive visit of the graph.

3.1 (s-t)-Reliability Via Minpath Analysis (MPA)

Given a network $G = (V, E)$ and two nodes (s,t), the following definition holds.

Definition 1. A (s,t)-*path* is a subset of elements (arcs and/or nodes) that guarantees the source s and the termination t to be connected if all the elements of this subset are up. A path H is a *minpath* if a subset of elements in H does not exist that is also a path.

Let $H_1, H_2, \ldots, H_h$ be the h minpaths between s and t. All the elements of a minpath must be working, therefore the elements of the minpath are logically connected in AND. The network connectivity $S_{(s,t)}$ can be represented as the logical OR of its minpaths, since it is sufficient that any one of the minpaths is operational to make the network working.

$$S_{(s,t)} = H_1 \vee H_2 \vee \ldots H_h \tag{3}$$

The two terminal reliability can be calculated as:

$$R_{(s,t)} = P\{S_{(s,t)}\} = P\{H_1 \vee H_2 \vee \ldots H_h\} \tag{4}$$

Formula 4 shows that the network reliability can be evaluated as the probability of the union of non-disjoint events.

Example 1 - Visual inspection on the network of Figure 1, assuming s=v_1 and t=v_5, shows that the graph possesses 5 minpaths (listed in order of their rank):

$$H_1 = \{6, 8\};\ H_2 = \{6, 3, 4\};\ H_3 = \{1, 7, 4\};\ H_4 = \{1, 2, 8\};\ H_5 = \{1, 2, 3, 4\} \tag{5}$$

Replacing (5) into (3), the network (s,t)-connectivity can be expressed as:

$$S_{(s,t)} = (6 \wedge 8) \vee (6 \wedge 3 \wedge 4) \vee (1 \wedge 7 \wedge 4) \vee (1 \wedge 2 \wedge 8) \vee (1 \wedge 2 \wedge 3 \wedge 4) \tag{6}$$

The connectivity expression (6) is a Boolean function for which the Shannon's decomposition can be applied and the related BDD constructed.

3.2 (s-t)-Reliability by Graph Visiting Algorithms (VA)

The BDD representation of the (s-t) connectivity of a graph, can be directly derived without passing from a preliminary search for the minpaths or mincuts. In [7], an algorithm is proposed that generates the BDD directly, via a recursive visit on the graph, without explicitly deriving the Boolean expression.

Given a graph $G = (V, E)$ and two nodes (s,t), the algorithm starts from s and visits the graph (according to a given but arbitrary visiting strategy) until t is reached. The BDD construction starts recursively once the sink node t is reached.

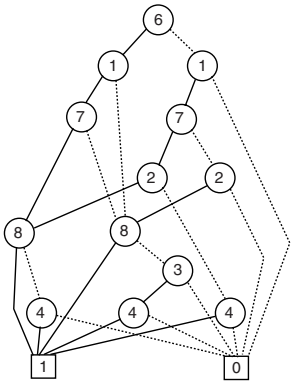


Fig. 2. The final BDD of the network of Figure 1

Example 2 - The construction of the BDD by means of the described visiting algorithm is illustrated step by step on the same symmetric directed network of Figure 1. The graph is visited according to the progressive (but arbitrary) number assigned to the nodes. Starting from the source node $\mathtt{s} = v_1$ the visit proceeds along nodes v_2, v_3, v_4, until the sink node $\mathtt{t} = v_5$ is reached. Once the sink is reached, the construction starts with the BDD representing the last visited arc 4. Then, the algorithm makes one step back to node v_3 where it finds a bifurcation and builds the BDD $((4 \wedge 3) \vee 8)$. Going one step back with the recursion, the algorithm revisits node v_2 and builds the BDD $(((4 \wedge 3) \vee 8) \wedge 2) \vee (4 \wedge 7)$. Finally, in the last step, the algorithm visits the source node v_1 and builds the BDD for the complete `(s,t)`-connectivity function.

$$S_{(\mathtt{s},\mathtt{t})} = (((((4 \wedge 3) \vee 8) \wedge 2) \vee (4 \wedge 7)) \wedge 1) \vee (((4 \wedge 3) \vee 8) \wedge 6) \qquad (7)$$

It is easy to see that formula (7) contains replicated terms that are simplified and folded automatically during the construction of the BDD reported in Fig. 2.

4 Scalable Benchmark

A scalable benchmark has been carried out to evaluate and to compare the efficiency of the software tool against the increasing complexity of different network topologies: namely Symmetric, Lattice, RG and SF networks. In all the experiments, a failure probability equal to $p = 0.9$ is uniformly assigned to all the arcs, only. The experiments have been run on a SUN machine with a dual AMD Opteron 2.3 GHz processor, and with 4 GB of memory.

Regular networks - Symmetric networks (Figure 1) depend on two parameters, the number of nodes N and the degree k. In all the experiments `s` and `t` are two adjacent nodes. Table 1 reports the results for increasing values of N and k.

Table 1. Benchmark Results on symmetric networks

N	k	# arcs	# minpath	# BDD nodes VA	# BDD nodes MPA	(s,t) reliability
10	2	20	55	33	274	0.97137
10	4	40	755	2156	755	0.99980
10	7	70	19900	355650	62764	1
20	2	40	6765	73	43041	0.96096
20	3	60	83929	594	*n.a.*	0.99793
30	2	60	832040	113	*n.a.*	0.95065
35	2	70	n.a.	133	*n.a.*	0.94554

The results obtained from the two algorithms differ only in the number of generated BDD nodes, since they use a different ordering for the variables. In Table 1, the column `(# BDD nodes VA)` reports the number of BDD nodes generated in the visit algorithm, while the `(# BDD nodes MPA)` reports the number of BDD nodes generated in the minpath algorithm and the value of the reliability.

Random Graphs and Scale Free Networks - Tables 2 and 3 display the results obtained on RG and SF networks, respectively. Networks are grown with an increasing number of final nodes N, while keeping constant the number of connections ($k = 2$), and, for RG networks only, the probability of attachment ($p = 1$). The first generated node is assumed as the source `s` and the last generated node as the sink `t`.

The first three columns of each table report the final number of nodes, the final number of edges and the clustering coefficient. Columns 4, 5, 6 report, respectively, the number of minpaths, the number of the nodes of the BDD with VA algorithm, the number of the nodes of the BDD with MPA algorithm and the reliability value.

Table 2. Benchmark Results on Random Graphs

#nodes	#arcs	clustering coefficient	# minpath	# BDD nodes VA	# BDD nodes MPA	(s,t) reliability
20	72	0.1632	2046	316360	10032	0.98885
30	112	0.1502	119033	*n.a.*	708801	0.98906
40	152	0.1379	6757683	*n.a.*	*n.a.*	*n.a.*

Table 3. Benchmark Results on Scale Free networks

#nodes	#arcs	clustering coefficient	# minpath	# BDD nodes VA	# BDD nodes MPA	(s,t) reliability
20	74	0.5869	263	10409	266	0.98010
30	114	0.3927	4707	*n.a.*	239788	0.98001
40	154	0.3888	73680	*n.a.*	*n.a.*	*n.a.*

All the results in Tables 2 and 3 have been obtained on incremental networks, in the sense that the network with $M > N$ nodes has been obtained by starting from the network with N nodes and adding the remaining $M - N$ nodes.

5 Conclusions

The paper has presented a tool for the reliability analysis of networks by means of two independent algorithms via the construction of a BDD. The ability of the algorithm to cope with different network topologies of increasing complexity has been tested in a series of preliminary experiments. The limits of the technique have been impinged and further experimentation is needed with more powerful computing facilities.

Acknowledgements. The research presented in this paper was partially motivated by the participation in the ENEA project Cresco (http://www.cresco.enea.it/), in the EU projects IRRIIS (http://www.irriis.org/) and CRUTIAL (http://crutial.cesiricerca.it/). The authors want to thank Hari Suman, Rajdeep Shyam and Antonio Spina that have contributed to the research documented in this paper.

References

1. Albert, R., Barabasi, A.L.: Statistical mechanics of complex networks. Review Modern Physics 74, 47–97 (2002)
2. Boccaletti, S., Latora, V., Chavez, M., Hwang, D.: Complex networks: structure and dynamics. Physics Reports 424, 175–308 (2006)
3. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
4. Newman, M.E.: The structure and function of complex networks. SIAM Review 45, 167–256 (2003)
5. Newman, M.E.: Power laws, Pareto distributions and Zipf's laws. Contemporary Physics 46, 323–351 (2005)
6. Rinaldi, S., Peerenboom, J., Kelly, T.: Identify, understanding, and analyzing critical infrastructure interdependencies. IEEE Control System Magazine (December 11–25, 2001)
7. Zang, X., Sunand, H., Trivedi, K.: A bdd-based algorithm for reliability graph analysis. Technical report, Department of Electrical Engineering, Duke University (2000)

DFT and DRBD in Computing Systems Dependability Analysis

Salvatore Distefano and Antonio Puliafito

University of Messina, Engineering Faculty,
Contrada di Dio, S. Agata, 98166 Messina, Italy
{salvatdi,apulia}@ingegneria.unime.it

Abstract. Many alternatives are available for modeling reliability aspects, such as reliability block diagrams (RBD), fault trees (FT) or reliability graphs (RG). Since they are easy to use and to understand, they are widely spread. But, often, the stochastic independence assumption significantly limits the applicability of such tools. In particular this concerns complex systems such as computing systems. Also more enhanced formalisms as dynamic FT (DFT) could result not adequate to model dependent and dynamic aspects. To overcome this limitation we developed a new formalism derived from RBD: the dynamic RBD (DRBD). In this paper we compare the DFT and the DRBD approaches in the evaluation of a multiprocessor distributed computing system. Particular attention is given to the analysis phase, in order to highlight the capabilities of DRBD.

1 Introduction

There are several approaches to represent and analyze system reliability. From these particular mention is for the *combinatorial models*, i.e. high level specific reliability/availability modeling formalisms such as *reliability block diagrams* (RBD) [1], *fault trees* (FT) [2] and *reliability graphs* (RG). Although RBD, RG and FT provide a view of the system close to the modeler, they are defined on the *stochastic independence* assumption among components. They do not provide any elements or capabilities to model reliability interactions among components or subsystems, or to represent system configuration changing, aspects conventionally identified as *dynamic*. In particular these remarks concern computing systems: load sharing phenomena could affect the network availability; standby redundancy and maintenance policies could be considered in the management; interference or inter-dependence among components could arise (wireless devices, sensors, ...); common cause failures could group electric devices (power jumps, sudden changes of temperature, ...). These argumentations awakened the scientific community to the need of new formalisms as the *dynamic fault trees* (DFT) [3]. DFT extend static FT to enable modeling of time dependent failures, introducing new dynamic gates. But, using DFT it is hard to compose dependencies reflecting characteristics of complex and/or hierarchical systems, to define customizable redundancy schema or policy, to represent load sharing

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 423–429, 2007.

and to adequately model reparability features. To overcome these lacks in reliability modeling, in [4,5] we have defined a new reliability/availability modeling notation named *dynamic reliability block diagrams* (DRBD), by extending the RBD formalism. In this paper we in depth compare the two approaches, evaluating the reliability of a computing system taken as case study. More specifically, in section 2 the DRBD notation is briefly introduced. The motivating example and the corresponding DFT and DRBD models are described in section 3, then, the analysis of the two models is reported in section 4. Lastly, section 5 provides some final considerations.

2 DRBD Overview

DRBD extend the RBD formalism to the systems' dynamics representation.Two are the key points of DRBD: the *unit dynamics* and the *dependency* concept. In a DRBD model each unit is characterized by a variable *state* identifying its operational condition at a given time. The evolution of a unit's state (*unit's dynamics*) is characterized by the *events* occurring to it, as depicted in Fig. 1. The states a generic DRBD unit can assume are: *active* if the unit works without any problem, *failed* if unit is not operational, following up its failure, and *standby* if it is operable but not committable. The *events* represent transitions between states: a *failure* models changes from active or standby to the failed states, a *wake-up* switches from standby to active states, the *sleep* from active to standby states, the *reparation* from failed to active state, the *adep-switch* represents transitions between two active states and *sdep-switch* between two standby states.

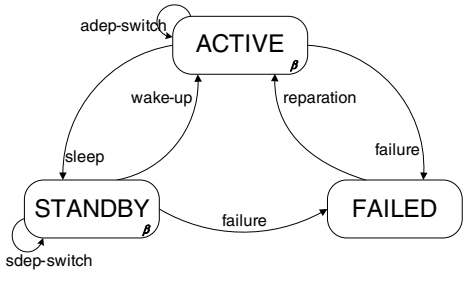


Fig. 1. DRBD unit's states-events finite state automata

The main enhancement introduced by DBRD is the capability to model dependencies among units concerning their reliability behaviours. A dependency establishes a reliability relationship between two units, a *driver* and a *target*. Informally a dependency works as follow: when a specified event, named *action* or *trigger*, occurs to the driver, the dependency condition is applied to the target. This condition is associated to a specific target event, named *reaction*. When the satisfied dependency condition becomes unsatisfied, the target unit comes back

to the fully active state. The dependent state (standby or active) is characterized by the *dependency rate*, weighting, in terms of reliability, the dependence of the target unit from the driver. This corresponds to the dormancy factor α of DFT ($\beta = 1 - \alpha$), but β could assume values greater than one. A dependency is characterized by the action (trigger) and the reaction events. Four types of trigger and reaction events can be identified: wake-up (W), reparation (R), sleep (S) and failure (F). Combining action and reaction, 16 types of dependencies are identified. The concept of dependence is exploited in DRBD as the basis to represent dynamic reliability behaviors. Details on the dynamics aspects modeling capabilities of DRBD can be found in [4,5].

3 The Multiprocessor Distributed Computing System

The scheme reported in Fig. 2 describes the multiprocessor computing system taken from literature [6,7] used for comparing the DFT and the DRBD approaches. It is composed by two computing module: CM_1 and CM_2. Each of them contains one processor (P_1 and P_2 respectively), one memory (M_1 and M_2) and two hard disks: a primary (D_{11} and D_{21}) and a backup disk (D_{12} and D_{22}). Initially, the primary disk is used for storing data while the backup disk is accessed only periodically for updating operations. If the primary disk fails, it is replaced by the backup disk. The computing modules are connected by the bus N; moreover, P_1 and P_2 are energized by the power supply PS: the failure of PS forces P_1 and P_2 to fail. M_3 is a spare memory replacing M_1 or M_2 in the case of failure. If M_1 and M_2 are operational, M_3 is just kept alive, but it is not accessed to load/store any data by the processors. When M_1 or M_2 fail, M_3 substitutes the failed unit. In order to work properly the multiprocessor computing system of Fig. 2 requires that at least one computing module (CM_1 or CM_2), the power supply PS and the bus N are operating correctly. A computing module is operational if the processor (P_1 and P_2), one between the local memory (M_1 and M_2) and the shared memory M_3 and one disk (D_{11} or D_{21} for CM_1 and D_{12} or D_{22} for CM_2) are not failed.

The DFT modeling the multiprocessor computing system is depicted in Fig. 3(a) as it is in [6,7], while Fig. 3(b) reports the corresponding DRBD model.

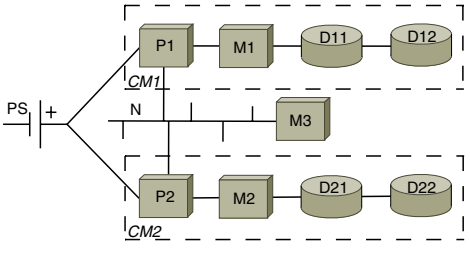
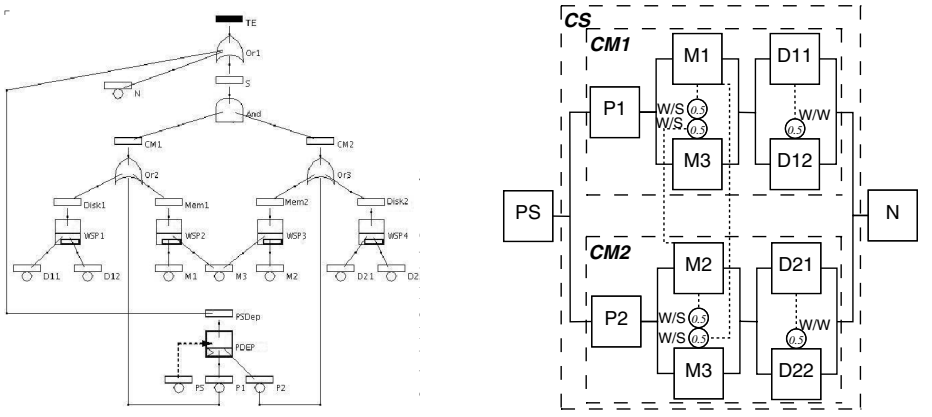


Fig. 2. Schematic representation of the Multiprocessor Distributed Computing System



(a) DFT (b) DRBD

Fig. 3. Multiprocessor Distributed Computing System models

The DFT is composed by a FDEP and four WSP. The FDEP gate models the dependency among the power supply PS and the two processors P_1 and P_2. Since the power supply PS energizes the P_1 and P_2 processors, the failure of PS does not imply the failure of P_1 and P_2 thus we represent this behaviour in the DRBD by a series between each processor and PS. The backup disks D_{12} and D_{22} are considered as spare units of the primary disks D_{11} and D_{21} respectively, thus D_{11} and D_{21} drive WSP1 and WSP4 DFT gates in the control of D_{12} and D_{22} respectively. The disks management policy is represented in DRBD by a wake-up/wake-up dependency: when the primary disks D_{11} and/or D_{21} are operational, the backup disks D_{12} and/or D_{22} respectively are partial active, maintaining the backup. The level of activity of the dependent components is numerically translated into the DFT by dormancy factor α and into the DRBD by the dependency rate β, related to α by $\beta = 1 - \alpha$.

The *partly-loaded standby redundancy policy* applied to the M_1, M_2 and M_3 memory units, is represented by the WSP2 and WSP3 DFT gates: if M_1 or M_2 fail, M_3 is activated. Wake-up/standby dependency are instead exploited to model the redundancy policy managing the memories. Such dependency must be applied to M_3 if and only if both M_1 and M_2 are at the same time operational; when one of these fails, M_3 must switch to the fully active state. To realize this condition two wake-up/standby dependencies, from M_1 to M_3 and from M_2 to M_3 are *series composed* [5]: when both are simultaneously satisfied the component M_3 is placed in standby, otherwise M_3 is active.

The other DFT gates are static: the internal events $DISK_1$ and $DISK_2$ represent the failure of the corresponding CM_1 and CM_2 storage blocks, while MEM_1 and MEM_2 represent the computing modules' memory block failure. The failure of the processor (P_1 and P_2) or of the memory block (MEM_1 and MEM_2) or of the disk block ($DISK_1$ and $DISK_2$) drives to the failure of the corresponding computing module ($CM1$ and $CM2$ internal events). Finally, if

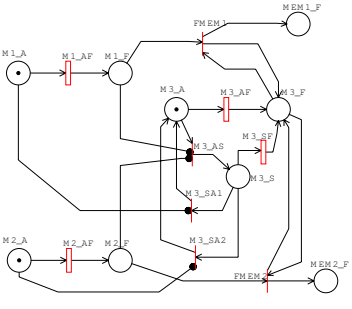
both the computing modules fail, or the power supply PS goes down, or the bus N fails, the overall system fault occurs, represented in the DFT as the top event TE. It corresponds to the series among the two computing modules parallel, the power supply PS and the bus N in the DRBD in Fig. 3(b).

4 The Analysis

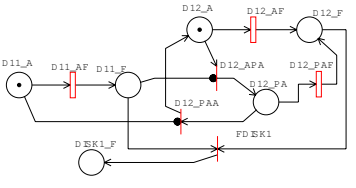
The example described and modeled in section 3 has been studied in depth by analyzing the overall system reliability cdf trend in time, knowing the components' reliability cdfs or the corresponding failure rates. All the components have been modeled by a constant failure rate λ as reported in Table 1, characterizing exponential reliability cdfs or *memoryless* systems.

Table 1. Parameters of the multiprocessor computing system

Component	λ	α	β
N	2		
P_1, P_2	500		
PS	6000		
$D_{11}, D_{21}, D_{12}, D_{22}$	80000	0.5	0.5
M_1, M_2	30		
M_3	30	0.5	0.5
CM_1, CM_2		0.9	0.1



(a) Memory Subsystem



(b) Disk Subsystem

Fig. 4. GSPNs modeling the memory (a) and the disk (b) blocks

Initially the multiprocessor computing systems DRBD reported in Fig. 3(b) is subdivided in three independent subsystems: the first, static, is composed by the series among the power supply PS and the bus N, series connected with the parallel between the two computing modules CM_1 and CM_2, that are the other two subsystems. Since these latter are identical, it is possible to study only one computing module subsystem and then apply the parallel structure equation to obtain the reliability of the two computing modules' parallel. A computing module subsystem is further subdivided into the series of three blocks: the processor,

Table 2. Unreliability results of the multiprocessor computing system analysis

Time	DBNet	DRPFTproc	Galileo	DRBD
1000	0.006009	0.006009	0.006009	0.006009
2000	0.012245	0.012245	0.012245	0.012245
3000	0.019182	0.019183	0.019183	0.019183
4000	0.027352	0.027355	0.027355	0.027354
5000	0.037238	0.037241	0.037241	0.037240

the memory and the disk. The memory and the disk blocks are dynamic parts. To study these dynamic parts the generalized SPNs (GSPNs) reported in Fig. 4 are exploited. These are generated by applying the DRBD-GSPN mapping algorithm specified in [5]. Thus, analyzing the two GSPNs through the WebSPN tool [8] and putting all together by applying the RBD structure equations, the results shown in the last column of Table 2 are obtained.

By the same way, the DFT model depicted in Fig. 3(a) corresponding to the motivating example discussed, has been analyzed in [7] by exploiting three different tool: DBNet [7], DRPFTproc [9] and Galileo [3]. The first analyzes the DFT by translating it into a dynamic Bayesian network (DBN) and therefore by solving the DBN. DRPFTproc is based on modularization and conversion to stochastic well-formed nets (SWN) of the dynamic gates, tracing back the problem to a SWN solution. Galileo approaches the problem by firstly modularizing it, then solving the obtained modules by exploiting binary decision diagrams and CTMCs.

Also the results obtained by such analysis are summarized in Table 2, where they are compared to the DRBD approach. Table 2 reports some system unreliability probabilities calculated in specific time instants. The time is expressed in hours. These results demonstrate and validate the effectiveness of the DRBD approach, providing consistent values for all the tests.

5 Conclusions

In this paper, the effectiveness of the DRBD methodology in the evaluation of system dynamic reliability is demonstrated. An in depth comparison among the DRBD and the DFT methodologies is investigated in the paper by exploiting a case study reporting the modeling and the analysis of a multiprocessor computing system, for which evaluate the overall system reliability. The results obtained allow to identify DRBD as a valid alternative in dynamic reliability/availability evaluation scenario.

References

1. Rausand, M., Høyland, A.: System Reliability Theory: Models, Statistical Methods, and Applications, 3rd edn. Wiley-IEEE (2003)
2. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. U.S. Nuclear Regulatory Commission, NUREG-0492, Washington D.C. (1981)

3. Sullivan, K.J., Dugan, J.B., Coppit, D.: The galileo fault tree analysis tool. In: Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, pp. 232–235. IEEE, Madison, Wisconsin (1999)
4. Distefano, S., Puliafito, A.: System modeling with dynamic reliability block diagrams. In: Proceedings of the Safety and Reliability Conference (ESREL06), ESRA (2006)
5. Distefano, S.: System Dependability and Performances: Techniques, Methodologies and Tools. PhD thesis, University of Messina (2005)
6. Malhotra, M., Trivedi, K.S.: Dependability modeling using petrinets. IEEE Transaction on Reliability 44(3), 428–440 (1995)
7. Montani, S., Portinale, L., Bobbio, A., Raiteri, D.C.: Automatically translating dynamic fault trees into dynamic bayesian networks by means of a software tool. In: Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, pp. 804–809. IEEE Computer Society Press, Los Alamitos (2006)
8. Scarpa, M., Puliafito, A., Distefano, S.: A parallel approach for the solution of non Markovian Petri Nets. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 196–203. Springer, Heidelberg (2003)
9. Bobbio, A., Franceschinis, G., Gaeta, R., Portinale, L.: Parametric fault tree for the dependability analysis of redundant systems and its high-level petri net semantics. IEEE Trans. Softw. Eng. 29(3), 270–287 (2003)

Development of Model Based Tools to Support the Design of Railway Control Applications

István Majzik, Zoltán Micskei, and Gergely Pintér

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
{majzik,micskeiz,pinterg}@mit.bme.hu

Abstract. The development standard for railway control software requires several design and verification methods. To support these methods we elaborated a coherent set of tools based on UML state diagrams. To avoid the problems of the ambiguous UML semantics, we propose a subset of UML state machines that includes the practical modeling concepts and has well-defined operational semantics elaborated definitely for software engineers. Based on this formalism we developed a tool chain supporting (i) the simulation of the behavior specified by the state diagram, (ii) static checking the completeness and consistency of the specification, (iii) generation of the C source of the application control flow, (iv) automatic construction of test cases on the basis of structural test coverage criteria and (v) automatic construction of the source code of run-time verification procedures that aim at checking high-level safety properties.

Keywords: UML state diagrams, static checking, assertions, test generation.

1 Introduction

The software development standard EN 50128 for computerized railway control systems prescribes several methods and techniques that are mandatory or highly recommended at a given safety integrity level. Among others, we can mention static analysis, failure assertion programming and structural testing. Some of these methods can be supported by automatic tools. However, tool application in the design and verification phases needs a clear understanding of the formalisms and models that form the input of the tools and allow the interpretation of the results produced by these tools.

In the framework of a project supported by the Hungarian National Office of Research and Technology[1] we have elaborated a coherent set of tools and techniques based on UML statecharts models. UML statecharts as a modeling language could be effectively used in the design of event-triggered state-based control systems, however, its use in safety critical applications was hindered by its ambiguous standard semantics (often reported in the literature) and usability problems appeared in connection with the formal semantics developed so far. Accordingly, we have elaborated a subset of UML 2.0 state machines called *Precise Statecharts*. It includes all practically meaningful modeling concepts and has a fully defined operational semantics that was

[1] Project nr. GVOP - 3.1.1 - 2004 - 05 - 0523/3.0.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 430–435, 2007.

elaborated definitely for *software engineers* (instead of computer scientists). This formalization step allowed the development of the following support tools: (i) *simulation* of the control flow specified by precise statecharts, (ii) *static checking* the completeness and consistency of statechart specifications, (iii) *generation of the C language representation* of the control flow of the application, (iv) automated construction of *test cases* on the basis of structural test coverage criteria and (v) automatic construction of the source code of *run-time verification procedures* that aim at checking high-level safety properties. The application of these tools (as they are not certified) is intended to increase the confidence of the designers in the correctness of the design. This is assured by automating those systematic and tedious (typically error-prone) construction and verification procedures that were executed manually. The following parts of this short paper present the general concepts of formalization and tool development.

2 A Formal Operational Semantics for UML Statecharts

Automated checking and implementation of systems specified by statecharts needs to assign *unambiguous meaning* to statecharts. As the UML standard does not define an unambiguous operational semantics, multiple approaches have been published in the literature. Common drawbacks of these formalisms are that (i) they focus on quite restricted subsets of statechart artifacts, (ii) they were developed for old versions of UML and (iii) their model-checking point of view results in their extensive use of mathematical formalisms that are hard to understand for software engineers. We decided to define a formal semantics that is both mathematically well established and easily applicable in the engineering practice. The key steps of the approach are as follows: (i) first we establish the *syntactic foundations* by introducing the concept of precise statecharts and defining their metamodel; then (ii) we outline a formalism for explicit representation of *compound transition and activity structures*, finally (iii) we outline the definition of *semantics* for statecharts by a Kripke transition system and the translation of this formalism to easy to understand *imperative algorithms*.

The *syntactic basis* of our approach is the UML statechart metamodel. In order to rule the complexity, we considered junction and choice pseudostates, history vertices and facilities for embedding state machines as advanced constructs (shorthand notations that make the visual modeling more comfortable) and defined a set of formal transformation rules for their substitution with basic concepts [1]. From this point on, we will focus on those statecharts that contain basic constructs only (or are mapped to this form) and we will call these statecharts as *precise statecharts* (PSC).

From the point of view of software engineers, the main deficiencies of *precise semantics definition* are related to the transition structures (fork, join), and the ordering of activities when a compound transition is fired. We introduced the following concepts:

- *Transition conglomerates*: There are many cases when some transitions of a statechart can not be considered in isolation, e.g., input and output transitions of a fork pseudostate. In order to facilitate consistent and uniform discussion of these compound transition structures we introduced the concept of *transition conglomerates* grouped into six classes (Fig. 1).

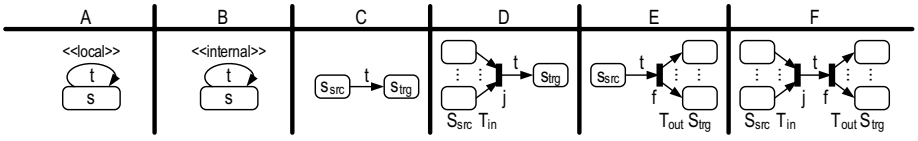


Fig. 1. Transition Conglomerate Classes

- *LCA and priority*: In the context of transition conglomerates we were able to provide a precise and intuitive formalization of the concepts of least common ancestor region (LCA), priority and conflict relations.
- *Compound activity structures*: When firing a transition conglomerate several activities are to be performed in a single step. The UML standard does not introduce a concept for handling compound activity structures either; however the unambiguous representation of strict subsequence relations and possibilities for parallel execution would be highly beneficial for exploiting the parallel processing capabilities of modern computing platforms. In order to overcome this weakness of the standard we introduced a formalism representing compound activity structures based on PERT graphs.

We introduced these concepts into the metamodel of precise statecharts and specified the *formal operational semantics* of UML 2.0 statecharts by a Kripke transition system (KTS). A KTS is defined by a three-tuple of states, state labeling function and the labeled state transition relation. In our case the states (statuses of the statechart) represent (i) the actual *configuration* of the statechart, (ii) the actual *evaluation of variables* and (iii) the actual *phase of operation* (e.g., run-to-completion step, terminated, etc.), thus the state labels represent three-tuples of this information. A transition of the KTS corresponds to a *step* between two statuses of the statechart representing (i) the *event* that triggered this step, (ii) the *transition conglomerates* that were *fired* in the step and (iii) the *compound activity structure* performed in the step.

We translated this declarative definition to easy to understand *imperative algorithms* implemented in the Microsoft AsmL executable specification language [1]. This way our approach is (i) mathematically well established (due to the rigorous formal semantics) and (ii) easily applicable in the engineering practice (due to the translation to imperative algorithms).

3 The Tool Chain

The AsmL imperative algorithms belonging to the formal semantics formed the basis of a *statechart simulator tool.* The modeler can construct an event sequence and the simulator calculates the transition conglomerates to be fired and the PERT graphs corresponding to the activity structures.

Besides this simulator, our tool set contains four other tools that are presented in the following subsections.

3.1 Static Checking of the Statechart Models

The compact representation of UML statecharts (including hierarchy, parallelism and nontrivial model elements) is a typical source of insufficiencies. Here we mention

only the following three criteria [2]: (i) *completeness* – in order to prevent the state machine from dropping an event, in all possible statuses of KTS, for all possible events, there must be a step transition defined which is triggered by the event; (ii) *determinism* – in each status, each event should trigger only a single step transition; and (iii) *reachability* – all states are reachable from the initial configuration.

These criteria can be formalized on the basis of the formal semantics, since it "unfolds" hierarchy, parallelism and the nontrivial model elements. Checking these criteria directly on the KTS requires the explicit generation of the KTS (i.e., the state space), which may lead to state space explosion in case of complex models. Accordingly, we adopted the approach of *static checking*: the criteria are adapted to *syntactic terms* (constructions of model elements) of precise statecharts to be able to check them directly on the model. We identified the hazardous scenarios belonging to the violation of the above criteria and on the basis of "reverse" semantic rules we defined static patterns that lead to these scenarios. The *consistency and completeness checker tool* applies a pattern matching algorithm to identify the concerned model elements.

3.2 Automatic Implementation of UML Statecharts

The *implementation* of a complex formalism like a statechart is definitely a nontrivial issue. The usual approaches (e.g., nested *switch* statements) are unable to handle such constructs as state refinement or parallel execution. Even the popular QHsm technique [3] is restricted to non-concurrent statecharts. The basis of our code generation is our *metamodel* of precise statecharts and our *algorithms* defining the operational semantics. We mapped the abstract concepts to the specialties of resource-constrained embedded systems: we substituted the complex AsmL algorithms (that calculate a possibly parallel execution order of various activities) with simple algorithms that calculate a *single valid sequence of activities*; substituted the recursive or mutually recursive function structures with *iterative algorithms*; introduced *compact representation of configurations* and similar data. We proved the semantic equivalence of these representations by comparing the corresponding algorithms line-by-line. In the final step, our tool implements the platform specific model in the ANSI-C language.

3.3 Automatic Test Generation for Statechart Implementations

To assess the quality of the test suites standards usually prescribe to meet certain *coverage criteria*, e.g., all statements and decisions must be at least once taken. Our *test generator tool* supports the model-based construction of a test suite satisfying *model based coverage criteria* (i.e., all states and transitions coverage) [5].

The components of our tool are depicted on Fig. 2. From the statechart model and a selected coverage criterion *abstract test cases* are generated that use the events described in the model. These abstract test cases are transformed to the format of the selected test execution engine. These *concrete tests* are then executed, and finally their code-based coverage is measured.

Our tool utilizes an external model checker to calculate the test cases: (i) the statechart is transformed into the input format of a model checker, (ii) each test requirement defined by the coverage criterion is formulated as a temporal logic expression, (iii) for each expression the negation of the formula is verified by the model checker.

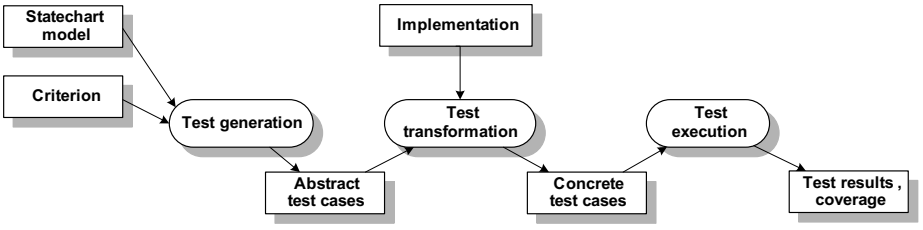


Fig. 2. Components of the Testing Environment

If there is an execution path in the model that does not satisfy the negated formula then it is presented by the model checker as a counter-example. This path becomes a test sequence belonging to the original test requirement. The input and output events are extracted from this path and saved as an abstract test case. The test transformation uses *test skeletons* that describe the test execution engine's format (currently JUnit or Rational Robot) and *templates* (e.g., event dispatching and action verifying code).

3.4 Runtime Error Detection in Statechart Implementations

In case of safety critical systems random faults are typically addressed in run-time by various fault confinement or fault tolerance mechanisms based on efficient *error detection*. We present two *runtime error detection techniques* that aim at the detection of not only random operational faults but also statechart model refinement faults and implementation faults.

Model refinement faults are addressed by defining a *temporal logic language* for the specification of key dependability requirements in the context of early draft models and automatically checking that these temporal correctness criteria hold for the execution of the implementation. On the basis of the formal semantics we defined PSC-PLTL, the propositional linear temporal logic for precise statecharts. It includes (i) *Boolean operators*, (ii) *temporal operators* (the *next-time* X and *until* U, together with shorthand notations like the temporal *future* F and *globally* G) and (iii) *atomic predicates* of the language. The actual connection of PSC-PLTL and statecharts appear in the semantics of *atomic predicates*: they refer to the actual state configuration, the transition conglomerates fired, and the activities performed (this information resides in the state and transition labels of the KTS). For the runtime evaluation of PSC-PLTL formulae we elaborated an efficient method [6]. The source code of the runtime evaluation is *generated automatically*.

Implementation faults may originate from the misunderstanding of the model, usual programming bugs, or from the undesired interference of generated and manually inserted code. In our approach these faults are detected by a monitor that observes the runtime behavior of the implementation and compares it to the statechart model of the application. This approach was inspired by the idea of traditional *watchdog processors* [7] that observe the execution of a program and detect if it deviates from the reference control flow specified by the control flow graph of the program. Although traditional watchdog solutions were successfully applied for detecting low-level errors, unfortunately none of them were capable of supporting such high-level reference structures as state refinement and concurrent execution featured by UML statecharts.

In our approach we instrument the application in such a way that when taking a transition, it sends the labels of the source state, labels of the transition and labels of the target state to the watchdog monitor. These labels and the possible valid sequences of them are defined by the KTS, and the monitor is an automaton accepting the language formed by these valid sequences. This idea was implemented in a tool that generates the *source code of the monitor* automatically on the basis of the PSC model.

4 Conclusions

In our paper we presented a coherent set of tools that support the UML statechart based design of event-driven applications that are typical in railway control systems.

The pilot application of our tools was a railway supervisory and control system [8]. Our experiments have shown that (i) for complex models, applications built according to our code generation approach delivered better performance with lower memory consumption than the common QHsm pattern, (ii) test suites for real-life models ($2 \cdot 10^8$ states) can be generated, (iii) the PSC-PLTL checker is able to detect errors caused by model refinement faults, and (iv) the watchdog monitor detects most of the errors caused by implementation faults and a considerable number of errors caused by physical faults (as demonstrated by a software based fault injection campaign). The utilization of the tools and techniques is envisaged in the SAFEDMI (Safe Driver Machine Interface for ERTMS Automatic Train Control) European project.

References

1. Pintér, G., Majzik, I.: Formal Operational Semantics for UML 2.0 Statecharts. Technical Report at Budapest University of Technology and Economics (DMIS/ESRG) (2005)
2. Pap, Z., Majzik, I., Pataricza, A., Szegi, A.: Methods of Checking General Safety Criteria in UML Statechart Specifications. Journal of Reliability Engineering and System Safety 87(1), 89–107 (2005)
3. Samek, M.: Practical Statecharts in C/C++. CMP Books, Lawrence, Kansas, USA (2002)
4. Object Management Group. MDA Guide (2003), http://www.omg.org
5. Micskei, Z., Majzik, I.: Model-based Automatic Test Generation for Event-Driven Embedded Systems using Model Checkers. In: Proc. Int. Conf. on Dependability of Computer Systems (DepCoS'06), pp. 191–198 (2006)
6. Pintér, G., Majzik, I.: Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements. In: Proc. HASE 2005, pp. 111–120 (2005)
7. Mahmood, A., McCluskey, E.J.: Concurrent Error Detection Using Watchdog Processors - A Survey. IEEE Transactions on Computers 37(2), 160–174 (1988)
8. Pataricza, A., Majzik, I., Huszerl, G., Várnai, Gy.: UML-based Design and Formal Analysis of a Safety-Critical Railway Control Software Module. In Proc. Formal Methods for Railway Operation and Control Systems (FORMS-2003), L' Harmattan, Budapest, 2003.

Formal Specification and Analysis of AFDX Redundancy Management Algorithms

Jan Täubrich[1] and Reinhard von Hanxleden[2]

[1] Philips Medical Systems DMC GmbH*
22335, Hamburg, Röntgenstr.. 40, Germany
jan.taeubrich@philips.com
www.medical.philips.com/de/
[2] Christian-Albrechts-Universität zu Kiel, Institut für Informatik**
24098, Kiel, Olshausenstr. 40, Germany
rvh@informatik.uni-kiel.de
www.informatik.uni-kiel.de/rtsys/

Abstract. Reliable communication among avionic applications is a crucial prerequisite for today's all-electronic fly-by-wire aircraft technology. The *AFDX* switched Ethernet has been developed as a scalable, cost-effective network, based upon IEEE 802.3 Ethernet. It uses redundant links to increase the availability. Typical consensus strategies for the redundancy management task are not feasible, as they introduce too heavy delays. In this paper, we formally investigate AFDX redundancy management algorithms, making use of Lamport's Temporal Logic of Actions (TLA). Furthermore, we present our experiences made with TLA$^+$ and the TLA$^+$ model checker TLC.

Keywords: Redundancy Management, AFDX, TLA, Model Checking, Case Study.

1 Introduction

Reliable communication between avionic subsystems is essential, especially as in 1988 with the Airbus A320 the *all-electronic fly-by-wire* technology attained commercial airline service. There are established avionic data communication protocols such as *ARINC 429* [2] and *MIL-STD-1553* [16]. Recently, the desire for increased performance and more cost-effective solutions has prompted the industry to also explore off-the-shelf alternatives such as IEEE 802.3 Ethernet [14]. The Ethernet specification, however, does not guarantee a maximum latency, as the package collisions are resolved through a *back off* strategy that may lead to a possibly unbounded latency. That is why the next-generation avionics data bus shall on the one hand allow usage of as much cost-efficient, IEEE 802.3 compliant hardware as possible and on the other hand shall guarantee a certain bandwidth and *Quality of Service*, which includes specifying maximal transmission latency. These requirements have lead to the ***A**vionics **F**ull **D**uplex*

* J. Täubrich performed this work while at CAU Kiel.
** R. von Hanxleden performed part of this work while EADS Airbus, Hamburg/Toulouse.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 436–450, 2007.

Switched Ethernet (AFDX), based upon IEEE 802.3 Ethernet technology, which is used today in the Airbus A-380.

AFDX transmits network frames over redundant networks (see Figure 1) and these redundant streams of frames get filtered at the receiving end system (Figure 2). As it turns out, the apparently simple problem of merging redundant streams of frames into a single non-redundant stream is not trivial and enforces some trade-offs in terms of availability, performance and resource requirements. A fairly thorough, but still informal investigation of this redundancy management (RM) problem has been performed in 2001 by von Hanxleden and Gambardella (documented in an internal, unpublished report [6]). This investigation appeared fairly thorough, but the apparent complexity of the problem[1] has prompted an interest in another, more formal investigation, using Lamport's Temporal Logic of Actions (TLA) [9] and the corresponding model checker TLC [17] (documented in a diploma thesis [15]).

This paper summarizes the findings of the first report and the subsequent formalization. The main contributions are, on the one hand, a formal definition and investigation of the redundancy management problem for frame-oriented communication protocols, such as AFDX, and, on the other hand, a fairly involved case study on the use of TLA and TLC for a safety-critical real-world problem.

The rest of this paper is organized as follows. The remainder of the introduction covers the basics of AFDX and TLA$^+$ and surveys related work. Section 2 presents the basics of frame ordering. Sections 3 and 4 describe an environment for the RM algorithms and the checked properties. Section 5 presents three alternative RM algorithms. Sections 6 and 7 summarize our experiences made with TLA$^+$, TLC, and the formal investigation of the RM algorithms.

1.1 AFDX

AFDX addresses the shortcomings of Ethernet using concepts of Asynchronous Transfer Mode (ATM) [5]. AFDX is a profiled network, meaning that configuration tables are loaded into switches at start-up. It is organized in a star topology with a maximum of 24 *End Systems* (ES) per switch. Larger systems can be realized through cascading.

Standard Ethernet suffers the possibility of an infinite chain of frame collisions and hence an unpredictable delay of messages. Therefore every ES is connected to a switch with two twisted pair cables, one pair for sending and the other for receiving frames, which makes AFDX full duplex. Each switch has the capability to buffer multiple packages for each ES in each communication direction. Consequently buffer-overflows and message delays due to congestion at the switch may cause erroneous behaviors. AFDX emulates a deterministic point-to-point network through the use of *Virtual Links* (VLs). Each VL builds a unidirectional path from one ES to one or maybe more other ESs. A certain predefined bandwidth is allocated for each VL, ensuring that the sum of allocations does not exceed the maximum available bandwidth of the whole network. AFDX can be run with either 10 Mbps or 100 Mbps. A minimal bandwidth and a maximum latency for end-to-end transmission is guaranteed.

[1] The original report contained 75 pages of rather terse technical writing, including 37 corollaries and 86 figures, most of which concerned with different communication scenarios. As far as such figures are meaningful, this does suggest a certain complexity of the problem.

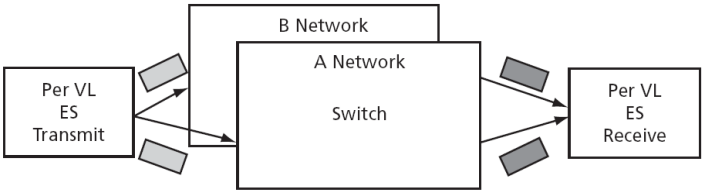


Fig. 1. Concept of redundant networks [6]

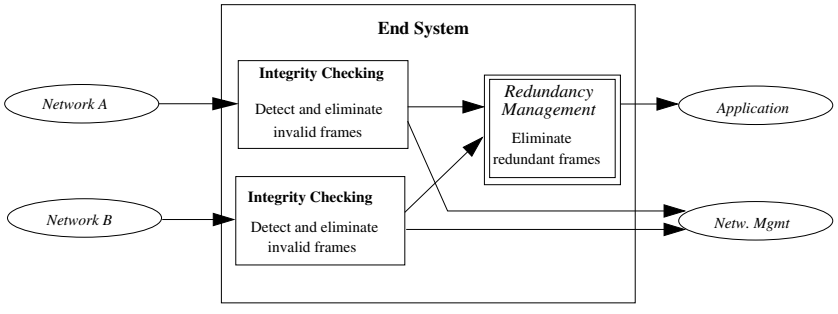


Fig. 2. Receiving End System [6]

In order to improve the reliability AFDX provides a redundant network scheme. Each frame is transmitted in parallel over two redundant networks and afterwards filtered by RM at the receiving ES (Figure 1). This shall reduce the probability of loosing frames and enable further operation even in presence of one faulty network. This redundancy has to be managed somehow, which leads to the task of *redundancy management* (RM). We consider RM to be part of the receiving ES. The task of RM can be formulated quite simply: forward all received frames to the application, but eliminate redundant copies. Hence redundancy shall be transparent to the application.

Each frame has first to pass integrity checking (see Figure 2). A basic form of integrity checking could just check if a frame is *well-formed* (e. g., whether it contains a correct CRC field). A more involved integrity check could validate whether the received frame was expected to be delivered next. Integrity checking, however, is not considered in this paper. It is assumed that integrity checking filters frames in a way that all frames reaching the RM are well-formed. No further assumptions about integrity checking are made.

1.2 Related Work

The technical problem addressed here, namely how to merge redundant streams of communicated frames (packets) into a single logical stream, should occur rather frequently in safety-critical applications. However, we have found that there is comparatively little published systematic work on this. A general approach to simplify modular specifications of dependable distributed systems is given by Sinha and Suri [12]. They propose

to define building blocks to specify and verify larger protocols. These blocks can be consensus, broadcast, redundancy management and many more. However, they do not go into further detail on how to specify these building blocks, as has been investigated in this paper for the redundancy management task.

Tanenbaum [14] gives a basic introduction about computer networks in general. Advanced topics on *ATM* and switched Ethernet are considered in Goralski [5] and Breyer [3].

Temporal logic was introduced by Pnueli [11] to describe system behaviors, a complete overview is given by Manna and Pnueli [10]. Most specifications consist of ordinary mathematics, however, temporal logic is important for describing system properties. The system properties define what a system is supposed to do and the specified automaton describes its real behavior. If the specified behavior implies the conjunction of the properties, the system behaves correctly with regard to the defined properties. In principle, a system's behavior could be defined with a single formula using this formalism. In Pnueli's logic, however, it can be hard to define certain properties of systems. TLA [8] is a variant of Pnueli's originally proposed logic. TLA was developed to permit the simplest, most direct formalization of assertional correctness proofs of concurrent systems. TLA$^+$ [9] is a specification language for concurrent and reactive systems that combines the temporal logic TLA with full first-order logic and Zermelo-Fränkel set theory. A very short introduction to the TLA$^+$ syntax is given in Lamport [7]. One main reason why we selected TLA$^+$ for formulating the redundancy management problem and associated algorithms was the ability to decompose specifications into modules and to specify reusable functions. Sommerfeld provides a case study on TLA$^+$ [13]. TLDA [1] extends TLA to compositionally specify distributed actions. TLC [17] is a model checker for debugging a TLA$^+$ specification by checking invariance and liveness properties of a finite-state model of the specification.

2 Ordering Frames

The redundancy management task requires to eliminate redundant as well as outdated frames. Consider two frames with equal content. What is needed to decide whether one frame is the redundant copy of the other? First we need to know from which *network* a frame was delivered. Two frames delivered by the same network cannot be redundant copies of each other. Furthermore an *order* of frames received from a network must be established. A common approach to identify and order frames is to include a *sequence number* (SN) in each frame. However, a problem here is that SNs are not really unique: since the number of frames sent is not a priori bounded and there are only limited resources for sequence numbers (in our case an 8-bit field), SNs eventually must wrap around. The following considerations address the problem of frame ordering with finite sequence numbers.

The *number of sequence numbers* is $SN\_CNT =_{def} 2^8$. Thus the *maximum sequence number* is $SN\_MAX =_{def} SN\_CNT - 1$. The *mid-point sequence number* is denoted as $SN\_HALF =_{def} SN\_CNT/2$. Consecutive frames have a sequence $SN(f_{i+1}) =_{def} (SN(f_i) + 1) \bmod SN\_CNT$, where SN maps frames to sequence numbers, f_i denotes frame i, and i is some conceptual frame index denoting sending sequence.

In a first step towards comparison operators for the above defined sequence numbers we define subtraction on sequence numbers as follows:

Definition 1. *(Sequence Number Subtraction) The subtraction operator $-_{SN}$ is:*

$$s_1 -_{SN} s_2 =_{def} ((s_1 - s_2 + SN\_HALF) \bmod SN\_CNT) - SN\_HALF.$$

It can be seen that for sequence numbers within a range of SN_HALF and without a wrap around the subtraction of sequence numbers ("$-_{SN}$") is equal to common subtraction on natural numbers modulo SN_CNT and can be used to establish an order on the frames. To order sequence numbers correctly even in the presence of wrap arounds, with the known restrictions on the range of the sequence numbers, the SN_HALF gets involved into the definition. Definition 1 can be used to define the following comparison operators.

Definition 2. *(Comparison Operators)*

$$\begin{aligned} s_1 <_{SN} s_2 &\Leftrightarrow_{def} (s_1 -_{SN} s_2) < 0, \\ s_1 =_{SN} s_2 &\Leftrightarrow_{def} (s_1 -_{SN} s_2) = 0, \\ s_1 >_{SN} s_2 &\Leftrightarrow_{def} (s_1 -_{SN} s_2) > 0. \end{aligned}$$

To prove the correctness of the operators is beyond the scope of this paper, but the corollary below should enable a straight forward proof. The *unwrapped sequence number* $USN(f)$ is needed to reason about sequence number operations. This number is a theoretical number for each frame f, which does not wrap around and thus is unbounded. In the following, *reset* refers to setting the sequence number count to zero; this occurs when an ES gets rebooted.

Corollary 1. *Let frames f_1 and f_2 be generated without intermediate sequence number reset, and let s_1 and s_2 be their respective sequence numbers. If $|USN(f_1) - USN(f_2)| < SN\_HALF$, then it is $s_1 -_{SN} s_2 = USN(f_1) - USN(f_2)$*

Hence a correct ordering of frames using sequence numbers can be established if the unwrapped sequence numbers differ at most by SN_HALF.

3 The Environment

In the following we describe and formally specify an appropriate environment to each of the tested redundancy management algorithms. Such an environment should feed the RM with a well-formed stream of frames and provide information to reason about the correctness of the redundancy management's decisions. The first step is to define the set of actions that the environment can perform. An appropriate environment can either send a frame, loose a frame, deliver a frame to the RM, reset the sequence number count, disable one network due to failures or it just can do nothing for a certain time.

The interaction specification of these allowed actions, hence the specified behavior of the environment without the RM part, is defined as in the TLA$^+$ fragment shown in Figure 3. (Note: for space considerations, we here refrain from a detailed explanation

Step of the environment
$EnvNext \triangleq \exists\, id \in networks : sendFrame \lor die(id) \lor reset$

Step of the redundancy management system
$SysNext \triangleq \exists \langle id,\ pos \rangle \in deliverable :$
$\lor\ extAcceptFrame(id,\ env.frames[id][pos][SN],\ pos)$
$\lor\ extRejectFrame(id,\ env.frames[id][pos][SN],\ pos)$
$\lor\ extWait$

Step of whole model
$Next \triangleq SysNext \lor EnvNext$

Fig. 3. Step definition of environment without RM

of the TLA$^+$ syntax, and would like to refer the reader instead to Lamport's excellent 7-page summary of TLA$^+$ [7], which is also available on-line.) The definition of *EnvNext* expresses that a step of the environment is either a *send* step, a step where a single network *dies* (gets disabled), or a step that *resets* the sequence numbers. A frame is represented as a pair of the transmitting network id and the position in the network queue. *SysNext*, the system's next step, states that a frame from the set of deliverable frames may be either accepted or rejected, or that the environment can just do nothing. All frames ahead of the currently delivered frame on the same network are considered lost. So the loss of frames is implicitly defined, which reduces the state space significantly (and hence speeds up model checking). Finally, the definition of *Next* states that a step of the full model is either a step of the environment or a step of the system, which can be directly mapped to a step of the redundancy management.

Providing an Oracle

A crucial point of the environments specification is to enable reasoning about the correctness of the redundancy management algorithms, as the ones presented in Section 5. To recognize faulty behaviors, a system must be introduced that, independently of the actual program state, marks pending frames correctly, corresponding to their status as either *normal*, *redundant* or *old*. A frame is considered to be *normal* **iff** no frame sent later to any network has yet been received by the RM and its *twin frame* (i. e., its redundant copy) has not yet been received. A frame is considered to be *redundant* **iff** its twin frame has already been delivered to the RM. The remaining frames—which are not redundant and where a later sent frame has already been delivered to the RM—are considered *old*.

The environment sends its frames to both networks in parallel and only if both networks still have capacity to buffer another frame. Thus for a given frame f from network N_1 it is

1. decidable whether f's twin frame is still transient on N_2, and
2. possible to find the position of f's twin frame in the sequence where it is located.

Figure 4 shows how an algorithm to mark all frames correctly with a minimum effort is realized in TLA$^+$. Figure 5 shows an example behavior. The first two frames of each

$$
\begin{array}{l}
tag[seq1 \in Seq([sn : (0\,..\,SN\_MAX),\ tag : \{\text{“n”},\ \text{“r”},\ \text{“o”}\}]),\\
\quad seq2 \in Seq([sn : (0\,..\,SN\_MAX),\ tag : \{\text{“n”},\ \text{“r”},\ \text{“o”}\}]),\\
\quad val \in (0\,..\,SN\_CNT),\ id \in networks] \triangleq\\
\quad\ \text{IF } Len(seq1) > Len(seq2) \text{ THEN}\\
\qquad \text{IF } Head(seq1)[TAG] = \text{“r”} \text{ THEN } \langle Head(seq1)\rangle \circ tag[Tail(seq1),\ seq2,\ val,\ id]\\
\qquad\ \text{ELSE } \langle [sn \mapsto Head(seq1)[SN],\ tag \mapsto \text{“o”}]\rangle \circ tag[Tail(seq1),\ seq2,\ val,\ id]\\
\quad\ \text{ELSE } \langle [sn \mapsto Head(seq1)[SN],\ tag \mapsto \text{“r”}]\rangle \circ Tail(seq1)
\end{array}
$$

Fig. 4. Marking algorithm in TLA$^+$

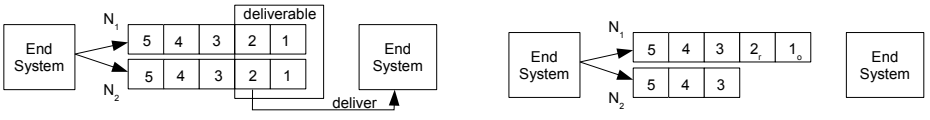


Fig. 5. The Marking Algorithm, with example behavior (left) and result (right)

network are considered to be deliverable to the receiving end system. When the second frame of N_2 gets delivered, the marking algorithm takes N_1 and N_2 without the first frame as input. As long as N_1 has more elements than the modified sequence N_2, the first element of N_1 is marked as old, if not already marked as redundant, and calls the function recursively with the tail of sequence N_1. If both sequences have equal length, the algorithm marks the first frame in N_1 as redundant and terminates, as can be seen in Figure 5. A full discussion of this algorithm and a correctness proof, based on Floyd's inductive assertion method for transition diagrams [4], can be found in [15].

4 The Properties

Traditionally an algorithm is examined to satisfy *safety* and *liveness* properties, stating that it behaves correctly and does not block. This is not appropriate in our situation as, before our formal investigation, we did not expect any of the redundancy management algorithms to be completely safe, which means neither forwarding redundant nor outdated frames to the application layer. The proposed algorithms in von Hanxleden [6] were not designed to satisfiy these safety requirements at all circumstances. That is why we refined the original properties relative to the behavior of the environment, e.g. one property allows the behavior to reset the sequence numbers, while a relaxed property would prevent such resets. This should help to distinguish the different algorithms and to decide about their quality. Besides safety and liveness a third class of requirements addresses properties regarding the *quality* of an algorithm. This includes the properties concerning the per frame loss as well as special scenarios with only one connected network. A fourth class of requirements, *availability*, addresses the behavior of the algorithms in case of network failures. As for the safety requirements, several refinement steps were defined to distinguish the analyzed algorithms. To give a complete description of all properties is beyond the scope of this paper, and we will restrict ourselves to give representatives of each of the four classes of properties. However, the numbering of the properties corresponds to the original thesis [15].

4.1 Safety

What does safety mean for RM? First of all, the RM shall not submit any redundant frames to the application layer. Secondly the RM shall preserve the order of frames and hence shall not submit *old* frames to the application layer. Each of these tasks can be weakened accordingly to the benignity of the environment.

***Redundancy 2*: If the environment does not reset anymore, the RM stabilizes and works properly from that time on.**

$$\begin{aligned} \mathit{Redundancy2} \triangleq\ & \Diamond\Box\neg[\mathit{reset}]_v \\ & \Rightarrow \Diamond\Box(\forall \langle \mathit{id},\ \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \text{ENABLED } \langle \mathit{extAcceptFrame}(\mathit{id},\ \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\ \mathit{pos})\rangle_v \\ & \quad \Rightarrow \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{TAG}] \neq \text{“r”}) \end{aligned}$$

The premise of the TLA$^+$-formula expresses that from some state σ_i on, all consecutive states $\sigma_n \rightarrow \sigma_m$ are neither a *reset* nor a stuttering step. If this holds the redundancy management algorithm shall, after a finite time, only accept frames which are not marked as redundant.

***Order 1*: No old frame shall ever be submitted to the application layer.**

$$\mathit{Ordering1} \triangleq \forall\, \mathit{frame} \in \mathit{out} : \mathit{frame}[\mathit{TAG}] \neq \text{“o”}$$

Obviously this is a very rigorous claim, which we expect to be failed by most of the proposed algorithms. Nevertheless this claim can be relaxed in the same way we relaxed the claim to never accept redundant frames to what is defined above in *Redundancy 2*.

4.2 Liveness

Of course the redundancy management algorithms shall not deadlock as long as it receives frames from its environment. More specifically:

***Liveness*: Each frame that is delivered to the RM will be either accepted or rejected.**

$$\begin{aligned} \mathit{Liveness} \triangleq\ & \forall \langle \mathit{id},\ \mathit{pos}\rangle \in \mathit{deliverable} : \\ & \vee \text{ENABLED } \langle \mathit{extAcceptFrame}(\mathit{id},\ \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\ \mathit{pos})\rangle_v \\ & \vee \text{ENABLED } \langle \mathit{extRejectFrame}(\mathit{id},\ \mathit{env.frames}[\mathit{id}][\mathit{pos}][\mathit{SN}],\ \mathit{pos})\rangle_v \end{aligned}$$

It is expected that all algorithms will satisfy this property as they are all of type: IF *condition* = TRUE THEN *accept* ELSE *reject*, which implies that there exists a unique decision for each frame. That is why we do not mind that our formula is stronger than needed as this formula reasons about all frames and not only the set of received frames. It is enough to know that the *Liveness* formula implies absence of deadlocks.

4.3 Quality

One of the original requirements [6] states that *the redundancy management shall maintain the availability of a single network*. In other words, it shall not increase the number

of frames lost that would be obtained with one of two networks normally running and alone. This is a difficult, but important demand. Assume a fast but unreliable, hence lossy network A and a slower, completely reliable network B. It follows that an algorithm would need to use buffering to solve this problem. Buffering is considered harmful since it produces possibly large delays. So a gradation was introduced to obtain a more realistic estimation for the performance of the algorithms.

***Quality 0*: If only one network is connected to the RM, all received frames are forwarded.**

$$\begin{aligned} Quality0 \triangleq\ & \Box(\forall\, id1,\, id2 \in networks : isAlive[id1] \wedge \neg isAlive[id2]) \\ & \Rightarrow \Box(\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad \neg\text{ENABLED}\ \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

Hence, if the RM receives only frames from one network, the whole ES shall behave as with only one network running alone.

***Quality 1*: If both networks are alive and at least one member of a *Twin Frame* reaches the RM, one member gets submitted.**

$$\begin{aligned} Quality1 \triangleq\ & \forall\, id \in networks,\, pos \in (1\,..\,MCFL) : \\ & \wedge\, isAlive[id] \\ & \wedge\, isAlive[TNid[id]] \\ & \wedge \langle id,\, pos\rangle \in deliverable \\ & \wedge\ \text{ENABLED}\ \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v \\ & \Rightarrow \exists\, frame \in out : frame[SN] = env.frames[id][pos][SN] \end{aligned}$$

Quality 1 is equivalent to the original requirement, but it is easier to formalize that if a frame gets rejected, its twin frame has already successfully passed the redundancy management algorithm.

4.4 Availability

The goal of redundancy is to raise the availability of a system by duplicating parts of a system. In our case communication is done over multiple networks, since one single network is not reliable enough. The *Quality* requirements are concerned with the availability of the system in case of both networks operating. Now the case is considered that one network dies. This is the most important part of the properties. Redundancy is used to remain operating in presence of partial failures. These properties tell us how good an algorithm serves this task and finally enables a final decision whether this algorithm is a feasible choice.

***Avail 1*: If one network fails, all consecutive frames of the other, remaining network are accepted.**

$$\begin{aligned} Avail1 \triangleq\ & \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2]) \\ & \Rightarrow (\forall \langle id,\, pos\rangle \in deliverable : \\ & \quad \neg\text{ENABLED}\ \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v) \end{aligned}$$

An algorithm that satisfies this property would be a preferred choice (unless it fails all other requirements). Although this formula looks similar to *Quality 0* they describe

different behaviors of the environment. While *Quality 0* specifies that from the beginning only one network is operating, *Avail 1* specifies that one network fails during execution. Obviously an algorithm that handles absence of one network correctly at any time, it especially handles the situation where only one network operates from the beginning correctly. Hence *Avail 1* $\Rightarrow$ *Quality 0* holds. To satisfy this formula, it would help if the RM could detect whether a network is down; this task, however, was explicitly moved to the *Network Management*.

***Avail 2*: If one network fails and no *reset* occurs from that time on, all consecutive frames of the remaining network are accepted.**

$$
\begin{aligned}
Avail2 \triangleq\ & \Box(\wedge \Box(status \neq \text{“reject”}) \\
& \wedge \exists\, id1,\, id2 \in networks : (isAlive[id1] \wedge \neg isAlive[id2]) \\
& \Rightarrow (\forall \langle id,\, pos\rangle \in deliverable : \\
& \quad \neg \text{ENABLED}\ \langle extRejectFrame(id,\, env.frames[id][pos][SN],\, pos)\rangle_v))
\end{aligned}
$$

5 Three Redundancy Management Algorithms

Of the thirteen RM algorithms considered in the original report [6], we now present a selection of three algorithms. As with the environment, the first step to model the algorithms behaviors is to define appropriate actions. This specification of actions is trivial, because all the RM can do is to receive a frame and decide whether it should submit or discard it.

5.1 RMA1

Recall that the *Sequence Number* (SN) of frame f is $SN(f) \in \{0 \ldots SN\_MAX\}$. The *Received Sequence Number* of frame f is denoted as $RSN(f)$ and it may be $SN(f) \neq RSN(f)$. However we assume those frames with $SN(f) \neq RSN(f)$ are not well-formed and thus discarded by the integrity checking. The *Previous Twin Network Frame* $PTN(f)$ for a frame f received on network N_1 denotes the last frame received on the other network N_2.

Definition 3. *(Sequence Number Skew) The Sequence Number Skew (SNS) of frame f is*

$$SNS(f) =_{def} RSN(f) -_{SN} RSN(PTN(f)).$$

This leads to the first RM algorithm considered here: **RMA1** specifies to *“accept a frame if and only if its sequence number skew is positive”*. The corresponding TLA$^+$ specification is given in Figure 6.

The formal analysis of RMA1 with TLC revealed that this algorithm works correctly if both networks are operational. However, if one network fails, RMA1 at some point starts rejecting frames from the remaining network due to the finite sequence numbering, as illustrated in Figure 7. This violates the property *Avail 2*. Moreover the algorithm will periodically discard non-redundant frames.

Accept frame IF frames are available AND $(SNS(f) > 0)$

$$
\begin{aligned}
&acceptFrame(id,\ sn) \triangleq \\
&\quad \wedge\ snSkew[id,\ sn] > 0 \\
&\quad \wedge\ rm' = [rm \text{ EXCEPT } !.rsn = sn,\ !.ptn[id] = sn]
\end{aligned}
$$

Reject frame IF frames are available AND $SNS(f) <= 0$

$$
\begin{aligned}
&rejectFrame(id,\ sn) \triangleq \\
&\quad \wedge\ snSkew[id,\ sn] \leq 0 \\
&\quad \wedge\ rm' = [rm \text{ EXCEPT } !.rsn = sn,\ !.ptn[id] = sn]
\end{aligned}
$$

Fig. 6. Specification of RMA1

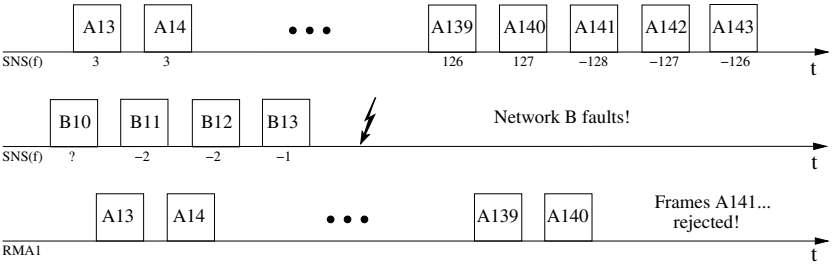


Fig. 7. Silent Network Scenario—Using RMA1. The upper time lines indicate frames received along the redundant network, along with their corresponding Sequence Number Skews. The lower time line indicates frames that RMA1 lets through to the application.

5.2 RMA3

As an evolution from RMA1, the next RM algorithm considered here compares not only the sequence numbers between frames of different networks, but also the difference of successive frames of the same network. Let $PAF(f)$ be the previously (from the RMA) accepted frame, prior to the reception of a frame f.

Definition 4. *(Previously Accepted Sequence Number) Let f be a frame with* $SN(f) \in \{0 \ldots SN\_MAX\}$. *Then the Previously Accepted Sequence Number (PASN) of f is given by*

$$PASN(f) =_{def} RSN(PAF(f)).$$

Definition 5. *(Sequence Number Offset) The Sequence Number Offset of a frame f is given by*

$$SNO(f) =_{def} RSN(f) -_{SN} PASN(f)$$

RMA3 specifies to *"accept if and only if the maximum of the sequence number skew and the sequence number offset is positive"*. The corresponding TLA$^+$ specifications to accept or reject the currently received frame are shown in Figure 8. Note how *paf* keeps track of accepted SNs. Model checking RMA3 proved that this algorithm will handle network failures better than RMA1. In case of a faulty network and no reset of the sequence number this algorithm will accept all subsequent frames. But as it turns out, RMA3 still needs two operating networks to cope with sequence number resets.

```
Accept frame IF frames are available AND (SNS(f) > 0 OR SNO(f) > 0)
acceptFrame(id, sn) ≜
  ∧ ∨ snSkew[id, sn] > 0 ∨ snOffset[sn] > 0
  ∧ rm' = [rm EXCEPT !.rsn = sn, !.paf = sn, !.ptn[id] = sn]

Reject frame IF frames are available AND SNS(f) < = 0 AND SNO(f) < = 0
rejectFrame(id, sn) ≜
  ∧ snSkew[id, sn] ≤ 0 ∧ snOffset[sn] ≤ 0
  ∧ rm' = [rm EXCEPT !.rsn = sn, !.ptn[id] = sn]
```

Fig. 8. Specification of RMA3

```
Exceed time bound:
wait ≜ ∧ rm.time = TRUE ∧ rm' = [rm EXCEPT !.time = FALSE]

Accept frame IF frames are available AND SNS(f) > 0
acceptFrame(id, sn) ≜
  ∧ ∨ rm.pan = "all" ∨ id = rm.pan ∨ rm.time = FALSE
  ∧ rm' = [rm EXCEPT !.pan = id, !.time = TRUE]

Reject frame IF frames are available AND SNS(f) < = 0
rejectFrame(id, sn) ≜
  ∧ rm.pan ≠ "all" ∧ id ≠ rm.pan ∧ rm.time = TRUE
  ∧ rm' = [rm EXCEPT !.time = TRUE]
```

Fig. 9. Specification of RMA13

5.3 RMA13

The last algorithm considered here, RMA13, circumvents this limitation by introducing the concept of time. TLA$^+$ allows the modeling of continuous real-time aspects. Unfortunately, a first, very detailed timing model was not checkable with TLC, because of state explosion. However, since the only real-time aspect in the redundancy management algorithms is a time-out, we could model this as a another ordinary action in TLA$^+$ that can occur under specific circumstances. **RMA13** is specified to *"accept if and only if the frame has the same network identifier as the last accepted frame or after time out"*. The corresponding TLA$^+$ specification is given in Figure 9. Note that in addition to the already known actions to accept or reject a frame, an action is introduced to model that the maximum time between two successive accepted frames is exceeded. RMA13 actually deviates from the *first valid wins* strategy, meaning that the first valid twin frame should always be passed to the application. Model checking revealed that, although RMA13 may cause a higher per frame loss than other algorithms, it behaves best in critical situations like network failures and sequence number resets. The most important advantage of RMA13 is that it satsfies all safety properties, which means that it never submits redundant or outdated frames to the application layer.

A complete and thorough analysis of the remaining algorithms would exceed the bounds of this paper. The interested reader can find the complete analysis in the thesis [15].

6 Assessment of the RMA Algorithms and Their Formal Modeling

In summary, the formal analysis results indicate that one of the simplest proposed algorithms, RMA13, is the best choice overall. Although it does not increase the macroscopic availability, it satisfies all safety related properties, as it never submits redundant or outdated frames to the application, which is considered more important for the redundancy management task. Furthermore, the formal investigation of the RM task revealed that the decision to have a relatively small sequence number range (only 8 bits, as opposed to 28 bits, as had been considered originally) is not only economical, but prevents some catastrophic behaviors, too.

In our experience, TLA$^+$ is very suitable to specify the redundancy management algorithms. Though the compact notations of TLA$^+$ might be a little bit confusing at a first glance, they are very practical and maintain a good readability. Moreover the concept of untyped variables turned out to be not as error-prone as expected and is indeed very flexible.

The experience with the model checker TLC was also positive overall, but we did encounter some complications and at times strange or (we think) even faulty behaviors. TLC allows to check a wide range of expressible TLA$^+$ properties, however, temporal formulas that shall be checked with TLC and which contain actions must be of the form $\Box\Diamond A$ ("always eventually A") or $\Diamond\Box A$ ("eventually always A"), where A denotes the action. A work-around for this is to introduce another variable that records the action actually taken and to use this variable instead of actions in the formulas.

The possibility of constraining infinite models with specifying constraints on the defined variables is a major quality of TLC. Nevertheless the user must take care that the specified and checked invariants not only hold on the constrained state space. TLC checks for each state if all invariants hold and subsequently if all constraints are satisfied. Thus a state, which is considered unreachable may cause a violation of an invariant. This increases the complexity of optimizing the TLA$^+$ specifications for model checking with TLC.

A complete and more detailed description of the observed limitations can be found in the original thesis [15].

7 Summary and Conclusions

Of course, model checking a large set of algorithms and properties has its limitations. In our case, there is no way to give reliable conclusions of how many frames one algorithm accepts and rejects. However, the formal specification of the RM algorithms did detect gaps in the original specification. The specification of one algorithm turned out to be wrong, such that this algorithm failed all defined properties.

Writing formal specifications forces the engineer to clearly express what an algorithm must and what it must not do. The formalization effort presented here therefore focused on stating a precise set of requirements, and checking which algorithms fulfill which requirements. In contrast, the original, informal investigation followed an evolutionary, scenario-based approach; it started with a simple RM algorithm, detected scenarios where this RMA failed, and subsequently extended/modified the RMA. We

have noticed that some RMAs fail the same properties, however, behave differently for certain scenarios. This suggests that the requirements in the formalization, which were already significantly refined compared to the original report, still could be refined further.

Acknowledgments

Eddie Gambardella has helped to formulate the redundancy management problem and to develop the original algorithms. We would also like to thank Leslie Lamport, for valuable discussions regarding TLA$^+$ and TLC, and the anonymous reviewers, for providing valuable feedback on this manuscript. Finally, we thank EADS/Airbus for kind permission to publish this work.

References

1. Alexander, A.: Komposition Temporallogischer Spezifikationen - Spezifikation und Verifikation von Systemen mit Temporal Logic of Distributed Actions. PhD thesis, Humboldt-Universität zu Berlin (2005)
2. ARINC Incorporated. Homepage, http://www.arinc.com
3. Breyer, R., Riley, S.: Switched, Fast and Gigabit Ethernet, 3rd edn. MacMillan Technical Publishing (1999)
4. Floyd, R.W.: Assigning meaning to programs. In: Proceedings AMS Symposium Applied Mathematics, pp. 19–31 (1967)
5. Goralski, W.: Introduction to ATM Networking. McGraw-Hill, New York (1995)
6. Hanxleden, R.V., Gambardella, E.: AFDX Redundancy Management (February 2001)
7. Lamport, L.: A summary of TLA+. http://research.microsoft.com/users/lamport/tla/tla.html
8. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16 3, 872–923 (1994)
9. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2002), http://research.microsoft.com/users/lamport/tla/book.html
10. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
11. Pnueli, A.: The temporal logic of programs. In: In Proceedings of the 18th Symposium on Foundations of Programming Semantics, pp. 46–57 (1977)
12. Sinha, P., Suri, N.: On simplifying modular specification and verification of distributed protocols. In: HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering, pp. 173–181. IEEE Computer Society Press, Washington (2001)
13. Sommerfeld, L.: Spezifikation eines 20 l-Perfusionsbioreaktor in TLA+. Diploma thesis, Universität Bielefeld (1997)
14. Tanenbaum, A.S.: Computernetzwerke, 4th edn. Prenctice Hall (2003)
15. Täubrich, J.: Formal specification and analysis of a redundancy management system with TLA+. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (March 2006), http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jat-dt.pdf

16. US Department of Defense. Aircraft internal time division command/response multiplex data bus (mil-std-1553b). US Department of Defense, 1978-09-21, http://dodssp.daps.dla.mil.
17. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999), http://link.springer.de/link/service/series/0558/bibs/1703/17030054.htm

Modeling and Automatic Failure Analysis of Safety-Critical Systems Using Extended Safecharts

Yean-Ru Chen[1], Pao-Ann Hsiung[2], and Sao-Jie Chen[1]

[1] Graduate Institute of Electronics Engineering,
National Taiwan University, Taipei, Taiwan−106, ROC
d95943037@ntu.edu.tw, csj@cc.ee.ntu.edu.tw
[2] Department of Computer Science and Information Engineering,
National Chung Cheng University, Chiayi, Taiwan−621, ROC
hpa@computer.org

Abstract. With the rapid progress in science and technology, we find ubiquitous use of *safety-critical systems* in avionics, consumer electronics, and medical instruments. In such systems, unintentional design faults might result in injury or even death to human beings. To avoid such mishaps, we need to verify safety-critical systems thoroughly, where formal verification techniques such as model checking play a very promising role. Currently, there is practically no automatic technique in formal verification used to formally model system faults and repairs. This work contributes in proposing an extension to the *Safecharts* model, with which faults and repairs can be easily modeled. Moreover, these Safecharts can be directly transformed into semantically equivalent *Extended Timed Automata* models for model checking. That is, after these models were integrated into a model checker, such as our previously proposed *State Graph Manipulators* (SGM) model checker, we can verify safety-critical systems. An application example is run to show the feasibility and benefits of the proposed model-driven verification method for safety-critical systems. As observed, the checking results, such as witnesses of property specifications representing hazards, provide more concrete and useful failure analysis information than the conventional Fault Tree Analysis (FTA).

Keywords: Safety-critical systems, Safecharts, FTO-failure, SO-failure, NC-failure, Effective repair actions, Ineffective repair actions.

1 Introduction

Safety-critical systems are systems whose failure causes damages to its environment. Traditional hazard analysis techniques, such as fault tree analysis (FTA), failure modes and effects analysis (FMEA), failure modes, effects, and criticality analysis (FMECA) have been successfully applied to several different real-world safety-critical systems [12]. Recently, formal verification techniques such as model checking [4] has become a promising verification method due to its automatic analysis capabilities. We propose an extended Safecharts model [5] to model and analyze failures and repairs in safety-critical systems and integrate it into a formal verification tool. Our contributions are

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 451–464, 2007.

three-folds. First, according to the classification of failure modes, we propose an intuitive representation of the faulty states in a safety-critical system using Safecharts model. Second, we transform the Safecharts model into a semantically equivalent *Extended Timed Automata* (ETA) model that can be directly input to model checkers. Finally, a compositional model checker, namely the *State Graph Manipulators* (SGM) [15], is used to not only verify a safety-critical system formally, but also provide automatic failure analysis results to help users rectify that system.

The remaining of this article is organized as follows. Section 2 describes the current state-of-the-art in the verification of safety-critical systems. Section 3 describes our work flow and how failure analysis is performed in model checking. The extension of Safecharts to cover faults and repairs is given in Section 4. Section 5 shows the details of the transformation from Safecharts to ETA. The implementation of our proposed method in the SGM model checker is described in Section 6. An application example is given in Section 7 along with several analysis results. The article is concluded and future research directions are given in Section 8.

2 Related Work

The conventional hazard analysis techniques, such as FTA and FMEA, have been successfully applied to several different real-world safety-critical systems. Nevertheless, a major limitation of hazard analysis is that phenomena unknown to the analysts are not covered in the analysis and thus hazards related to these phenomena are not foreseen. Safety-critical systems are getting more and more complex and thus there is a trend to use methods [3] that are more automatic and exhaustive than hazard analysis, for example model checking.

The verification of safety-critical systems using formal techniques is not new [12]. However, as noted by Leveson [12], this approach is inadequate because in the system models we are assuming that all the components do not fail and the system is proved to be safe under this assumption. However, the assumption is not always valid, so transforming each hazard into a formal property for verification as in [9] is not sufficient. As described in the following, our work on using Safecharts to verify safety-critical systems contributes in several ways to the state-of-the-art in formal verification.

1. The Unified Modeling Language (UML) is an industry de facto standard for model-driven architecture design. Safecharts, being an extension of the UML Statecharts, blends naturally with the semantics of other UML diagrams for the design of safety-critical systems. The work described in this article automatically transforms Safecharts into the timed automata model which can be accepted by conventional model checkers.
2. The extended Safecharts model allows and requires explicit modeling of component failures and repairs within a newly proposed failure layer in the model. This is very helpful not only for the safety design engineers but also for the safety verification engineers. Through its unique features of risk states, transition priorities, and component failure and repair modeling artifacts, Safecharts can be successfully used for verifying safety-critical systems.

3. The model checking paradigm helps generate witnesses to the satisfaction of properties that provide more accurate and informative safety analysis results than conventional methods such as FTA.

3 Automatic Failure Analysis

The failure analysis performed in a model checker is as proposed and illustrated in Figure 1. A safety-critical system is first analyzed for all possible failures and corresponding repairs. Safecharts are then used to model the functional, safety, and failure characteristics of the system, according to the description in Section 4. As described in Section 5, these Safechart models are then automatically transformed into equivalent ETA models that are input to the SGM model checker as system models. Each hazard is then represented as a temporal logic property. On model checking the ETA against the properties, we obtain witnesses to the hazard properties that represent how the system might face a hazard. The witnesses are system traces that contain more information than traditional minimum cut results from FTA.

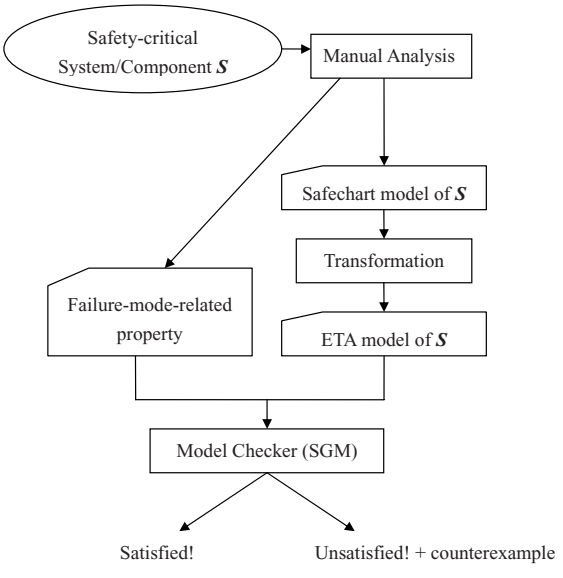


Fig. 1. Work flow of automatic failure analysis in model checking

4 Extending Safecharts

The main focus of the original Safecharts [5], as defined in Definition 1, was on proposing functional and safety layers. Failures are due to faults, but not all faults result in failures. In this work, we focus on failures and extend the original Safecharts with a more

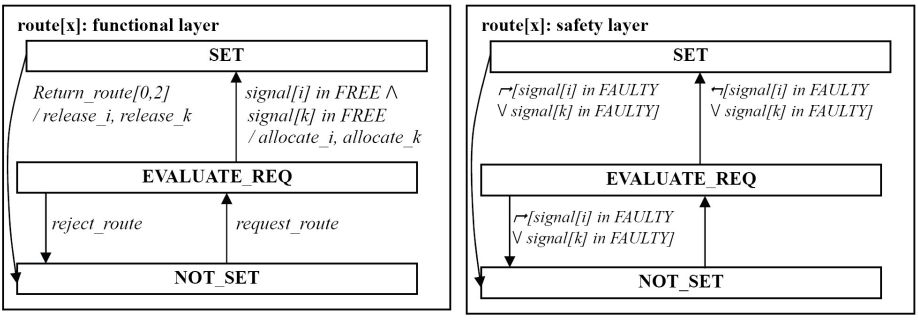


Fig. 2. Safechart for route[x] with functional and safety layers

comprehensive modeling capability for different types of failures and repairs, in a new and explicit *failure layer*. In this extension, we use a generic classification of failures and repairs to help safety-critical system designers categorize the possible failures and repair actions in the system to be modeled and verified.

4.1 Safecharts

Figure 2 shows the *functional* and *safety* layers of a Safecharts model route[x] for setting a route of a railway system. The functional layer specifies the normal functions of requesting and setting or rejecting a route. The safety layer enforces the safety restrictions for setting or unsetting a route. Notations $\nwarrow$ and $\nearrow$ in the safety layer will be defined in Definition 1, which, respectively, restricts setting a route or enforces the release of a route when any of the signals in that route is faulty. Recognizing the possibility of gaps and inaccuracies in safety analysis, Safecharts do not permit transitions between states with unknown relative risk levels [14].

Definition 1. *Safecharts*
Given two comparable states s_1 and s_2, a risk ordering relation $\preccurlyeq$ specifies their relative risk levels, that is $s_1 \preccurlyeq s_2$ specifies s_1 is safer then s_2. Transition labels in Safecharts have an extended form: $e[fcond]/a[l,u)\Psi[G]$ where e, $fcond$, and a are the conventional Statechart event, transition guard, and action, respectively, and $/$ separates the transition guard from the transition action. The time interval $[l,u)$ is a real-time constraint on a transition t and imposes the condition that t does not execute until at least l time units have elapsed since it most recently became enabled and must execute strictly within u time units. The expression $\Psi[G]$ is a safety enforcement on the transition execution and is determined by the safety clause G. The safety clause G is a predicate, which specifies the conditions under which a given transition t must, or must not, execute. Ψ is a binary valued constant, signifying one of the following enforcement values:

- *Prohibition enforcement value*, denoted by $\nwarrow$. Given a transition label of the form $\nwarrow[G]$, it signifies that the transition is forbidden to execute as long as G holds.
- *Mandatory enforcement value*, denoted by $\nearrow$. Given a transition label of the form $[l,u) \nearrow [G]$, it indicates that whenever G holds the transition is forced to execute within the time interval $[l,u)$, even in the absence of a triggering event.

4.2 Failure Modes

The failure modes of safety-critical systems can be classified into three types as defined in the following [2].

- *Fail-To-Operate* (FTO-failure): A system does not respond correctly to an abnormal operating condition. This is a critical failure.
- *Spurious Operation* (SO-failure): A system initiates an automatic unexpected action without the presence of an abnormal operating condition. This is also a critical failure.
- *Non-critical* (NC-failure): In this case, maintenance of a system is required even though the main service of the system has been preserved, that is it is neither an FTO-failure nor an SO-failure in the safety-critical system. With this type of failure, a system can still work, but its performance may be poorer than expected. We assume that when a system has an NC-failure, it must be repaired before other critical failures can occur.

The representation of failure modes in Safecharts is given in Section 4.4.

Example: Take an automatic electric lamp as an example, which is supposed to be turned on automatically within three seconds when the room is too dark, however if it fails, then an FTO-failure occurs. In addition, if the lamp is turned on unexpectedly when the room is bright, it is said that an SO-failure occurs. However, if the lamp blinks after being turned on, we take this as an NC-failure.

After an FTO- or SO-failure, as shown in Figure 3 and Figure 4, respectively, when more than one intermediate repair actions are needed to return to normal state, then we assume that the component or system enters an NC-failure mode, which is an intermediate state between an FTO- or SO-failure and a normal state.

A component FTO- or SO-failure can lead to a system FTO- or SO-failure unless this is prevented by fault-removal or fault-prevention techniques [2]. In this work, we assume that each component is under the control of its controller and the controllers could be taken as error-free. We do not discuss controller fault tolerance techniques in this work.

4.3 Safety Repair Actions

A system may return to normal operating condition after a failure is either automatically or manually repaired. Repair actions of safety-critical systems can be classified into two types as follows.

- *Effective repair actions*: A failure is completely eliminated after an effective repair action is taken. We denote the set of all effective repair actions as $U = \{\mu\}$.
- *Ineffective repair actions*: A failure is not completely eliminated, but its severity may be reduced after ineffective repair actions are taken. These repair actions could be taken as intermediate safety actions. Let $U' = \{\mu'\}$ denote the set of all ineffective repair actions. Moreover, $U \cap U' = \phi$. □

For example, in an automatic electric lamp system, if the electric lamp is not turned on automatically within three seconds when the room is too dark, an FTO-failure occurs. We should turn it on manually to eliminate this failure. This repair action is effective if the electric lamp becomes bright after we turn it on. Otherwise, it is an ineffective repair action.

4.4 Representation of Failures and Repairs in Extended Safecharts

Using some conventional safety analysis technique, a designer can identify all the possible failures and their effects in a system to be modeled and verified. Further, based on the classifications of failures and of repair actions, as described in Sections 4.2 and 4.3, respectively, a system designer can categorize all the possible failures and corresponding repair actions of the system. We now give the representations of these different failures and repair actions in the extended Safecharts model.

Recall from Definition 1, in Safecharts, the transition label $e[fcond]/a$ appears in the *functional* layer, while $[l,u)\Psi[G]$ may appear in the *safety* layer. In this work, as mentioned before, we propose a new *failure layer*, and three new symbols in Safecharts to represent failure conditions, namely, *inoperable* ($\overline{\Rsh}$), *spurious* ($\overline{\Lsh}$), and *non-critical* ($\leftarrow\!\!\rho$) failures, which are modeled in the failure layer of extended Safecharts.

In this work, we found that there is an interesting coupling between safety conditions and failure conditions. Whenever a mandatory condition ($[l, u) \Rsh [G]$) is violated, an FTO-failure ($[u, \infty)\overline{\Rsh}[G]$) occurs. Whenever a prohibitory condition ($\Lsh [G]$) is violated, an SO-failure ($\overline{\Lsh}[G]$) occurs. As far as the non-critical ($\leftarrow\!\!\rho$) condition is concerned, it can be used in more flexible ways than the other two critical failure conditions. System designers can model an NC-failure for a component or a system wherever it is required.

Figures 3, 4, and 5 show the representation of different failure modes and their corresponding repair actions for a component or system modeled in Safecharts. Each component or system may be modeled with one or more failure modes. The condition and assignment label $e[fcond]/a[l, u)\Psi[G]$ on a transition t, from the original Safecharts (Definition 1), is now extended to cover both safety conditions as well as failure conditions, as classified in the following.

1. *Safety conditions*: According to [5], safety conditions are modeled into the functional and safety layers.
   - $e[fcond]/a$: This is just a normal functional transition without any safety clause.
   - $e[fcond]/a \Lsh [G]$: There is *prohibition* enforcement value on a transition t. It signifies that the transition t is forbidden to execute as long as G holds.
   - $e[fcond]/a[l, u) \Rsh [G]$: There is *mandatory* enforcement value on a transition t. It signifies that the transition is forced to execute within [l,u) whenever G holds.
2. *Failure conditions*: Failure conditions are modeled into the failure layer of Safecharts.
   - $e[fcond]/a[u, \infty)\overline{\Rsh}[G]$: Transition t represents an FTO-failure. It signifies that under the condition G, if a corresponding mandatory transition t' with label

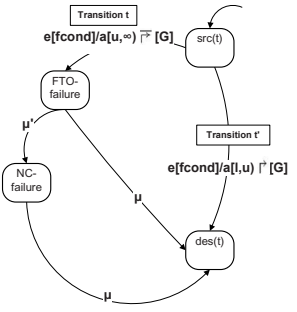


Fig. 3. FTO-failure in Safechart

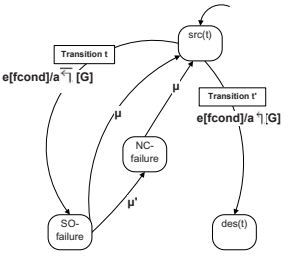


Fig. 4. SO-failure in Safechart

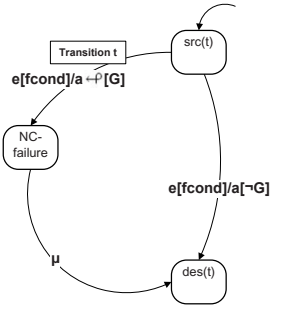


Fig. 5. NC-failure in Safechart

$e[fcond]/a[l,u)$ $[G]$ is not executed within the time interval $[l,u)$, then there is a critical FTO-failure.

- $e[fcond]/a$ $[G]$: Transition t represents an SO-failure. It signifies that under the condition G, if a corresponding prohibitory transition t' with label $e[fcond]/a$ $[G]$ is unexpectedly executed, then there is a critical SO-failure.
- $e[fcond]/a$ $[G]$: Transition t represents an NC-failure. The condition G means that the main service of the safety-critical system is not supported as good as expected. It signifies that repair actions are needed when G holds.

Here, we assume that when a component or system has failed, the failure must be repaired before another failure can occur.

Let us take the automatic electric lamp system as an example again to illustrate the three failure modes represented by the above described transitions. Table 1 shows the conditions and assignments for normal safety operation and for the three possible failure modes of the electric lamp.

In Figure 3, the transition from *src(t)* to *des(t)* has mandatory evaluation. By definition, the transition is supposed to be executed before the deadline u. If it fails, by the failure definitions, we can specify that an FTO-failure has occurred. Similarly, we can find an SO-failure may occur in Figure 4. Note that there is an important concept here: the transition from *src(t)* to *des(t)* is executed, therefore, in a real system, the *des(t)* state appears to be the same as the SO-failure state. However, in modeling, we cannot model the contradictory conditions of taking and not taking the safety transitions in the same mode. In Safecharts modeling, we have to specify these conditions into two different modes. When the safety transitions are executed under the expected semantics, then the *des(t)* state is a correct final destination. Otherwise, this state represents a failure state. As shown in Figure 5, we allow more flexible ways for system designers to model non-critical failures, because non-critical failures are defined case by case. In our work, we also assume that each type of failure could be recovered by some repair actions. If not, there might be too many dead states in the model. Nevertheless, we do not focus on how to deal with the dead states, except using them to specify properties representing system hazards.

Table 1. Safecharts Normal operations and possible failure modes for the Electric Lamp System

Type of condition (Transition Label)	**Safecharts Transition Label Descriptions**
Mandatory $(e[fcond]/a[l,u)\ \Rsh\ [G])$	e: someone enters the room
	$fcond$: lamp is normally functioning
	a: automatically turn on the lamp
	$[l,u) = [0,3)$
	G: insufficient light indoor
Prohibitory $(e[fcond]/a\ \Lsh\ [G])$	e: someone enters the room
	$fcond$: lamp is normally functioning
	a: automatically turn on the lamp
	G: sufficient light indoor
FTO $(e[fcond]/a[u,\infty)\overline{\Rsh}[G])$	electric lamp does not operate normally by the deadline time u.
SO $(e[fcond]/a\overline{\Lsh}[G])$	electric lamp automatically turns on when there is sufficient light indoor.
NC $(e[fcond]/a \leftrightarrow [G])$	e: someone enters the room
	$fcond$: true
	a: record non-critical problem
	G: the main service of the lamp is not supported as good as expected, for example, illumination is poor.

5 Transformation from Safecharts to Extended Timed Automata

The user-given Safecharts are automatically transformed into extended timed automata (ETA) models [8], which are simply timed automata [1] with discrete variables and synchronization labels. We first show in detail how system designers might model their safety-critical system in Safecharts, and then how the model is transformed into ETA for model checking.

5.1 Transformation from Safecharts to ETA

To analyze failures automatically in model checking, one of the primary goals is to model check the Safecharts model of a component or a system. However, Safecharts cannot be accepted as system model input by most model checkers, most of which accept only flat automata models. For example, extended timed automata (ETA) can be accepted by SGM. The three Safecharts layers, namedly safety, functional, and failure, must also be transformed into equivalent modeling constructs in ETA and specified as properties for verification.

There are three types of states in Safecharts: OR, AND, and BASIC. An OR-state or an AND-state, consists generally of two or more substates. All the substates in an AND-state are active simultaneously, while an OR-state is in exactly one of its substates.

A BASIC-state is represented simply by an ETA mode. The translations for OR-states and AND-states are performed as described in [11].

Before we translate Safecharts to ETA, we need to introduce three different types of transition urgency semantics [10]:

1. *Eager Evaluation* (ε): Execute the transition as soon as possible, *i.e.*, as soon as a guard is enabled. Time cannot progress as soon as a guard is enabled.
2. *Delayable Evaluation* (δ): Can put off execution until the last moment the guard is true. So time cannot progress beyond the *falling edge* of a guard, except when there is another enabled transition outgoing from the same state.
3. *Lazy Evaluation* (λ): The transition may or may not be executed.

The transition condition and assignment label $e[fcond]/a[l,u)\Psi[G]$ as described in Section 4.4 can be translated to the following equivalent semantics in ETA.

- $e[fcond]/a$: There is no safety clause on a transition in Safecharts, thus we can simply transform it to a similar transition in ETA. We have translated it in [6].
- $e[fcond]/a \upharpoonleft [G]$: There is *prohibition* enforcement value on a transition t. It signifies that the transition t is forbidden to execute as long as G holds. We also have performed this transformation in [6].
- $e[fcond]/a[l,u) \upharpoonright [G]$: There is *mandatory* enforcement value on a transition t. It signifies that the transition is forced to execute within [l,u) whenever G holds. The transformation to ETA was done in [6].
- $e[fcond]/a[u,\infty)\overline{\upharpoonright}[G]$: As shown in Figure 6, the transformations of the safety and the functional layers are same as those for a transition t' with the mandatory enforcement value, that is, $e[fcond]/a[l,u) \upharpoonright [G]$. A transition from some state *src(t)* to an *FTO-failure* state in Safecharts, as depicted in Figure 3, will be translated into a transition labeled with $(timer \geqslant u)^{\epsilon}$ from state *translator(t)* to an FTO-failure state in ETA, where $translator(t)$ is a state generated during the transformation of mandatory evaluation [6].
- $e[fcond]/a\overline{\upharpoonleft}[G]$: As shown in Figure 7, the transformations of the safety and the functional layers are same as those for a transition t' with the prohibition enforcement value, that is, $e[fcond]/a \upharpoonleft [G]$. The transition from some state *src(t)* to an *SO-failure* state in Safecharts, as depicted in Figure 4, will be translated into a transition labeled with $(e[fcond \wedge G]/a)^{\lambda}$ from state *src(t)* to an SO-failure state in ETA.
- $e[fcond]/a \upharpoonleftright [G]$: As shown in Figure 8, we translate a transition from some state *src(t)* to an *NC-failure* state in Safecharts, as depicted in Figure 5, into a transition labeled with $(e[fcond \wedge G]/a)^{\lambda}$ from state *src(t)* to an NC-failure state in ETA.

As shown in Figures 6, 7, and 8, the transitions representing repair actions in Safecharts are translated into similar corresponding transitions, without any special transformation. All the proposed transformations from the Safecharts layers to ETA modeling artifacts are trivially equivalent in semantics.

6 Implementation

The proposed model extensions in Safecharts for explicit failure representation and the related techniques for supporting failure analysis have all been integrated into the SGM

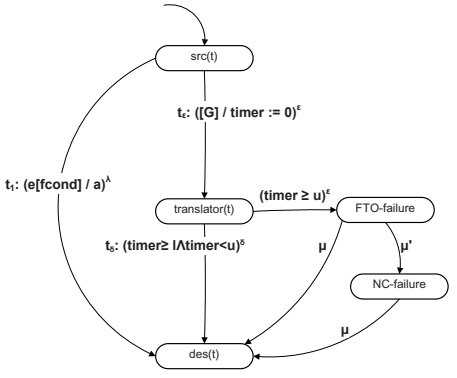


Fig. 6. FTO-failure in ETA

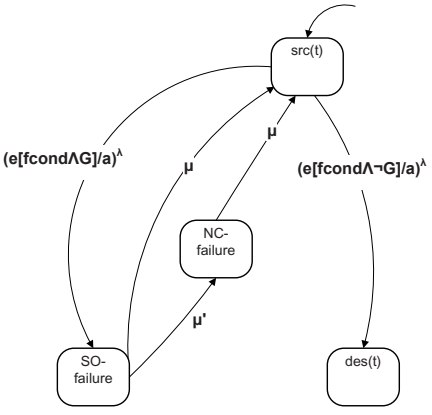


Fig. 7. SO-failure in ETA

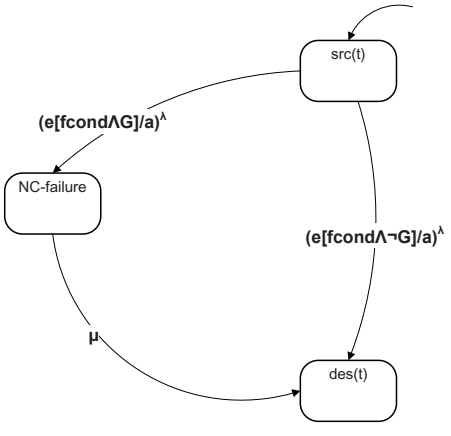


Fig. 8. NC-failure in ETA

model checker, on which we can perform verification and failure analysis. Mainly, what we had done are listed in the following.

- Extend the semantics of Safecharts to include the failure layer,
- Flatten the extended Safecharts into ETA models, and
- Transform all safety and failure modeling artifacts into equivalent constructs in the ETA models, which include the following.
  - Transition priority calculation from Safechart risk bands,
  - Support for prioritized transitions, and
  - Support for urgent transitions.

Details of the implementation for supporting prioritized transitions in the SGM model checker were given in [13]. The main issue is priority results in non-convex clock zones

which cannot be represented by a single difference bound matrix (DBM), a data-structure for representing time in a model checker. We solved this issue by proposing an optimal zone partitioning algorithm.

In our work, we also implemented support for urgent transitions in the SGM model checker by proposing a novel zone capping operation that restricts the clock zone in a mode (symbolic set of states) [7]. Due to page-limit, this part of the work is out-of-scope here and will not be described in details.

7 Application Example

To illustrate how the failure-extended Safecharts model aids in verifying safety-critical systems, we use an automatic water sprinkler system that is found in almost all high-rise apartments and garages for safety from fires. In such a system, the most important functionality is that when the sensors detect an abnormally high temperature degree, the controller sends a command to the sprinkler, which is supposed to start sprinkling water. We assume that the controller polls the temperature sensor once every second. We have used the extended Safecharts model to specify the relations between the failure state and the normal operating state for the sprinkler system, as shown in Figure 9, where *clock* is the triggering event for the system, and it arrives once every second. Totally, four failure states are specified, of which two are FTO-failures and two are SO-failures.

To analyze the possible failures in this sprinkler system, we have to first introduce its design. When the sensors detect an abnormally high temperature degree, it asserts a signal F, which is polled by the controller. If the sprinkler system is in the *Non-sprinkling* state, it is supposed to start sprinkling water within two seconds after the assertion of signal F. Otherwise, an FTO-failure occurs. When the sprinkler system is in the *Sprinkling* state, if the sensors detect a normal temperature degree, then the system is supposed to stop sprinkling within three seconds. Otherwise, there is also an FTO-failure. It is prohibited that the sprinkler system transits from the *Non-sprinkling* state to the *Sprinkling* state without an asserted signal F. It is also required in the design that the sprinkler system is prohibited from transiting from the *Sprinkling* state to the *Non-sprinkling* state while the signal F is asserted. Otherwise, an SO-failure occurs.

In Figure 9, $F = 1$ represents abnormally high temperature, $F = 0$ represents normal temperature, $R := 0$ is the action to stop sprinkling, and $R := 1$ is the action to start sprinkling. The sprinkler system is quite simple, thus only two effective repair actions, namely stop sprinkling and start sprinkling, are modeled in this system. There are no ineffective repairs and no NC-failure in this system.

We transformed the Safecharts model for this system automatically and generated the ETA model as shown in Figure 10, which was then input to the SGM model checker for verification and failure analysis. As we can see, the ETA model in Figure 10 is much more complex than the Safecharts model in Figure 9. Table 2 shows the transformation results for the sprinkler system. Even for this small example, the Safecharts model gives a reduction of 25% in the number of states and 25% in the number of transitions, compared to the generated ETA model. For larger and more complex systems, our proposed Safecharts with failure extensions will give greater benefits in terms of reduced sizes of models, compared to the more complex ETA models. The designers can thus focus

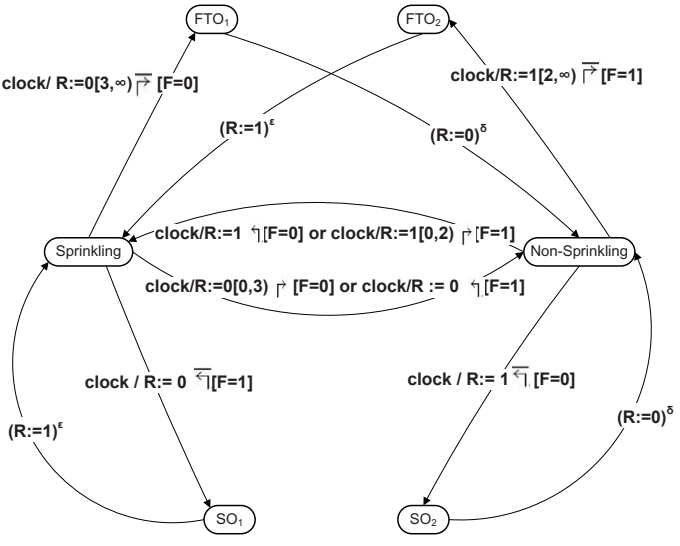


Fig. 9. Sprinkler System Controller in Safecharts

their attention and time to failure identification, classification, and safety analysis, rather than spending a lot of time and efforts in modeling them.

For this sprinkler system, we specified four different properties in CTL to analyze the four failures in the Safecharts model, namely, FTO_1, FTO_2, SO_1, and SO_2, as follows.

$$EF(mode = HAZARD), \text{where } HAZARD \in \{FTO_1, FTO_2, SO_1, SO_2\}$$

On model checking, the sprinkler system model satisfied all of the above properties. Witnesses to the satisfaction of these properties, generated by SGM, helped us in analyzing how the system would be faced with such hazards. For example, the witness $\langle(\textit{Non-Sprinkling}, F = 0), (\textit{Non-Sprinkling}, F = 1), (\textit{Non-Sprinkling}, F = 1 \wedge timer \geqq 2), (FTO_2)\rangle$, shows that one possible way in which the sprinkler system could face the FTO_2 failure is when the system is not sprinkling (*Non-Sprinkling*), the temperature becomes abnormally high ($F = 1$), and the timer has expired ($timer \geqq 2$). This property witness generated from a model checker contains more information than the minimum cut generated from *Fault Tree Analysis* (FTA) [12], which is a conventional and widely used analysis method for safety-critical systems. A minimum cut only

Table 2. Transformation Results for the Sprinkler System Example

	Numbers of Modes	**Numbers of Transitions**
Safecharts	6	12
	(-25%)	(-25%)
ETA	8	16

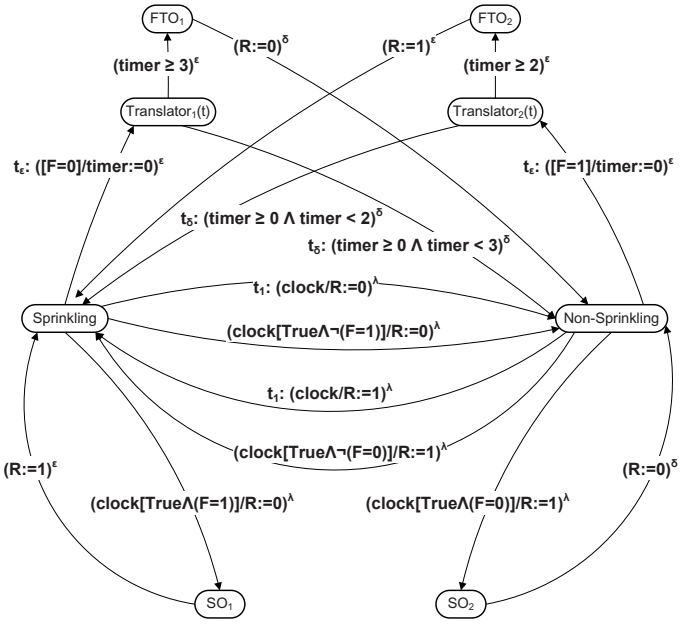


Fig. 10. Sprinkler System Controller in ETA

shows the logical combination of different faults or failures in a system that results in some hazard, however it cannot show the temporal sequencing of the faults that leads to a hazard. The same logical combination could have different temporal sequences, of which only some might result in a hazard and others would not. However, property witnesses show the exact computation run that would result in a hazard. Hence, there is more accurate information in a witness compared to that in a minimum cut.

Besides automatically generating a hazard witness for a single component failure, model checking our failure-extended Safecharts can also be used to verify a common safety-critical system property that specifies no single component failure results in a system failure, where a system failure is defined as the simultaneous occurrence of two component failures. We can thus check if the following property is satisfied.

$$AG\neg \bigvee_{i,j} [(mode(C_i) = HAZARD) \wedge (mode(C_j) = HAZARD)] \qquad (1)$$

where $HAZARD$ is either an FTO-failure or SO-failure and C_i is the ith component.

8 Conclusion

Nowadays, safety-critical systems are becoming more and more pervasive in our daily lives. To reduce the probability of tragedy, we must use a formal and accurate methodology to verify whether a safety-critical system is safe or not. We have proposed a formal method to verify safety-critical systems based on the Safecharts model and model

checking paradigm. Our methodology can be applied widely to safety-critical systems with a model-driven architecture. Through examples, we have shown the benefits of the proposed verification method and system model. We hope our methodology can have some real contribution in making the world a safer place to live in.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Bodsberg, L., Hokstad, P.: A system approach to reliability and life-cycle-cost of process safety systems. IEEE Transactions on Reliability 44(2), 179–186 (1995)
3. Bozzano, M., Villafiorita, A.: Improving system reliability via model checking: the FSAP/NuSMV-SA safety analysis platform. In: Anderson, S., Felici, M., Littlewood, B. (eds.) SAFECOMP 2003. LNCS, vol. 2788, pp. 49–62. Springer, Heidelberg (2003)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. Dammag, H., Nissanke, N.: Safecharts for specifying and designing safety critical systems. In: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, October 1999, pp. 78–87. IEEE Computer Society Press, Los Alamitos (1999)
6. Hsiung, P.-A., Chen, Y.-R., Lin, Y.-H.: Model checking safety-critical systems using Safecharts. IEEE Transactions on Computers 56(5), 692–705 (May 2007)
7. Hsiung, P.-A., Lin, S.-W., Chen, Y.-R., Huang, C.-H., Yeh, J.-J., Sun, H.-Y., Lin, C.-S., Liao, H.-W.: Model checking timed systems with urgencies. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 67–81. Springer, Heidelberg (2006)
8. Hsiung, P.-A., Wang, F.: A state-graph manipulator tool for real-time system specification and verification. In: Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA), pp. 181–188 (1998)
9. Johnson, M.E.: Model checking safety properties of servo-loop control systems. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 45–50. IEEE Computer Society Press, Los Alamitos (2002)
10. Gößler, G., Altisen, K., Sifakis, J.: Scheduler modeling based on the controller synthesis paradigm. Real-Time Systems 23, 55–84 (2002)
11. Lavazza, L. (ed.): A methodology for formalizing concepts underlying the DESS notation. In: ITEA (2001)
12. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Reading (1995)
13. Lin, S.-W., Hsiung, P.-A., Huang, C.-H., Chen, Y.-R.: Model checking prioritized timed automata. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 370–384. Springer, Heidelberg (2005)
14. Nissanke, N., Dammag, H.: Risk bands - a novel feature of Safecharts. In: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE), October 2000, pp. 293–301 (2000)
15. Wang, F., Hsiung, P.-A.: Efficient and user-friendly verification. IEEE Transactions on Computers 51(1), 61–83 (2002)

Using Deductive Cause-Consequence Analysis (DCCA) with SCADE

Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{guedemann,ortmeier,reif}@informatik.uni-augsburg.de

Abstract. Esterel Technologies' SCADE Suite is one of the most important development tools for software for safety-critical systems. It is used for designing many critical components of aerospace, automotive and transportation applications. For such systems safety analysis is a key requirement in the development process.

In this paper we show how one formal safety analysis method – Deductive Cause-Consequence Analysis (DCCA) – can be integrated in the SCADE framework. This method allows for performing safety analysis largely automatically. It uses SCADE's semantical model and SCADE's built in verification engine *Design Verifier*. So the whole analysis can be done within one tool. This is of big importance, as a key feature for the acceptance of formal methods in broad engineering practice is, that they can be applied in an industrial development suite.

We illustrate the method on a real world case study from transportation domain and discuss possible next steps and limitations.

Keywords: formal methods, safety critical systems, deductive cause consequence analysis, dcca, safety analysis, dependability, SCADE.

1 Introduction

The role of software in embedded systems development is becoming more important, as more and more features are implemented in software instead of hardware. One example for this are the *x-by-wire* features in automotive application or avionics. Here, safety critical tasks like steering or breaking are controlled by electronic devices which are connected electrically. In contrast to traditional techniques, there no longer exists a physical link between controller and actuator. The result is that, the controlling software now becomes a safety critical system itself, as it takes over a large portion of safety critical tasks.

Systems for avionics must for example be DO178B [14] compliant. To achieve this, software must also be certified. For embedded systems SCADE Suite of Esterel Technologies is a widely accepted, state-of-the-art toolkit which allows to develop software graphically and has a certified development process for safety critical systems. The developed models are automatically converted into executable C code (for various target systems) using a certified code generator.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 465–478, 2007.

SCADE Suite also allows to automatically verify safety properties of the developed software using the built-in *Design Verifier* of Prover Technologies.

Traditionally SCADE models are mainly used for designing software. Hardware elements and safety analysis are not being modelled. On the other hand (formal) safety analysis methods need to reason about hardware and software, to give answers like "How many failures can be tolerated?" etc. In the last years a lot of advances in the domain of formal safety analysis have been made like formalisations of fault tree analysis (FTA) [15, 5, 2], fault injection (FI) [1] or the unifying methods of formal cause-consequence analysis (DCCA) [13, 12]. Most of these techniques are semantically grounded on either linear temporal logic (LTL) or computational temporal logic (CTL) and use some variation of finite automata for system modelling. Unfortunately, SCADE does not have proof support for LTL nor for CTL.

In this paper we show how one of the listed methods can be adopted, such that it can be used with SCADE's verification capabilities. This is of big importance for practical applications as it allows system developers to analyse the model which will be used for code generation (and do not need to manually translate it into another modelling language). The paper is structured as follows: Sect. 2 gives an introduction on DCCA. Sect. 3 shows how to conduct DCCA in SCADE. It also sketches a proof idea on why this adaptation of DCCA is semantically correct. Sect. 4 describes a case study and illustrates the application of DCCA in SCADE. Sect. 5 concludes the paper.

2 DCCA

This section describes briefly the formal semantics of DCCA. The formalisation is done with Computational Tree Logic(CTL) [4] using finite automata as system models. The use of CTL and finite automata allows to use powerful model checkers like SMV [9] to verify the proof obligations. The goal of DCCA is the following:

> Given an unwanted, hazardous situation H and a set of component failure modes Γ. Determine, which combinations of failures modes may (1) potentially cause an hazard and (2) are minimal in the sense, such that no proper subset of these failure modes can cause the hazard.

This is the standard question, which most safety analysis methods try to answer. In the following we assume that a set of hazards $\{H\}$ on system level and a set of possible basic component failures Δ modes is given. Both data may be collected by other safety analysis techniques like failure-sensitive specification [11] or HazOp [6].

2.1 Failure/Hazard Automata

For formal safety analysis failure modes must be explicitly modelled. The modelling can be naturally split into two parts. One part is modelling the occurrence pattern of the failure mode and the other is modelling the failure mode itself.

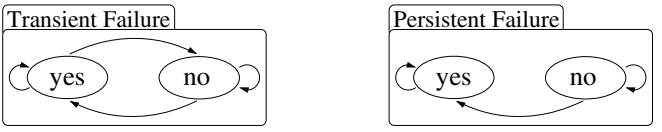


Fig. 1. Failure automata for transient and persistent failures

An "occurrence pattern" describes how and when the failure mode occurs. For example does the failure mode occur nondeterministically (like packet loss in IP traffic) or does it occur once and forever (like a broken switch) or does it occur only during certain time intervals (like until the next maintenance). To model this failure automata are used. Figure 1 shows two such failure automata.

The left automaton models a transient failure which can nondeterministically occur and disappear. The right one models a persistent failure, which happens once and stays forever (e.g. a broken relay). Maintenance etc. may be modelled analogously. Failure predicates δ are then defined as "failure automaton for failure mode $\tilde{\delta}$ in state *yes*". For readability the symbol δ is used for both the predicate and the automaton describing the occurrence pattern.

The second step is to model the direct effects of failure modes. This is usually done by adding transitions to the model of the system with conditions of the form $\varphi \wedge \delta$. This assures, that these additional transitions – which reflect erroneous behaviour – may only be taken, when a failure automaton is in state *yes* i.e. when a failure occurs[1].

A similar approach may be used to define predicates for system hazards. If the system hazard can not be described by a predicate logic formula directly, then often an observer automaton may be implemented such that whenever the automaton is in an accepting state, the hazard has occurred before [15]. This allows to describe the hazard as a predicate logic formula on the states of the observer automaton. However in practical applications hazards may mostly be described by predicate logic formulas.

2.2 Critical Sets

The central part of the analysis is the definition of a temporal logic property which says, whether a certain combination of failure modes may lead to the hazard or not. This property is called *criticality* of a set of failure modes.

Definition 1. *critical set / minimal critical set*
For a system SYS and a set of failure modes Δ a subset of component failures $\Gamma \subseteq \Delta$ is called critical for a system hazard, which is described by a predicate logic formula H if

$$SYS \models \mathbf{E}(\overline{\Gamma}\ \mathbf{until}\ H) \ \ \textit{where}\ \ \overline{\Gamma} := \bigwedge_{\delta \in (\Delta \setminus \Gamma)} \neg\, \delta$$

[1] It can also be shown, that the integration of failure modes is monotone – wrt. to traces inclusion to the original model – if a certain set of modelling rules is followed (see [10] for details).

We call Γ a minimal critical set if Γ is critical and no proper subset of Γ is critical.

Here, **E(φ until ψ)** denotes the existential CTL-UNTIL-operator. It means there exists a path in the model, such that φ holds until the property ψ holds. The property *critical set* translates into natural language as follows: "There exists a path such that the system hazard occurs without the previous occurrence of any failures except those which are in the critical set". In other words this means, it is possible that the systems fails, if only the component failures in the critical set occur.

Intuitively, criticality is not sufficient to define a cause-consequence relationship. It is possible that a critical set includes failure modes, which have nothing to do with the hazard. Therefore, the notion *minimal critical set* also requires that no proper subset is critical. Minimal critical sets really describe what one would expect for a cause-consequence relationship in safety analysis to hold: the causes may - but not necessarily - lead to the consequence and second all causes are necessary to allow the consequence to happen. The goal of DCCA is to find minimal critical sets of failure modes. A DCCA is called complete if all minimal critical sets are found.

Testing all sets by brute force would require an effort exponential in the number of failure modes. However, DCCA may be used to formally verify the results of informal safety analysis techniques. This reduces the effort of DCCA a lot, because the informal techniques often yield good "initial guesses" for solutions. Note also, that the property critical is monotone with respect to set inclusion i.e. $\forall \Gamma_1, \Gamma_2 \subseteq \Delta : \Gamma_1 \subseteq \Gamma_2 \Rightarrow (\Gamma_1$ is critical set $\Rightarrow \Gamma_2$ is critical set). This helps to reduce proof efforts a lot.

3 DCCA in SCADE

Integrating DCCA in SCADE is not directly possible. This is because of two major problems. Firstly, SCADE does not allow for nondeterministic automata. The idea behind this "feature" is, that the toolkit was traditionally used for software development (where nondeterminism is mostly unwanted). The second (somewhat harder) problem is that SCADE only allows to verify properties of the form "on all paths it is at all times the case that φ holds" or short in CTL notation "AG(φ)". Therefore some adaptations are necessary and integration is only possible for a specific type of failure modes.

3.1 Semantics and Syntax of SCADE Models

The semantics of SCADE models is based on data flows. Each single data flow can be seen as a sequence of values for a variable. The set of all possible data flows defines the semantics of the model. So semantically this model is very similar to the set of traces defined by a Kripke structure.

On the other hand specification in SCADE is very different. Every SCADE model has a fixed set of input and output variables. A syntactic convention is

that all inputs are drawn on the left side of a block and all outputs are drawn on the right side. SCADE models are built of blocks. Each block has a fixed input and output interface. Basic blocks are composed to larger models by connecting outputs to inputs. System inputs maybe be connected to any component's input and system outputs can be any component's output. Note that input data is processed *immediately* throughout the whole model. To avoid inconsistencies direct feedback is not allowed[2]. However, there exists a special operator **FBY** whose output is the input of the last step[3]. This operator is often used if data feedback is needed. A simple SCADE model is shown in Fig. 2.

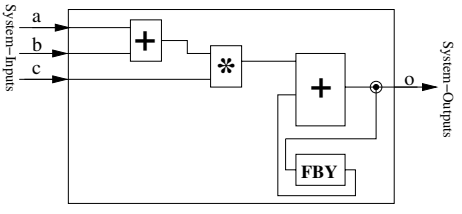


Fig. 2. A simple SCADE model

This model has three inputs a, b and c. The first two inputs are added ("+"-operator) and multiplied ("*"-operator) with the third input. The output o is the accumulation of all previous results ("FBY"-operator). An example data flow $(a, b, c, o)_i$ of the system is: $(1, 2, 3, 9), (1, 2, 2, 15), (2, 2, 2, 23), ...$[4].

SCADE also allows to embed state machines in single blocks. Here, the semantics is that the state machine executes *exactly* one step for every step of the data flow. Note that state machines in SCADE must always be deterministic. So direct modelling of failures as described in Sect. 2.1 is not possible.

3.2 Semantics of DCCA in SCADE

Occurrence patterns of failure modes are modelled in specific failure mode blocks and the hazard is modelled as a SCADE block as well. The output of these blocks is true if the failure mode/hazards occurs and false otherwise.

The basic architecture for applying DCCA to a SCADE model is shown in Fig. 3. Outputs of failure blocks are connected as inputs to the system model. The hazard is connected to the system outputs (outputs may also be some internal variables which are only used for defining the hazard). Failure blocks use as input specific (system) failure inputs.

In Fig. 4 a failure block is shown. The block contains a state machine which is similar to the failure automata shown in Fig. 1. However, there are slight

[2] This is checked by a syntactic analysis.

[3] For the initial state the output of this operator must be defined explicitly (for more details see the documentation of SCADE).

[4] Because: (1+2)*3 + 0=9; (1+2)*2+9=15; (2+2)*2+15=23.

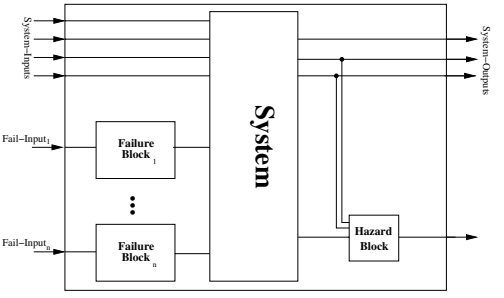


Fig. 3. System model

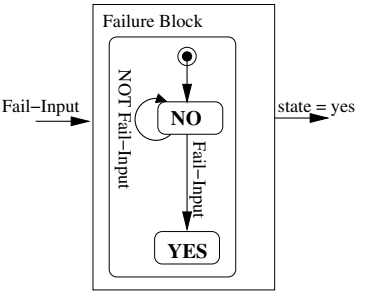


Fig. 4. Failure block

differences. First of all the block has an input, which indicates when the failure starts occurring ($Fail-Input$). This is not redundant as only system inputs are allowed to be nondeterministic in SCADE; state machine must be deterministic. The output of the block evaluates to true if and only if the state machine is in state "yes". This output is then triggering the direct effects of the failure mode as usual. Similar blocks for transient failures are possible.

For DCCA in SCADE two restrictions are necessary. The first one is, that failure blocks must not contain a feedback link from the system. So for example a failure block may not also depend on any of the systems outputs[5]. The second restriction is, that each failure block must have the choice to stay in state "no" whenever it is in state "no". Informally this means: "Every failure happens nondeterministically." The consequence is that at all times there exists some future in which the failure will not occur (if it hasn't already occurred).

It is clear that these two restriction are an abstraction of the real world. However this is not a real problem in the context of safety analysis, because the only relevant question in this domain is, which failures caused hazards (i.e. which components failed before the whole system failed). This approach and the architecture of Fig. 3 lead to the following definition for DCCA:

Definition 2. *critical set / minimal critical set (SCADE)*
For a system SYS and a set of failure mode inputs Δ a subset of failure mode inputs $\Gamma \subseteq \Delta$ is called critical for a system hazard, which is described by an hazard block H if the boolean output flow of block H is not always false *– under the restriction that all failure mode inputs of $\Delta \setminus \Gamma$ are permanently false. It is called minimal critical, if no proper subset of Γ is critical.*

This means in informal language: If no failures of $\Delta \setminus \Gamma$ occur and the hazard H occurs, then the failures modes of Δ are critical. So the process for DCCA is as follows: (1) Add failure mode blocks and hazard blocks, (2) connect failure mode blocks, hazard block and system block as shown in Fig. 3, (3) model direct effects of failure modes (this is analogous to modelling direct effects in finite

[5] An example where this could make sense, is that a crash of an aircraft deterministically triggers the failure mode "loss of power".

automata) and (4) for each set of failure modes check (with SCADE's *Design Verifier*) if the output of the hazard block is always false under the restriction that all other failure mode inputs are false.

3.3 Correctness of DCCA Implementation

This section outlines a sketch of a proof why this adaptation of DCCA is correct. CTL semantics are only where necessary. The following CTL-operators are used:

- "A" denotes the "on-all-paths"-operator
- "E" denotes the "there-exists-a-paths"-operator
- "G" means "at-all-times"
- "F" means "in-some-future-time"

Complete semantics may be found - for example - in [3]. The semantics of the CTL proof obligation for DCCA ($\mathrm{SYS} \models \mathbf{E}(\overline{\Gamma}\ \mathbf{until}\ H)$) is defined as follows:

$$\exists \pi \in \mathrm{SYS} : \exists k \geq 0 : \mathrm{SYS}, \pi_k \models H \text{ and } \forall j : 0 \leq j < k : \mathrm{SYS}, \pi_i \models \overline{\Gamma}$$

This means that there exists a trace π in the system on which at some point in time k the hazard H occurs and on all points in time before no failures in the set $\Delta \setminus \Gamma$ occurred.

As said above, this is not directly expressible in SCADE, as only safety properties of the form "on all paths at all times" can be expressed. Therefore the negated hazard is integrated as proof goal into a verification project as additional boolean output data flow and for every critical set Γ, the range of possible **failure inputs** is restricted, such that all failure inputs of $\Delta \setminus \Gamma$ are permanently set to false. Semantically this means that a system SYS_Γ is used for verification. The formal definition of this system is:

$$\pi \in \mathrm{SYS}_\Gamma := (\pi \in \mathrm{SYS} \wedge \pi \models \mathbf{G}\overline{\Gamma})$$

Here $\overline{\Gamma}$ is defined as in definition 1. The verification goal for showing the criticality of a set of failure modes in SCADE is then:

$$\mathrm{SYS}_\Gamma \not\models \mathbf{AG}\neg H$$

For correctness of this integration it must be shown, that proving the above formula is equivalent to proving the DCCA formula of definition 1. So the following equivalence must hold:

$$\mathrm{SYS}_\Gamma \not\models AG\neg H \Leftrightarrow \mathrm{SYS} \models \mathbf{E}(\overline{\Gamma}\ \mathbf{until}\ H)$$

This means informally: "A set of failure modes can be proven critical in SCADE if and only if it can be proven critical in a corresponding CTL

proving environment." The sketch of the proof is as follows (note that H and $\overline{\Gamma}$ propositional formulas):

$$\begin{gathered}
\mathrm{SYS}_\Gamma \not\models \mathbf{AG}\neg H \\
\Leftrightarrow \\
\exists \pi \in \mathrm{SYS}_\Gamma : \pi \models \mathbf{F}H \\
\Leftrightarrow (*) \\
\exists \pi \in \mathrm{SYS} : \pi \models \mathbf{F}H \text{ and } \pi \models \mathbf{G}\overline{\Gamma} \\
\Leftrightarrow (**) \\
\exists \pi \in \mathrm{SYS} : \pi \models (\overline{\Gamma} \textbf{ until } H) \\
\Leftrightarrow \\
\mathrm{SYS} \models \mathbf{E}(\overline{\Gamma} \textbf{ until } H)
\end{gathered}$$

For proving (*) the definition of SYS_Γ is needed. The downwards equivalence of (**) is trivial. Proof of the upwards direction of $(**)$ is a little tricky. Informally one must show, that for every path (which has a finite prefix where $\overline{\Gamma}$ holds in all states and on which H occurs) there also exists a continuation of this path such that $\overline{\Gamma}$ never occurs.

Formally:

$$\begin{gathered}
\pi \in \mathrm{SYS} : \exists i \in N : \forall j < i \; \pi_j \models \overline{\Gamma} \text{ and } \pi_i \models H \\
\Rightarrow \\
\exists \tilde{\pi} \in \mathrm{SYS} : \forall j \leq i : \pi_i = \tilde{\pi}_i \text{ and } \tilde{\pi} \models \mathbf{G}\overline{\Gamma}
\end{gathered}$$

This implication is true for the considered SCADE models. All variables that appear in the formula $\overline{\Gamma}$ are failure block outputs. More specifically they are of the form "¬(state machine in state yes)". Initially the state machine is in state "no". Because of the restrictions described in Sect. 3.2 a transition to state "yes" is only possible if the external input "Fail-Input" is true (see Fig. 4). This input is a nondeterministic system input. Therefore every trace where the state machine is in state "no" can be extended such that the state machine stays always in this state. This closes the proof of the upward direction of (**).

4 Application

As an example for the application of DCCA we present an analysis of a radio-based railroad crossing. This case study is the reference case study of the German research councils (DFG) priority program 1064. This program aims at bringing together field-tested engineering techniques with modern methods of the domain of software engineering.

The German railway organisation, Deutsche Bahn, prepares a novel technique to control railroad crossings: the decentralised, radio-based railroad crossing control. This technique aims at medium speed routes, i.e. routes with maximum speed of 160 km/h. An overview is given in [7].

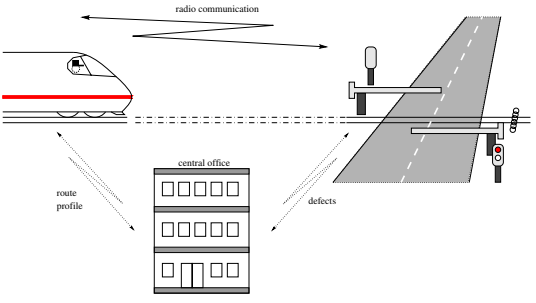


Fig. 5. Radio-based railroad crossing

The main difference between this technology and the traditional control of railroad crossings is that signals and sensors on the route are replaced by radio communication and software computations in the train and railroad crossing. This offers cheaper and more flexible solutions, but also shifts safety critical functionality from hardware to software.

Instead of detecting an approaching train by a sensor, the train computes the position where it has to send a signal to secure the level crossing. To calculate the activation point the train uses data about its position, maximum deceleration and the position of the crossing. Therefore the train has to know the position of the railroad crossing, the time needed to secure the railroad crossing, and its current speed and position. The first two items are memorised in a data store and the last two items are measured by an odometer. For safety reasons a safety margin is added to the activation distance. This allows for compensating some deviations in the odometer. The system works as follows:

The train continuously computes its position. When it approaches a crossing, it broadcasts a 'secure'-request to the crossing. When the railroad crossing receives the command 'secure', it switches on the traffic lights, first the 'yellow' light, then the 'red' light, and finally closes the barriers. When they are closed, the railroad crossing is 'secured' for a certain period of time. The 'stop' signal on the train route, indicating an insecure crossing, is also substituted by computation and communication. Shortly before the train reaches the 'latest braking point' (latest point, where it is possible for the train to stop in front of the crossing), it requests the status of the railroad crossing. When the crossing is secured, it responds with a 'release' signal which indicates, that the train may pass the crossing. Otherwise the train has to brake and stop before the crossing. The railroad crossing periodically performs self-diagnosis and automatically informs the central office about defects and problems. The central office is responsible for repair and provides route descriptions for trains. These descriptions indicate the positions of railroad crossings and maximum speed on the route. The safety goal of the system is clear: it must never happen, that the train is on the crossing and a car is passing the crossing at the same time. A well designed control

system must assure this property — at least as long as no component failures occur. The corresponding hazard H is "a train passes the crossing and the crossing is not secured". This is the only hazard which we will consider in this case study.

4.1 Formal Model

We now give a brief description of the formal system model in SCADE notation. In Fig. 6 the SCADE model of the described system is shown.

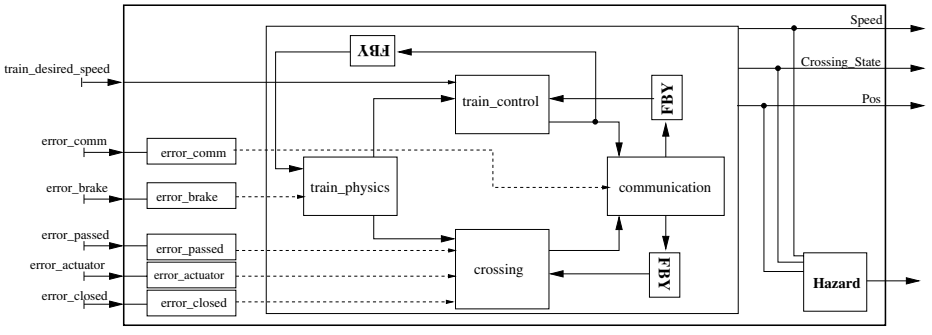


Fig. 6. SCADE model of the system

The system model itself is made of four blocks: one for modelling the physics of the train, one for modelling the railroad crossing, one for the communication between the train and the crossing and one for modelling the control logic of the train. To this system model failure mode blocks and a hazard block have been added.

The following five different types of failure modes were taken into account:

- **Failure of the brakes:** *error_brake* - This error describes the failure of the brakes. It's direct effects are modelled in block *train_physics*.
- **Failure of the communication:** *error_comm* - This error describes the failure of the radio communication. It's direct effects are modelled in block *communication*.
- **Failure of the barriers closed sensor:** *error_closed* - This error describes that the crossing signals *Crossing_Secured*, although it is not closed. It's direct effects are modelled in block *crossing*.
- **Failure of the barriers' actuator:** *error_actuator* - This error describes that the actuator of the crossing fails. It's direct effects are modelled in block *crossing*.
- **Failure of the train passed sensor:** *error_passed* - This error describes that the sensor detecting trains which passed the crossing fails. It's direct effects are modelled in block *crossing*.

The hazard of this system is, that the train passes an insecure crossing. We call this hazard collision H_{Col}. This is modelled by the following formula:

$$H_{Col} := Pos \leq Pos_{ds} \wedge Pos + Speed > Pos_{ds} \wedge \neg Crossing\_State = closed$$

In this formula Pos_{ds} is an abbreviation for the position of the crossing (ds = danger spot). It describes the location of the crossing. H_{Col} evaluates to true, if and only if the train passes the crossing and the barriers are not closed. In Fig. 7 the SCADE block for this formula is shown.

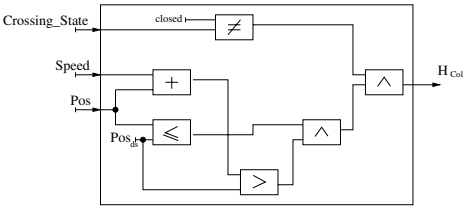


Fig. 7. Hazard block

Because of lack of space we will not show the whole model here, but only describe the block "crossing" in detail.

The block in Fig. 8 shows the model of the crossing. It has inputs for a received closing request (*comm_close_rcv*) and for data of the sensor on the track after the crossing (*sensor_passed*). Additionally, direct effects of failure modes "error_actuator", "error_passed" and "error_close" are modelled in this block. We will not explain how direct failure effects of failure modes can be modelled but refer to [10] where methodology and modelling rules are described.

The component works as follows: Initially the barriers are *opened*. When the crossing receives a close request from an arriving train - i.e. input *comm_close_rcv* becomes true, the barriers start *closing*. This process takes some time. This is modelled by timer block *Timer_Closing*. After a certain amount of time the barriers are *Closed*. They will remain closed until the train has passed the crossing (input *sensor_ passed*). The barriers reopen automatically after a defined time interval. This is a standard procedure in railroad organisation, as car drivers tend to ignore closed barriers at a railroad crossing if the barriers are closed too long. So it is better to reopen the barriers, than having car drivers slowly driving around the closed barriers[6]. The reopening is controlled by another timer *Timer_Closed*.

The direct effects of failures which are modelled in this component are marked with grey boxes in Fig. 8. A faulty signal from the sensor, which detects when the train has passed the crossing will also open the crossing. This is modelled by *error_passed*. The barriers may also get *stuck*, if the actuator fails (*error_actuator*).

[6] The target systems for this technologies are railroad crossings in rural areas where only one lane of the road is blocked by a barrier.

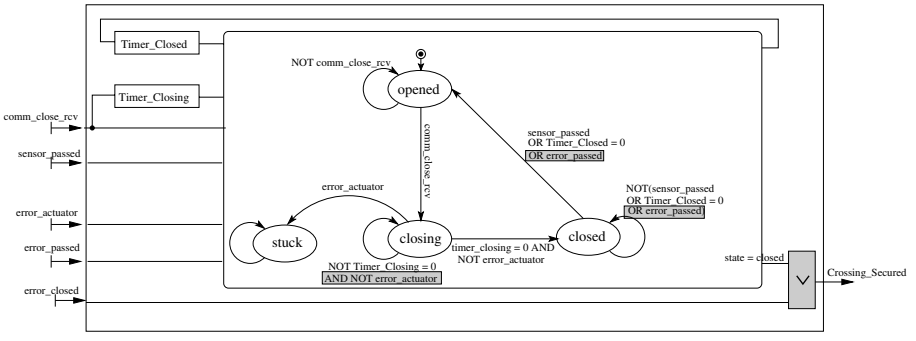


Fig. 8. Model of the crossing

Error_Closed models the failure of the sensor which is detecting the end position of the barrier. This may lead to a scenario, where the crossing signals *Crossing_Secured* although the barriers are not closed. The other components are modelled analogously.

4.2 Results of DCCA

This model was used to analyse the system with DCCA as described in Sect. 3. All proofs were done using the *Design Verifier* model checker from *Prover Technologies* which is integrated into SCADE. This verification engine allows for two different verification strategies: proof and debug. The first strategy can give rigorous proof of a property (by covering the complete state space of the model) while the debug-strategy is much faster but aims at finding counter examples. Unfortunately using the proof strategy for proving sets of failure modes to be non-critical did not work, as even for one failure mode, running time was more than three days. This is because linear time model checking has a higher complexity than for example CTL model checking [8] where the necessary proofs took less than one minute [12] with the SMV model checker.

Criticality of sets of failure modes can be shown with the *debug strategy*. The verification to show that a set is critical was really fast (a couple of seconds to a minute). For all of these critical sets a simulation file is generated which holds the counterexample with values for input variables that lead to the violation of the corresponding proof objective. This file can then be loaded and simulated on the given model to observe its behaviour. The design verifier could prove the following sets to be critical:

- {error_passed}
- {error_comm, error_brake}
- {error_closed, error_actuator}
- {error_brake, error_actuator}

It was not possible (in a runtime of three days before interruption) to show, that the other sets of failure modes are not critical. We did the same analysis with

the SMV tool (by translating the model into an equivalent Kripke structure). The result was that (1) all critical sets could also be proven critical with SMV, (2) one additional critical set ({error_comm, error_closed}) was found[7] and (3) that all other sets were not critical. The runtime of SMV is – for this example – very fast. Building the BDD and verifying all 13 proof obligations[8] took less than one minute.

5 Conclusion

Integration of DCCA into an industrial system development environment is possible. However, due to the limited expressiveness of the verification engine restrictions to occurrence patterns of failure modes were necessary. On the other hand the possibility to carry out formal safety analysis without the need to translate system models by hand is a great step forward for integration of formal methods into an industrial development process. We showed the application to a real world case study, presented the results and compared them in terms of time needed for verification to CTL based verification methods.

Using the violation of certain safety properties is a rather natural way of expressing hazards in SCADE. Defining failure modes via nondeterministic failure blocks is a rather general concept for integration of failures into a system model, which integrates smoothly into SCADE. This process can easily be done by an experienced system engineer. The algorithmic capabilities of SCADE's verification engine and the temporal logic available is not as strong as that of state-of-the-art symbolic model checking tools. Nevertheless, formal safety analysis can be done in the framework and yields new results. It can be used within the SCADE framework with only little additional effort and can be used as help for system developers to rate their systems.

Future work will now investigate, if SCADE models can be automatically (and semantically equivalent) translated into Kripke structures. This will allow for much more powerful proof support as well as for more expressive logics.

References

[1] Abdulla, P.A., Deneux, J., Stalmarck, G., Agren, H., Akerlund, O.: Designing safe, reliable systems using SCADE. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, Springer, Heidelberg (2006)

[2] Coppit, D., Sullivan, K.J., Dugan, J.B.: Formal semantics of models for computational engineering: A case study on dynamic fault trees. In: International Symposium on Software Reliability Engineering, IEEE, Los Alamitos (2000)

[3] Peled, D.A., Clarke Jr., E.M., Grumberg, O.: Model Checking. The MIT Press, Cambridge (1999)

[7] For this set of failure modes the Design Verifier repeatedly crashed with an exception. This bug has been fixed now by the developers.

[8] In conclusion, by use of monotony instead of 2^5 proof obligations only 13 proofs were necessary to determine all minimal, critical sets.

[4] Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics, pp. 995–1072. North-Holland Pub. Co./MIT Press (1990)
[5] Górski, J., Wardziński, A.: Formalising fault trees. In: Redmill, F., Anderson, T. (eds.) Achievement and Assurance of Safety, Springer, Heidelberg (1995)
[6] Kletz, T.A.: Hazop and HAZAN notes on the identification and assessment of hazards. Technical report, Inst. of Chemical Engineers, Rugby, England (1986)
[7] Klose, J., Thums, A.: The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg (2002)
[8] Kupferman, O., Vardi, M.Y.: Relating linear and branching model checking. Technical Report TR98-301, 18 (1998)
[9] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1990)
[10] Ortmeier, F.: Formale Sicherheitsanalyse. PhD thesis, Universität Augsburg (in German) (2006)
[11] Ortmeier, F., Reif, W.: Failure-sensitive specification: A formal method for finding failure modes. Technical Report 3, Institut für Informatik, Universität Augsburg (2004)
[12] Ortmeier, F., Reif, W., Schellhorn, G.: Formal safety analysis of a radio-based railroad crossing using deductive cause-consequence analysis (DCCA). In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, Springer, Heidelberg (2005)
[13] Ortmeier, F., Reif, W., Schellhorn, G.: Deductive cause-consequence analysis (DCCA). In: Proceedings of IFAC World Congress, Elsevier, Amsterdam (2006)
[14] RTCA. DO-178B: Software considerations in airborne systems and equipment certification (December 1st 1992)
[15] Thums, A.: Formale Fehlerbaumanalyse. PhD thesis, Universität Augsburg, Augsburg, Germany (in German) (2004)

Experimental Assessment of Astrée on Safety-Critical Avionics Software

Jean Souyris and David Delmas

Airbus France S.A.S.
316, route de Bayonne
31060 TOULOUSE Cedex 9, France
{Jean.Souyris,David.Delmas}@Airbus.com

Abstract. Astrée is a parametric Abstract Interpretation based static analyser that aims at proving the absence of RTE (Run-Time Errors) in control programs written in C. Such properties are clearly safety properties since the behaviour of a C program is undefined after a RTE. When it analyses a program of the class for which it is specialised, Astrée is far more precise than general purpose static analysers. Nevertheless, for safety and industrial reasons, the small number of false alarms first produced by the tool must be reduced down to zero by a new fine tuned analysis. Through the description of experiments made on real programs, the paper shows how Abstract Interpretation based static analysis will contribute to the safety of avionics programs and how a user from industry can achieve the false alarm reduction process via a dedicated method.

Keywords: avionics software, safety, verification, Abstract Interpretation, static analysis, run-time errors, Astrée.

1 Introduction

Safety-Critical avionics systems are composed of sensors, actuators, hydraulic pipes, electrical cables and computers. All these components must contribute to system safety. The contribution of hardware components to the safety objectives is being assessed using well-known techniques, that unfortunately do not apply to software components. Avionics software is considered "good" provided its development process is DO178B compliant, "good" meaning that the program implements its specification safely.

A way to view software verification from a safety-oriented perspective could be to get some safety properties as inputs of the software development process and to demonstrate that these properties hold. How do we perform such a demonstration?

Basically, software verification involves three kinds of techniques: testing, intellectual analyses and formal verification. The first one, i.e., testing, is the most popular. It has the advantage of being based on actual executions of the program under test. Moreover, tests can be performed in an environment very close to the operational one. Nevertheless, testing is not sound. Indeed, even during the heaviest possible test campaign, it is impossible to verify every possible execution of the program under test.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 479–490, 2007.

The second verification technique, i.e., intellectual analyses, is often used as a complement to testing. But this not really a proof that a property holds. The last technique, i.e., formal verification, aims at formalising the proof that a program satisfies some properties. But "formal" is not enough, for obvious industrial reasons it must also be automatic.

Ideally, safety properties of software should be demonstrated formally by automatic tools. Indeed, when any sound formal technique allows to prove a property, this means there exists no execution of the program that falsifies the property. We recall that, by principle, testing is unable to do so.

Existing formal proof techniques are Model-Checking, Theorem Proving and Abstract Interpretation ([3], [4], [5]) based Static Analysis. The items to be verified being final products, i.e. source or binary code, Model-Checking is not considered relevant. For different reasons, the use of Theorem Proving to verify safety properties on complete real-size programs seems far beyond software engineers capabilities.

Currently, to prove safety properties on real-life safety-critical programs, good candidates are static analysers aiming at proving a specific class of properties, i.e., WCET (Worst-Case Execution Time), Stack analysis, Floating-point calculus, Run-Time Errors, Memory usage properties. Indeed, Abstract Interpretation based static analysers now do scale, i.e., they are able to analyse complete safety-critical programs. They are a first step towards the proof of almost all safety properties of software. A second big step will be the proof of so-called "user-defined" safety properties, as opposed to the above analysers, in which the properties are "hard-coded. For instance, Fluctuat C (CEA, French nuclear research centre) does not analyse C programs in order to prove properties submitted by the user, but to compute a safe approximation of the rounding errors in floating-point calculus and stability properties.

The rest of the paper is focused on the proof of absence of Run-Time Errors in safety-critical avionics programs. Such a property is clearly a safety objective for software since after most RTE, the behaviour of a program is undefined. Indeed, although some RTE raise an interrupt, e.g., floating-point overflow, other errors like *out-of-bounds array access* or *dereferencing an invalid pointer*, might let the erroneous program do very bad things before any failure is detected.

This paper is essentially an industrial experience report on the use of the Astrée Abstract Interpretation based static analyser.

2 Astrée

Astrée is a parametric Abstract Interpretation based static analyser that aims at proving the absence of Run-Time Errors in programs written in C.

The absence of RTE has been defined in [2, §2]: "*The absence of run-time errors is the implicit specification that there is no violation of the C norm (e.g., array index of bounds), no implementation-specific undefined behaviours (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetic operators on short variables should not overflow the range* `[-32768,32767]` *although, on the specific platform, the result can be well-defined through modular arithmetics).*"

Notion of false alarm. To prove the absence of Run-Time Errors, Astrée computes an abstract invariant that safely represents (thanks to the Abstract Interpretation theoretical framework) a superset of all possible executions of the analysed program. Actually, the first abstraction Astrée implements consists in computing an over-approximation of the reachable states of the analysed program, without storing the execution traces, i.e., the sequences of states. Intuitively, a program state is a pair (program location, memory state). The abstract invariant is the set of "per program location abstract invariants", otherwise called local invariants. A local invariant is defined for a particular location in the program and it is an over-approximation of all possible memory states at that point.

By definition of Run-Time Errors, Astrée knows at which program locations an RTE might occur. Therefore, at each such location in the program, the diagnosis part of Astrée checks if the RTE condition might be satisfied by the local invariant.

Unfortunately, as the abstract invariant is an over-approximation of all possible reachable states and, consequently, of all possible executions of the analysed program, any RTE condition, like "division by zero", might be satisfied by "false" executions only. By "false executions" we mean: executions that cannot occur in reality. When it is the case, the alarm is called a "false alarm".

Zero false alarm is mandatory, for industrial reasons**.** The main result of a Astrée is a list of alarms raised from the abstract invariant it computes. For each alarm, the user is informed about the program location at which the corresponding error could happen during real execution. Each alarm also comes with additional contextual information. Unfortunately, there is no simple and direct way to state whether an alarm is true or false. For each alarm, the user must look at the analysed program and analyse it backwards from the location at which the Run-Time Error might occur in a real execution. This user backward analysis aims at deciding whether it exists a run-time scenario causing the error or not.

By principle of Abstract Interpretation, if all the alarms raised by an analysis, i.e., one run of an analyser, are false, then the proof of absence of Run-Time Errors is completed. The problem here is that such a human analysis is error-prone and time-consuming.

Therefore, the fewer false alarms an Abstract Interpretation based static analyser like Astrée produces, the better for the safety of the demonstration that the analysed program cannot produce Run-Time Errors.

An analyser is said more precise than another one if it produces less false alarms for the same analysed program. The maximum precision is reached when all alarms, if any, are true.

2.1 Specialisation of Astrée

To achieve the above mentioned "zero false alarm" objective, an Abstract Interpretation based static analyser needs to be "specialised". Here, specialisation deals with the ability to capture[1] precise enough abstract invariants by taking into account the kind

[1] In the Abstract Interpretation framework, this capture is performed via so-called "abstract domains".

of computations performed by the analysed program. The "per program family specialisation" is performed at design time by the developers of the analyser. As explained in [1, §3.1], it is obtained by refinement of a previous more general-purpose analyser, making the new analyser more precise for any program of a targeted class of programs.

One example of the specialisation of Astrée has consisted in turning it able to precisely – and safely, of course – over-approximate the output stream of floating-point values produced by digital filters.

Specialisation to synchronous avionics programs. Such programs perform control/command operations and are those on which Astrée is the most precise, i.e., on which it produces the smallest number of false alarms. In avionics, these programs are automatically produced from SCADE™ (formerly SAO) detailed specifications and have very peculiar computational characteristics. The following four paragraphs describe the most important specialisation issues.

Synchronous programs. First, their scheduling is fully statically defined. It means that although they are made of parallel tasks, their serialisation is performed at design time, i.e., the execution of any piece of code only depends on the date at which it must execute. Such a date is given by the program's clock. Mainly for that reason, these programs are said to be "synchronous". There is no event driven task, no pre-emption, etc. For automatic analysis, the first important consequence in terms of precision, is that it is bounded by the maximum value of the clock, i.e., the maximum duration of an "aircraft mission". Consequently, the "clock abstract domain" was the first due to the specialisation.

Linear control flow – Intensive use of Booleans. Another characteristic of these automatically produced programs stands in their extremely linear code structure. As any code instruction belongs to a small library component, conditionals, i.e., "`if(){}else{};`", or loops, e.g., "`while() {};`" are necessarily very local and contain a very limited number of instructions. For the loops, the consequence is that they are quite easy to analyse precisely. With respect to the conditionals, the consequence of their extreme locality is that the program contains a lot of Booleans. Typical scheme is the computation of a Boolean value as the result of a condition on floating-point values, and then the test of this Boolean value elsewhere in the code for performing appropriate actions. In "classical" programming, this is done by testing the condition in an "`if(){}else{};`" statement and performing the actions in the scope of the first or second pair of "`{}`". For a static analyser, the two situations are not similar. Indeed, without dedicated "abstract domain", there is a severe loss of precision when the control flow is "encoded in Booleans", like in the first of the two situations described above. To avoid this kind of loss of precision, Astrée contains an Abstract domain called the "Boolean tree domain".

Floating-point calculus. Synchronous control/command avionics programs use a large number of floating-point variables (over 10,000): inputs, state variables and outputs. The operations performed for computing the state variables from inputs and the output variables from the state variables at each clock tick (every 10ms, for instance)

represent thousands of lines of C code and are subject to accumulating rounding errors. To be sound, Astrée abstracts each single floating-point operation in such a way that the computed set of values includes all possible rounding errors. It is a real challenge, also because the proper tool, i.e., Astrée, makes its own computations by using floating-point arithmetic.

Digital filters. As mentioned above, the control/command program computations are those induced by the control theory. Beyond the arithmetic operations, typical basic operators are delays, derivation, integration, first and second order digital filters. The very first versions of Astrée did not take into account the last two specificities. Consequently, the abstract domains existing at that time, e.g., intervals, clock, octagons, were not able to compute tight shapes of values for the outputs of the filters. As there are cascades of such filters, a lot of false overflows of floating-point values were raised by Astrée. The implementation of the Filter domain led to the suppression of all false alarms consecutive to the excessive over-approximation of the filter outputs.

Last step in specialisation: fine tuning by the user. In spite of its dedicated domains, Astrée might produce a few false alarms, which it is better to suppress by launching the tool again with a new set of "parameters", if possible.

In fact, after the user is convinced that a given alarm is false, next step is to understand why the analyser could not compute an invariant precise enough to avoid this false alarm. Since abstraction consists in not taking into account some characteristics (or properties) of the real executions of the program, it might be the case that none of the Astrée abstract domains is able to capture the program properties that would have avoided the false alarm. In this case, there is no other way than asking the Astrée team to improve the tool.

On the other hand, it might be that the domains able to capture the missing information are there, but their complexity is too high to be applied a priori uniformly to the whole program. In this case, some domains like the octagons or the domain of partitions can be locally used by means of Astrée directives to be inserted in the analysed program. This final "per program specialisation process" matches the adaptation by parameterisation described in [1, §3.2]. It is sound, provided the user only inputs facts about the environment of the program, and hints (to be checked by the tool) on how to improve the precision.

3 An Engineering Method to Handle Astrée Outputs

The first analysis of a complete real-world application usually floods the user with many alarms, so one hardly knows how to get started. Let us try to define a methodical approach to deal with these alarms.

3.1 The Need for Full Alarm Investigation

Exhaustive alarm analysis is absolutely necessary: every single alarm must either disappear by means of a more precise automated analysis, or be proved by the user to be impossible in the real environment of the program. Indeed, every time Astrée signals

an alarm, it assumes the execution of the analysed program to stop whenever the pre-condition of the alarm is true. As a consequence, any true alarm may "hide" more alarms. Let us give a simple example with variable `X` of type `int` in interval `[0,10]`:

```
Y = 1/X;
Z = 1/X;
```

Astrée will report a warning on line 1 (possible division by zero), but not on line 2: since the execution is assumed to stop whenever `X` happens to equal zero on line 1, line 2 cannot be executed unless $X \neq 0$. As a consequence, Astrée assumes `X` to be in interval `[1,10]` from line 2.

3.2 How to Read an Alarm Message

Here is an example of alarm reported by Astrée:

```
bnrvec.c:127.2-10::
    [call#APPLICATION_ENTRY@449:loop@466>=4:
     call#SEQ_C3_4_P@543:
     call#P_9_1_6_P@247:
     call#BNRVEC_P@442:]:
WARN: float->signed int conversion range [-6999258112,
6991659520] not included in [-2147483648, 2147483647]
```

Let us explain how this message is to be understood. Astrée warns that some `float->signed int` conversion may cause an integer overflow. It points precisely to the operation that may cause such a RTE: line 127 of the (pre-processed) file `bnrvec.c`, between columns 2 and 10[2]:

```
R3=E1*A3;
```

where R3 has type `int`, while `E1` and `A3` have type `float`.

All other information describe the stack. The analysis entry point[3] is the `APPLICATION_ENTRY` function, defined line 449 of file `appl_task.c`. This function contains a loop at line 466. From the fourth loop iteration, at least in the abstract semantics computed by the tool, there exists an execution trace such that :

- function `SEQ_C3_4_P` is called on line 543 of file `appl_task.c`;
- `SEQ_C3_4_P` calls function `P_9_1_6_P` on line 247 of file `seq_c3_4.c`;
- `P_9_1_6_P` calls function `BNRVEC_P on` line 442 of file `P9_1_6.c`;
- the floating-point product of the operands `E1` and `A3` ranges from `-6999258112` to `6991659520`.

[2] Line numbers start from 1, whereas column numbers start from 0.

[3] The user provides Astrée with an entry point for the analysis, by means of the `--exec-fn` option. Usually, this is the program's entry point.

However, this interval is not a subset of `[-2147483648, 2147483647]`, hence contains values that cannot fit into a 32 bit signed integer.

Of course, this does not necessarily mean there exists such an erroneous execution in the concrete semantics of the program: one is now to address this issue via a dedicated method.

3.3 How to Deal with Alarm Investigation

As explained above, every alarm message refers to a program location in the preprocessed code. It is usually useful to get back to the corresponding source code, to obtain readable context information.

To investigate an alarm, one makes use of the global invariant of the most external loop of the program, which is available in the Astrée log file (provided the `-dump-invariants` analysis option is set). Considering every (global) variable processed by the operation pointed to by an alarm, one may extract the corresponding interval, which is a sound over-approximation of the range of this variable.

Then, we have to go backwards in the program data-flow, in order to get to the roots of the alarm: either a bug or insufficient precision of the automated analysis. This activity can be quite time-consuming. However, it can be made easier for a control/command program that has been specified in some stream language such as SAO, SCADE™ or Simulink™. The engineering user can indeed label every arrow representing a global variable with an interval, going backwards from the alarm point. The origin of the problem is usually found when some abrupt unforeseen increase in variable ranges is detected.

At this point, we know whether the alarm originated in some "random code" or in some definite specialized operator (i.e., function or macro-function). Indeed, an efficient approach is first to concentrate on alarms in operators that are used frequently in the program, especially if several alarms with different stack contexts point to the same operators: such alarms will usually affect the analysis of the calling functions, thus raising more alarms.

Once we have probed into the roots of the alarm, we will usually need to extract a reduced example to analyse it. Therefore, we:

- write a small program containing the code at stake;
- build a new configuration file for this example, where the input variables `V` are declared `volatile` by means of the `__ASTREE_volatile_input((V [min, max]));` directive. The variable bounds are extracted from the global invariant computed by Astrée on the complete program;
- run Astrée on the reduced example (which takes far less time than on a complete program).

Such a process is not necessarily conservative in terms of RTE detection. Indeed, as the abstract operators implemented in Astrée are not monotonic, an alarm raised when analysing the complete program may not be raised when analysing the reduced example. In this case, this suggests (though does not prove) that the alarm under investigation is probably false, or that the reduced example is not an actual slice of the complete program with respect to the program point pointed to by the alarm.

However, this hardly ever happens in practice: every alarm raised on the complete program will usually be raised on the reduced example as well. Besides, it is much easier to experiment with the reduced example:

- adding directives in the source, to help Astrée increase the precision of its analysis;
- tuning the list of analysis options;
- changing the parameters of the example to better understand the cause of the alarm.

Once a satisfactory solution has been found on reduced examples, it is re-injected into the analysis of the complete program: in most cases, the number of alarms decreases.

4 Verifying a Safety-Critical Control/Command Program with Astrée

We will now present experiments realised on a periodic synchronous control/command program developed at Airbus. Most of its C source code was generated automatically from a higher-level synchronous data-flow specification. Most generated C functions are essentially sequences of calls of macro-functions coded by hand. Like in [1, §4], it has the following overall form:

```
declare volatile input, state and output variables;
initialise state variables;
loop forever
    read volatile input variables,
    compute output and state variables,
    write to volatile output variables;
    wait for next clock tick;
end loop
```

This program is composed of about 200,000 lines of (pre-processed) C code processing over 10,000 global variables. Its control-flow depends on many state variables. It performs massive floating-point computations and contains digital filters.

Although an upper bound of the number of iterations of the main loop is provided by the user, all these features make precise automatic analysis (taking rounding errors into account) a great challenge. A general-purpose analyser would not be suitable.

Fortunately, Astrée has been specialised to deal with this type of programs: only the last step in specialisation (fine tuning by the user) has to be carried out. The automated analyses are being run on a 2.6 GHz, 16 Gb RAM PC. Each analysis of the complete program takes about 6 hours.

The very first analysis produces 467 alarms. With this program, after options related to loop unrolling and widening parameters have been tuned, we get 327 remaining alarms to be further analysed by the industrial user, in order to decide whether they are false alarms or not.

4.1 Analyzing a False Alarm

For instance, in this program, many calling contexts of the widely used linear two-variable interpolation function G_P give rise to alarms within the source code of this function. Here are two examples:

```
g.c:191.8-55::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C1_P@49
8:call#P_2_7_1_P@360:call#G_P@967:if@119=false:if@124=f
alse:loop@132=2:if@152=false:if@156=false:loop@164=2:]:

WARN: float arithmetic range ([-inf, inf]

g.c:191.8-55::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C3_3_P@
522:call#P_2_7_4_P@202:call#G_P@694:if@119=false:if@124
=false:loop@132=3:if@152=false:if@156=false:loop@164=2:
]:

WARN: float arithmetic range ([-inf, inf]
```

To understand the problem and be able to tune the analysis parameters, one is to build a reduced example from one of these contexts, say `P_2_7_1_P`. The following code is being extracted from the original `P_2_7_1_P` function:

```
void P_2_7_1_P () {

    PADN13 = fabs(DQM);

    PADN12 = fabs(PHI1F);

    X271Z14 = G_P(PADN13, PADN12, G_50Z_C1, G_50Z_C2, &
    G_50Z_C3 [0][0], & G_50Z_C4 [0][0], ((sizeof(
    G_50Z_C1 )/sizeof(float))-1), (sizeof( G_50Z_C2
    )/sizeof(float))-1);

}
```

where:

- `fabs` returns the module of a floating-point number;
- `DQM`, `PHI1F`, `PADN13`, `PADN12` and X271Z14 are floating-point numbers;
- `G_50Z_C1`, `G_50Z_C2`, `G_50Z_C3` and `G_50Z_C4` are constant interpolation tables.

`DQM` and `PHI1F` are declared as volatile inputs in the analysis configuration file. Their ranges are extracted from the global invariant computed by Astrée on the full program:

```
DQM in [-37.5559, 37.5559]

PHI1F in [-199.22, 199.22]
```

On this reduced example, we get the same alarms as on the full program. All of them suspect an overflow or a division by zero in the last instruction of the G_P function:

```
return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);
```

However, reading the code of G_P, one notices that G2=R3+1 always holds at this point. Moreover, in this reduced example, the interpolation table G_50Z_C2 is such that G_50Z_C2[i+1]-G_50Z_C2[i]>1 for any index i. Hence, these alarms are false alarms; we must now tune the analysis to get rid of them.

To do so, we have to make Astrée perform a separate analysis for every possible value of R3, so it can check no RTE can possibly happen on this code. The way to do it, is to ask for a local partitioning on R3 values between:

- the first program point after which R3 is no longer written;
- the first program point after which R3 is no longer read.

Let us implement this, using Astrée partitioning directives:

```
__ASTREE_partition_begin((R3));

G2=R3+1;

Z1=(X1-C1[R2])*(*(C4+(TAILLE_X)*R3+R2)) +
(*(C3+(TAILLE_X+1)*R3+R2));

Z2=(X1-C1[R2])*(*(C4+(TAILLE_X)*G2+R2)) +
(*(C3+(TAILLE_X+1)*G2+R2));

return(Z2*(Y2-C2[R3])+Z1*(C2[G2]-Y2))/(C2[G2]-C2[R3]);

__ASTREE_partition_merge(());
```

This hint makes the alarms disappear on the reduced example. It has the same effect on the complete program. Besides, many alarms depending directly or indirectly on variables written after a call of function G_P disappear as well: the overall number of alarms boils down to 11.

4.2 Analysing a True Alarm

Let us describe one of the (few) remaining alarms:

```
P8_4_1.c:506.0-38::

[call#APPLICATION_ENTRY@449:loop@466=1:call#SEQ_C4_7_P@
609:call#P_8_4_1_P@152:]:

WARN: implicit unsigned int->signed int conversion
range {3090427953} not included in [-2147483648,
2147483647]
```

This alarm points to the following (pre-processed) instruction:

```
MODULE_NUMBER= 0x80000000 + 0x38343031 ;
```

where MODULE_NUMBER has type int.

This code is fairly straightforward, in the sense that its execution does not depend on any input. Such a context allows for a very precise analysis, which is why Astrée does not report an interval, but a single possible value for the result of the addition (`3090427953`).

The reason for this alarm is the following: the ISO/IEC 9899 international standard, describing the semantics of integer constants, specifies the type of an integer constant to be the first of the corresponding list in which its value can be represented. For a hexadecimal constant without a suffix, the list is:

1. int
2. unsigned int
3. long int
4. unsigned long int
5. unsigned long long int
6. long long int

As `0x80000000` = 2^{31} does not fit into a 32 bit `int`, this constant must have type `unsigned int`. `0x38343031>0`, hence the result of the addition has type `unsigned int` and its value (`3090427953`) is outside the range of signed 32 bit integers, i.e. `[-2147483648, 2147483647]`.

Most compilers will know how to deal with such a conversion, however, Astrée soundly and genuinely warns that the semantics of this code is unspecified by ISO/IEC 9899. Indeed, its behaviour is implementation-defined.

One way to fix it (without using type `long long int`, which Astrée does not support) is to replace `0x80000000` by `(-2147483647-1)`, which is probably what the programmer had in mind in the first place.

4.3 Results

On this control/command program, it has been possible for a non-expert user from industry to reduce the number of false alarms down to zero.

5 Conclusion and Future Work

Several theoretical papers, such as [1] and [2], already explained the solutions to the scientific and technological issues which the Astrée team had to deal with to demonstrate that an Abstract Interpretation based static analyser, Astrée, can analyse real-size industrial programs and be extremely precise.

But it was a point of view of scientists.

In our paper, we have tried to present an industrial point of view. The challenge for software engineers from industry was to be able to use the analyser Astrée on a real-size program without altering it before the analysis. We have succeeded, as shown in the previous sections, thanks to two kinds of skills. First, being software engineers, we were not afraid of reading code. Secondly, we have had the confirmation that, to use an Abstract Interpretation based static analyser, it is mandatory to know enough about its underlying principles, and that it was within the reach of software engineers.

A proof that a safety-critical avionics program is free from Run-Time Errors is obviously not a proof that this program is safe. Although we have not described it in this paper, we are also addressing complementary safety objectives such as WCET assessment [6], safe memory use, precision and stability of floating-point computations [7]. To meet all these objectives is a first step towards the safety assessment of a program.

Other future work will address the proof of absence of RTE on multi-threaded asynchronous programs.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN'2003 Conf. PLDI, San Diego, CA, US, 7–14 June 2003, pp. 196–207. ACM Press, New York (2003)
2. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRE'E analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
3. Cousot, P.: Interprétation abstraite. Technique et Science Informatique 19(1-2-3), 155–164 (2000)
4. Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. In: Wilhelm, R. (ed.) Informatics. LNCS, vol. 2000, pp. 138–156. Springer, Heidelberg (2001)
5. Cousot, P., Cousot, R.: Basic Concepts of Abstract Interpretation. In: Jacquard, R. (ed.) Building the Information Society, pp. 359–366. Kluwer Academic Publishers, Dordrecht (2004)
6. Souyris, J., Le Pavec, E., Himbert, G., Jégu, V., Borios, G., Heckmann, R.: Computing the worst case execution time of an avionics program by abstract interpretation. In: Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pp. 21–24 (2005)
7. Goubault, E., Martel, M., Putot, S.: Static Analysis-Based Validation of Floating-Point Computations. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Numerical Software with Result Verification. LNCS, vol. 2991, pp. 306–313. Springer, Heidelberg (2004)

Detection of Runtime Errors in MISRA C Programs: A Deductive Approach

Ajith K. John[1], Babita Sharma[1], A.K. Bhattacharjee[1], S.D. Dhodapkar[1,⋆], and S. Ramesh[2,⋆⋆]

[1] Software Reliability Section, Bhabha Atomic Research Centre, Mumbai 400085, India
{ajith,babita,anup,sdd}@barc.gov.in
[2] Centre for Formal Design and Verification of Software, Indian Institute of Technology, Bombay, Mumbai 400076, India
ramesh@cse.iitb.ac.in

Abstract. In this paper, we describe a method for detecting runtime errors for programs which are written in an industrially sponsored safe subset of C called MISRA C. The method is based on a novel model of C programs: each C program is modeled as a typed transition system encoded in the specification language accepted by PVS theorem prover. Since the specification is strongly typed, proof obligations are generated, for possible type violations in each statement in C, when loaded in the PVS theorem prover which need to be discharged. The technique does not require execution of the program to be analysed and is capable of detecting runtime errors such as array bound errors, divide by zero, arithmetic overflows and underflows etc. Based upon the method, we have developed a tool, which converts MISRA C programs into PVS specifications automatically. The tool has been used in checking runtime errors in several programs developed for real-time control applications.

1 Introduction

Ensuring the absence of errors in software used in safety-critical systems is extremely important as the failure of such software can lead to system failure causing a loss of safety function. *Run Time Errors (RTEs)* are the most subtle but crucial type of errors found in software leading to system failures. Examples of such errors are array indices out of bound, divide by zero and arithmetic overflows/underflows etc. This paper describes our work in developing techniques for detection of such RTEs using a deductive approach.

Runtime errors can be detected using testing but testing cannot guarantee the absence of such errors because of the very large number of test cases required. Detecting run time errors statically through program analysis, helps in reducing efforts in activities like Debugging, White box testing and Code reviews.

⋆ Corresponding author.
⋆⋆ Presently with GM Research Lab, Bangalore.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 491–504, 2007.

Static Analysis tools like SPLint [7] are very useful in detecting RTEs related to buffer overflow and pointer arithmetic but they do not address RTEs associated with arithmetic expressions. The program supervision tools e.g. those built using Valgrind [12] require target program execution and suffer from the same limitations as that of testing. Semantic Analysis based on Abstract Interpretation [4] is a more rigorous approach to detect runtime errors statically. This technique is based on data flow analysis which computes program properties by converting programs into equations over datatypes represented as lattices and then solving these equations over these lattices. Tools such as PolySpace [14], ASTREÉ [6] use the abstract interpretation technique for the detection of runtime errors in C programs. However these tools work on an abstract model (sound but not complete) of a program and are prone to *false positives* which are possible errors requiring further effort to investigate. Commercial tools also do not allow tuning to reduce false positives. The tool ASTREÉ was designed to work on a specific class of C programs called synchronous C programs [6], and is being extended to general class of C programs. There are also excellent works on software model checking like BLAST [8], SLAM [1] etc., which are used for checking specifications related to software interfaces, shared resources in device drivers etc. These tools model check specified property and do not specifically address the issues of arithmetic runtime errors in a global manner.

In this paper, we explain an alternative approach for detecting run time errors based on Type System [3] and describe an in-house tool developed based on this approach. The technique is based upon modeling C programs as state transition systems encoded in the specification language of Prototype Verification System (PVS) [13]. *State Transition System* model of a program describes how variables in the program are modified as program executes from beginning to end. The description of a given program as state transition system in PVS language is called *PVS Specification* or *PVS Model* of the program. Since the specification language of PVS is strongly typed, the possible runtime errors in the C program result into type inconsistencies in the PVS specification. When the PVS specification is typechecked, these type inconsistencies automatically generate proof obligations called Type Correctness Conditions (TCCs). The PVS prover commands can be used for discharging the proofs of these TCCs. If all the TCCs are proved, the program can be declared as free of type violations and therefore corresponding runtime errors. Presence of any unproved TCC indicates that the program is not typesafe and it can be traced back to a possible runtime error in the C program. Based on this method, we have developed a tool, which is currently capable to detect array bound errors, divide by zero, arithmetic overflows/underflows and illegal values for function parameters. This is a deductive technique and requires user interaction but can handle infinite state systems. The proof process is automated to some extent using PVS strategies as scripts. However this approach of checking properties cannot be fully automated as it sometimes requires powerful theorem proving techniques that can be applied only interactively. Although the technique can be applied to general class of C programs, it is specifically

targeted towards safety-critical software developed following software engineering practices like controlling module sizing, complexity(McCabe) etc.

The rest of the paper is organized as follows. Section 2 gives an overview of the method. In section 3, we describe how C programs are modeled in PVS. A brief account on how TCCs are proved in PVS and how to locate the RTE is given in section 4. Section 5 presents a small example to illustrate the method. Section 6 summarises our experience and provides some discussion related to future work.

2 Overview

The method is based on modeling the C programs as state transition systems in PVS specification language. The states are encoded as the evaluation of the program variables over the data domains. The transitions are the execution of statements and modeled as the effect of modifying the set of state variables participating in the statement. Each state is expressed as a function of the previous state and thereby the entire program is encoded as a list of states each in terms of the previous states.

The datatypes of C (which are represented in the machine in finite size) are modeled in PVS as subtypes of PVS types with limited size. For example *int* datatype of C is represented as *restr_int* which is a subtype of *int* datatype of PVS restricted between the integer maximum and integer minimum. Each operator in C is modeled as a PVS function with operands having subtypes and return value having PVS type. For example, the binary + operator on *int* datatype of C is represented as a PVS function *add_restr_int* with *restr_int* operands and *int* return value. When + operator is used in any expression in the C program, it is replaced by the function *add_restr_int* in the PVS model. This modeling scheme causes the typechecker to generate TCCs for all possible runtime errors. This is explained in detail in section 3.

Most of the software development process standards like IEC [10] used in developing software for safety-critical applications recommend use of programming rules which prohibit usages of unsafe constructs of a programming language. MISRA C:2004 [11] standard is a set of programming rules for C language categorized in *Required* and *Advisory* rules. These set of rules together define a subset of C Language. We have adopted this subset of C as it is a well accepted standard and one can check compliance to MISRA standard using commercial tools. Use of MISRA C also helps in meeting recommendations of standards for safe subsets of languages. Limiting the method to MISRA C subset which does not allow union datatype, pointer arithmetic and dynamic memory allocation eliminates difficulties with PVS model generation and reasoning. These features (allowed in general POSIX standard) are considered unsafe in context of safety critical software.

Fig. 1, shows the steps that are followed in carrying out runtime error detection. The source code compliant to the MISRA C standard is annotated with formal annotations extracted from the constraints on the input data of the

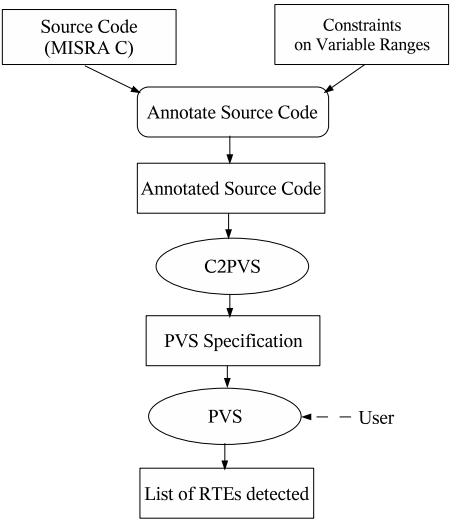


Fig. 1. Runtime Error Detection Process

program. These constraints are referred in the Fig. 1 as *Data Range Specifications*. The translator *c2pvs*, developed as part of this work, translates the annotated MISRA C program to a *PVS specification* which is loaded in the PVS theorem prover. The type inconsistencies in the PVS specification are detected by the PVS typechecker causing generation of TCCs (Type Correctness Conditions), which need to be proved before the specification can be considered typesafe. An unprovable TCC shows that the program is not typesafe and indicates presence of a runtime error.

The annotations are written as comments in a predefined syntax, so that they will be ignored by the compiler and the behavior of the source code will not be changed. The annotations are translated by *c2pvs* to axioms or lemmas which are used in the proofs. We explain the method with the help of the example shown in Fig.2, where the value of `z` is computed based on the parameters `a` and `b`.

There are two parameters a and b for the given function. Let the ranges of values a and b can take at the entry to the function be,

```
1500 <= a <= 2000
500 <= b <= 1000
```

The function `find_z` can be annotated with these constraints using annotations of type *pre*-conditions denoted as *pre*. The annotated function is shown in Fig.3. It can be seen that there is a divide by zero error in the statement $z = (a * 4000)/(p - b)$; at $l2$ (shown by an arrow) as the expression $p - b$ evaluates to zero at this program point (Note that the statement $z = (a*1000)/(p-b)$;

```
extern int ext_glob;
int find_z(int a,int b)
{
    int p,z;
    if(ext_glob>0)
        p=a;
    else
        p=b;
    if(ext_glob>0)
        l1: z=(a*1000)/(p-b);
    else
        l2: z=(a*4000)/(p-b);
    return z;
}
```

Fig. 2. Example Program

```
extern int ext_glob;
int find_z(int a,int b)
{
    int p,z;
    /*pre a≥1500 AND a≤2000
    AND b ≥500 AND b ≤ 1000 end*/
    if(ext_glob>0)
        p=a;
    else
        p=b;
    if(ext_glob>0)
        l1: z=(a*1000)/(p-b);
    else
        l2: z=(a*4000)/(p-b);        ←
    return z;
}
```

Fig. 3. Annotated Program

at $l1$ does not cause divide by zero error). The actual steps in the proof process are further explained below.

As shown in Fig.1, the annotated function is given as input to *c2pvs*. The *c2pvs* translator translates the function to a state transition system encoded as PVS specification. The *pre*-condition is translated as an axiom in the PVS specification. The PVS specification generated by *c2pvs* is then loaded in PVS and typechecked. The PVS typechecker generates the following TCC to guarantee that $p - b$ is not zero at $l2$ (*sub_restr_int* is the PVS representation of the binary $-$ operator in C).

```
OBLIGATION sub_restr_int(p, b) /= 0
```

The proof of this TCC is tried using the *pre*-condition and the PVS prover commands. The proof fails, which indicates that $p - b$ is zero at $l2$ and the statement $z = (a * 4000)/(p - b)$ can lead to a divide by zero error. A similar TCC is generated at $l1$ for the statement $z = (a * 1000)/(p - b)$ also; but it gets proved indicating that the statement does not result into division by zero.

The annotations of type *pre*-conditions used here specify the ranges of values function parameters and global variables can take at the entry to a function. In general *pre*-conditions can be used to specify the constraints on the input data (ranges of input data) and any condition which is true at the entry to the function. It must be noted that these annotations are only used to capture the ranges of the input data and do not capture the functional specifications of each C function. Tools like PolySpace use similar *Data Range Specifications* for global variables. The different types of annotations supported by the tool and their syntax are listed in table 1.

Table 1. Formal annotations and their syntax

Annotation Syntax	Meaning
/∗ *pre* **formula** *end* ∗/	*Conditions true at entry to function*
/∗ *post* **formula** *end* ∗/	*Conditions true at exit of function*
/∗ *postfunc* **formula** *end* ∗/	*Model effect of function calls*
/∗ *prefunc* **formula** *end* ∗/	*Conditions true before function calls*
/∗ *loopinv* **formula** *end* ∗/	*Loop invariants*

Generating PVS specification of a C program involves modeling the datatypes and modeling the execution semantics. The next section describes the modeling scheme in detail.

The method can be used for runtime error detection of sequential C programs at the unit level (C function level). However, to guarantee the absence of runtime errors across the entire program, one can use compositional technique.

3 Modeling C Programs in PVS

3.1 Modeling Datatypes of C in PVS

The PVS type system has *number, real, rational, integer, and natural* as number related types [13]. The variables of these types can have values varying from -∞ to +∞ (except natural where the range is 0 to +∞). Unlike PVS, in C the ranges of values for datatypes are restricted. Hence we have modeled the C datatypes as restricted types (subtypes) in PVS.

Modeling Integer and Character Datatypes: Integer and character datatypes of C are modeled as subtypes of the PVS datatype *int*, restricted between the maximum and minimum representable values. For example, *int* datatype of C is modeled as a subtype *restr_int*.

```
restr_int:TYPE =
    {x:int| x<=INT_MAX AND x>=INT_MIN AND INT_MAX>=INT_MIN}
```

where INT_MAX and INT_MIN are constants indicating the integer maximum and integer minimum respectively for the machine on which the C program will be executed. Similar modeling is used for other integer types and character types.

Operators on integer and character datatypes are modeled as functions with subtype arguments and PVS type return value. For example, arithmetic operators on *int* are modeled as functions with *restr_int* arguments and *int* return value. Binary + on *int* is modeled as a function *add_restr_int*.

```
add_restr_int(x:restr_int,y:restr_int):int=x+y
```

Similar modeling is used for other operators on integer and character datatypes.

Modeling Floating Point Datatypes: Floating point datatypes of C (*float*, *double*, and *long double*) are modeled as subtypes of PVS datatype *real* restricted to the normalized range [9].

For example, *float* datatype of C is modeled as a subtype *restr_float*.

```
restr_float:TYPE={x:real|((x <= MAX AND x >= MIN) OR
    (x <= -1*MIN AND x >= -1*MAX) OR (x = 0)) AND (MAX >= MIN)}
```

Here MAX and MIN are constants indicating the normalized float maximum and normalized float minimum respectively.

Operators on floating point datatypes are also modeled as functions with subtype arguments and PVS type return value. But these functions modeling the floating point operators first perform the operations with infinite precision and then the result is converted to finite precision. Hence three rounding functions *round_to_nearest_float*, *round_to_nearest_double* and *round_to_nearest_long_double* are defined, which round an infinite precision real number to the nearest representable float, double and long double number respectively. For example, binary + operator on *float* is modeled as a function *add_restr_float*.

```
add_restr_float(x:restr_float, y:restr_float): real
    = round_to_nearest_float(x+y)
```

Similar modeling is used for other operators on floating point datatypes.

Generation of TCCs: The modeling scheme described above causes the typechecker to generate proper TCCs for all possible runtime errors. We will illustrate this with the help of two simple examples.

Let us consider the statement $c = a + b$; where a, b, c are *int*. The value of $a + b$ can go beyond INT_MAX or INT_MIN which can get assigned to c causing an integer overflow/underflow. Let us see how the typechecker detects this.

The statement $c = a + b$ is modeled in PVS as,

```
c = add_restr_int(a,b)
```

The typechecker finds that the type of c is *restr_int* which does not match with the type of *add_restr_int(a,b)*. Hence it generates the following TCC to ensure that the value of *add_restr_int(a,b)* does not go beyond INT_MAX or INT_MIN so that it is of type *restr_int*.

```
add_restr_int(a,b)<=INT_MAX AND add_restr_int(a,b)>=INT_MIN
```

Proving this TCC ensures that $a + b$ does not overflow/underflow in this statement.

Let us consider another statement $c = a/b$; where a, b, c are integers. There are two possible runtime errors in the statement.

1. b can be zero resulting in *divide by zero* error.
2. a/b can *overflow/underflow*.

The statement is encoded in PVS as,

```
c = div_restr_int(a,b)
```

The typechecker expects *restr_int* and *nonzero_restr_int* (*restr_int* with zero excluded) as the argument types for *div_restr_int*. But it encounters *restr_int* as the second argument of *div_restr_int*. Similarly it expects *restr_int* as the type

of *div_restr_int* and encounters *int* instead of it. The typechecker generates two TCCs.

1. `b /= 0`
   This TCC is to ensure that b is not zero.
2. `div_restr_int(a,b)<=INT_MAX AND div_restr_int(a,b)>=INT_MIN`
   This TCC is to ensure that the operation a/b does not overflow/underflow.

Proving these TCCs ensures that b is not zero a/b does not overflow/underflow in this statement.

3.2 Modeling Execution Semantics of C in PVS

In our method, a C function is modeled as a state transition system, encoded as a PVS theory (A PVS specification is composed of theories similar to the way a C program is composed of functions). In the state transition system considered here, a state is a type consistent valuation of all the program variables and statements are the transitions between the type consistent states. A state is type consistent if the valuations of state variables are within proper ranges defined for that type. We encode the state transition system as a list of such states with each state defined in terms of the previous states. i.e. a statement S causing the $s_i \rightarrow s_j$ transition is represented by s_j where s_j is defined as a function of s_i. A program is type safe if all the reachable states are type consistent.

Modeling of State: State is modeled in PVS by a tuple of values. Each component of the tuple corresponds to a variable of the program. In general consider a C program with n variables $v_1, v_2, \ldots, v_i, \ldots, v_n$ of types $t_1, t_2, \ldots, t_i, \ldots, t_n$. The states of the program are modeled as tuples of type $[t_1, t_2, \ldots, t_i, \ldots, t_n]$. For example a state s_a is modeled as

$$s_a : state = (s_a`1, s_a`2, \ldots, s_a`i, \ldots, s_a`n)$$

Now the components of the tuple $s_a`1, s_a`2, \ldots, s_a`i, \ldots, s_a`n$ denote the values of the program variables $v_1, v_2, \ldots, v_i, \ldots, v_n$ respectively at the state s_a.

Assignment Statement: An assignment statement modifies the value of a program variable and thus changes the state of the program. It can therefore be modeled in PVS as a new state resulting from the application of a state transition on the present state.

Sequential Composition: Consider a sequence of statements $S_1; S_2$; Let the current state of the program be s_a. Let the statement S_1 changes the program state to s_{a+1}. The statement S_2 now acts on the state s_{a+1}, to get s_{a+2} as the resulting state. We represent this in PVS as

$$s_{a+1} : state = state\ transition\ for\ S_1(s_a)$$
$$s_{a+2} : state = state\ transition\ for\ S_2(s_{a+1})$$

The statements after S_2 will act on the state s_{a+2} and are represented in the same manner as S_1 and S_2.

If-else Statement: Consider an if-else statement which occurs at a program state s_a as shown in the table 2. After the if-else statement at state s_a, the program can go to two possible states s_b (if the condition is true) or s_d (if the condition is false). The statements inside the if part will operate on the state s_b and those inside the else part will operate on s_d. Let s_c be the final state in the if part and s_e the final state in the else part. The states s_b to s_c and s_d to s_e are not reachable always and their reachability depends on the if condition. The state after the whole if-else statement s_f will be either s_c or s_e. Hence modeling the if-else statement includes modeling of all the states from s_b to s_f.

Loops: Consider the loop `while(B) S` , where B is the loop condition and S is the statement part of the loop. We take loop invariants as input from the user (A loop invariant is defined as a formula which is true before control enters the loop, remains true each time the program executes the body of the loop, and is still true when control exits the loop). Now the state after the loop can be modeled as any state satisfying the condition:

```
(loop invariant) AND NOT(loop condition)
```

The accuracy of the proofs depends on the correctness and accuracy of the user supplied invariants. Tools like STeP [2], which can generate loop invariants automatically can be used for this purpose.

The possible runtime errors inside the loop are detected by handling the statements inside the loop separately i.e. by putting the translation of these statements into another theory. This theory is named as *looptheory_i* for the i^{th} loop in the function. As the loop invariant conjuncted with loop condition is true before entry to the loop body, it is put as a *pre*-condition to these statements. Typechecking this theory generates TCCs representing possible runtime errors inside the loop, the proofs of which are tried by using the *pre*-condition. Thus a C function with n loops will generate a PVS specification with $n + 1$ theories i.e. n loop theories and one main theory.

The PVS transformations of different constructs in a C program with n variables $v_1, v_2, \ldots, v_i, \ldots, v_j, \ldots, v_n$ of types $t_1, t_2, \ldots, t_i, \ldots, t_j, \ldots, t_n$ respectively are illustrated in detail in the table 2. All the constructs are labeled with identifiers indicating the states at which they occur.

4 Proving TCCs in PVS and Tracing RTEs to Source Code

Most of the TCCs generated in PVS can be proved by using the axioms in the PVS theory, type predicates on the program variables and type predicates on the loop states followed by the application of prover command *grind*. The tool automates the proofs of such TCCs by generating a strategy which performs the above steps. Such a strategy is generated for each theory.

Table 2. The C constructs and their PVS transformations

C construct	PVS transformation
$s_a : v_i = expr;$	$s_{a+1} : state = (s_a`1, s_a`2, \ldots, expr, \ldots, s_a`j, \ldots, s_a`n)$
$s_a : v_i = expr;$ $v_j = expr;$	$s_{a+1} : state = (s_a`1, sa`2, \ldots, expr, \ldots, s_a`j, \ldots, s_a`n)$ $s_{a+2} : state = (s_{a+1}`1, s_{a+1}`2, \ldots, s_{a+1}`i, \ldots, expr, \ldots, s_{a+1}`n)$

$s_a : if(cond)$
{
$s_b : statements$
$s_c :$
}
$else$
{
$s_d : statements$
$s_e :$
}
$s_f :$
	
	$s_b : state = if\ cond(s_a) = true\ then\ s_a\ else\ unreachable\ endif$
	$s_c : state = state\ transition\ for\ last\ statement\ in\ if\ part$
	$(state\ previous\ to\ s_c\ in\ if\ part)$
	
	$s_d : state = if\ cond(s_a) = false\ then\ s_a\ else\ unreachable\ endif$
	$s_e : state = state\ transition\ for\ last\ statement\ in\ else\ part$
	$(state\ previous\ to\ s_e\ in\ else\ part)$
	$s_f : state = if\ reachable(s_c)\ then\ s_c\ else\ s_e\ endif$

$s_a : while(B)\ S$ $s_b :$	 $substate : type = \{s : state \mid loop\ invariant(s)\ and\ not(B(s))\}$ $s_b : substate$

But there exists TCCs which cannot be proved this way; particularly those related to multiplication, and division. This is due to the reason that *grind* cannot handle nonlinear arithmetic. For example, let us consider that we want to prove in PVS the following

```
{-1}  a >= 0
  |-------
{1}   a * a >= 0
```

grind alone will not prove this. We need to use a lemma *le_times_le_pos* from the prelude (prelude consists of theories that are built into the PVS system), properly instantiate it and then invoke grind. The need to select and use lemmas from prelude and instantiating them makes the automatic discharge of such TCCs difficult. Such TCCs should be tried interactively.

Presence of any unproved TCC indicates the presence of possible RTEs in the original C program. Each TCC generated by PVS is identified by the state and the operation from which it is generated. The name of a TCC $s_i\_TCC_j$ indicates that it is generated from the j^{th} operation of the i^{th} state in the PVS model. As we know the number of states generated by each type of C statement, given that a TCC $s_i\_TCC_j$ is unproved we can infer the C statement causing the possible RTE from the state number i. The exact operation causing the RTE can be inferred from the operation number j. The type of RTE detected (array bound errors, divide by zero etc.) can be found out by analyzing the TCC description generated by PVS. Currently this functionality of tracing the RTEs from the unproved TCCs is not incorporated into the tool and effort is on to automate it.

5 Example

We now present an example to illustrate the method.

```
int average(int n)
{
     int i,sum,avg;
     /*pre n>=1 AND n<=100 end*/
     i=n;
     sum=0;
     while(i>0)
     {
      /*loopinv (i>=0) AND (i<=n) AND (n>=1) AND (n<=100) AND
      (sum = (n*(n+1)-(i+1)*i)/2) end*/
      sum=sum+i;
      i=i-1;
      }
      avg=sum/n;
      return(avg);
}
```

The function *average* finds out the average of first n natural numbers. The user supplied *pre-* condition and the loop invariant are inserted as formal annotations. The PVS specification generated for the function is shown below.

```
looptheory_1:THEORY
BEGIN
 ...
state:TYPE=[bool,void,restr_int,restr_int,restr_int,restr_int,restr_int]
s1:state
axiom_3: AXIOM  (s1‘5>=0) AND (s1‘5<=s1‘4) AND (s1‘4>=1) AND
    (s1‘4<=100) AND (s1‘6=(s1‘4*(s1‘4+1)-(s1‘5+1)*s1‘5)/2) AND
    great_restr_int(s1‘5,0)
   ...
s3:state=IF s2‘1=FALSE THEN s2 ELSE
    (s2‘1,s2‘2,s2‘3,s2‘4,sub_restr_int(s2‘5,1),s2‘6,s2‘7) ENDIF
END looptheory_1

average:THEORY
BEGIN
.....
state:TYPE=[bool,void,restr_int,restr_int,restr_int,restr_int,restr_int]
s1:state
axiom_3: AXIOM  s1‘4>=1 AND s1‘4<=100
.....
s3:state=IF s2‘1=FALSE THEN s2 ELSE (s2‘1,s2‘2,s2‘3,s2‘4,s2‘5,0,s2‘7)
    ENDIF
loopcondition_1:[state->bool]=LAMBDA(u:state): (great_restr_int(u‘5,0))
loopinvariant_1:[state->bool]=(LAMBDA(u:state): ( (u‘1=s3‘1) AND
    (u‘2=s3‘2) AND (u‘3 =s3‘3) AND (u‘4=s3‘4) AND (u‘7= s3‘7) AND
    (u‘5>=0) AND (u‘5<=u‘4) AND (u‘4>=1) AND (u‘4<=100) AND
```

```
    (u‘6=(u‘4*(u‘4+1)-(u‘5+1)*u‘5)/2)))
substate_1:TYPE={s:state | loopinvariant_1(s) AND
    NOT(loopcondition_1(s))}
axiom_4: AXIOM EXISTS (x1: substate_1): TRUE
s4:substate_1
 .....
s6:state=IF s5‘1=FALSE THEN s5 ELSE
    (s5‘1,s5‘2,s5‘3,s5‘4,s5‘5,s5‘6,s5‘6) ENDIF
END average
```

The PVS specification consists of two theories. The first theory *looptheory_1* is the translation of the loop and the second is the translation of the function *average*. The statements inside the loop are translated to states s_1 to s_3 in *looptheory_1*. *loop invariant AND loop condition* is translated to axiom *axiom_3* in *looptheory_1*. In theory *average*, the *pre*-condition is translated to *axiom_3* and the loop invariant is translated to [*state* → *bool*] function *loopinvariant_1*. s_4 is the state after the loop defined as a constant of the type *substate_1* which is a subtype of the *state* type satisfying *loopinvariant AND NOT(loop condition)*.

The tool generates strategy *looptheory_1_strategy* for the theory *looptheory_1* and *average_strategy* for the theory *average*. The PVS specification is type checked. Table 3 shows the TCCs generated, their significance and how they are proved. All the TCCs generated are proved. Hence there are no *arithmetic overflows/underflows* and *divide by zero* in the function *average*.

Table 3. The TCCs, their significance and proofs for function average

TCC Name	Theory Name	Significance of the TCC	How the TCC is proved
$s2\_TCC1$	*looptheory_1*	*To ensure sum + i does not overflow/underflow inside the loop*	*Interactively*
$s3\_TCC1$	*looptheory_1*	*To ensure i − 1 does not overflow/underflow inside the loop*	(*looptheory_1_strategy*)
$s5\_TCC1$	*average*	*To ensure n is nonzero in operation sum/n*	(*average_strategy*)
$s5\_TCC2$	*average*	*To ensure sum/n does not overflow/underflow*	*Interactively*

6 Conclusion and Future Work

In this paper, we have presented an approach for detecting run time errors in MISRA C programs by transforming the input programs into typed PVS specifications and subsequently discharging TCCs using PVS. The method is capable of detecting RTEs such as array out of bound, divide by zero, overflow/underflow. With tools based on abstract interpretation resolving false positives requires manual review of the code. In the proposed approach, an unproved TCC has a

direct correlation with the source line and it is easy to track the cause of the RTE in the source code. Also the loop invariants can be tightened to reduce the false positives.

The tool was initially validated with a number of examples. Table 4 gives an abstract of these examples as a list with the RTEs detected. The names are indicative of the operation performed by each function. These examples were also checked using PolySpace.

The tool was used for checking a library of function blocks for runtime errors. This function block library was used in *SCHEMER* (an Integrated Development Environment (IDE) for Function Block Diagram (FBD) based Programmable Controllers). *SCHEMER* contains a code generator that generates C code from the Function Blocks based process logics diagram. *SCHEMER* is being used for developing process logics and controls in safety critical applications of nuclear reactors. The constraints on the function parameters were extracted from the specifications in the User Requirements Document (URD) and were inserted into the code as annotations of type *pre*-conditions. The annotated code was given as input to the tool which checked for any possible runtime errors. The function block library consisted of 40 functions. Out of them some were found to have possible arithmetic overflow/underflow and divide by zero.

Table 4. List of examples used for testing the tool

Function Name	TCCs generated	TCCs unproved	RTEs
Sqare_Root	11	2	*Illegal value for function parameter*
Sqare_Root_concv	10	0	*No RTEs*
Close_To_Zero	11	4	*Floating point overflow/ underflow*
PID	15	7	*Floating point overflow/ underflow, divide by zero*
Non_Infinite_Loop	11	0	*No RTEs*

One of the requirements for the application of the tool is to provide the loop invariant for programs having loops. This may be difficult for general complex programs but in safety critical software the functions tend to be simple and our experience has been that with little effort the invariants can be written down.

In the present implementation, the proof process is not completely automated. User interaction with PVS is needed for proving TCCs related to loops and nonlinear arithmetic operations. Tracing the RTEs from the unproved TCCs is also not automatic. We are developing a front end GUI which can help the user to manage the proof process. There are also few issues with PVS cache size which we are trying to resolve. This influences the time taken to discharge the TCCs and the size of the program which can be checked for RTEs.

It would be an interesting future work to combine the general theorem proving capabilities with BLAST like model checking techniques to improve the automation of the class of properties that we are interested. The tools like PVS are,

indeed, useful for this as they can incorporate many automatic decision theories. In this sense, the proposed work, being based upon PVS, would be useful and forms a first step for combining theorem proving and model checking techniques for the class of properties that we are interested in.

Acknowledgments. We wish to thank the Board of Research in Nuclear Sciences (BRNS) for supporting this work.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J.: Microsoft Research, Thorough Static Analysis of Device Drivers. In: EuroSys 2006 (2006), Web page http://research.microsoft.com
2. Bjorner, N., Anca, B., Eddie, C., Chang, M., Kapur, A., Manna, Z., Simpa, H., Uribe, T.: The Stanford Temporal Prover Educational Release, Version 1.4-α, Computer Science Department, Stanford University (July 1998)
3. Cardelli, L.: Type Systems, Digital Equipment Corporation, Systems Research Center, Web page: http://lucacardelli.name/Papers/TypeSystems1stEdition.A4.pdf
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the ACM Symposium on principles of programming languages, ACM Press, New York (1979)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
7. Evans, D., Larochelle, D.: Improving Security Using Extensible Lightweight Static Analysis. IEEE Software (January, February 2002)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003), http://www.eecs.berkeley.edu/~blast
9. IEEE-754 Standard for Binary Floating-Point Arithmetic, IEEE 754-1985
10. International Electrotechnical Commission, Nuclear Power Plants I& C Systems Important to Safety Software aspects for Computer Based Systems Performing Category A Functions, IEC60880 Ed 2.2004 (2004)
11. MISRA-C:2004 - Guidelines for the use of the C language in critical systems The Motor Industry Software Research Association (October 2004)
12. Nethercote, N., Seward, J.: Valgrind: A Program Supervision Framework Electronic Notes in Theoretical Computer Science, vol. 89(2) (2003)
13. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, Version 2.4. SRI International (December 2001)
14. PolySpace Technologies Home Page, http://www.polyspace.com

A Taxonomy for Modelling Safety Related Architectures in Compliance with Functional Safety Requirements

Jesper Berthing* and Thomas Maier

Danfoss Drives A/S, Ulsnæs 1, DK-6400 Gråsten, Denmark
{F35594,TM}@danfoss.com

Abstract. This paper presents an extension and refinement to the modelling of architectures for safety functions as suggested in IEC61508-6. We propose an implementation oriented taxonomy providing an easy and unambiguous approach to model real life architectures in conformance with IEC61508.

Modelling safety related architectures with this taxonomy results in the following benefits: (1) A clear and unambiguous approach to the selection of required diagnostic techniques and measures (IEC61508-2 and IEC61508-3) based on the Safety Integrity Level (SIL); (2) Quick estimates of Probability of Failure on Demand (PFD)/ Probability of Failure per Hour (PFH) / Safe Failure Fraction (SFF) values in relation to the quantitative SIL requirements; (3) Optimising the design and performance by allocating specific diagnostic techniques to specific elements of the architecture; (4) Improved overview and understanding of the architecture supporting the development and certification process. The taxonomy is part of ongoing effort to automate the selection and conformance checking of diagnostic techniques and measures with IEC61508.

Keywords: dependable architectures, safety related architectures, IEC61508.

1 Introduction

A fundamental step in the development of safety-related functions to a given SIL in accordance with IEC 61508 is the identification of an adequate and realisable system architecture. The standard provides, in its part 6, some guidance for this step. Typical structures with different degrees of redundancy ("MooN"[1]), with or without diagnostics, are presented. For each structure, formulas and tables are provided for calculating Probability of Failure on Demand (PFD) / Probability of Failure per Hour (PFH) / Safe Failure Fraction (SFF) values.

* Thanks are due to Prof. Christo Angelov (University of Southern Denmark) and Jørgen Born Rasmussen (Center for Software Innovation, Denmark) for inspiration and supervision, and The Danish Ministry of Science, Technology and Innovation for a scholarship.

[1] MooN - M out of N, e.g. 1oo1, 1oo2, 2oo3, etc.

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 505–517, 2007.

These "MooN(D)"[2] architectures represent a high level of abstraction, and model a rather "ideal world" of safety-related systems. The developers, however, must implement real-life systems, where safety functions may share resources with non safety-related functions, required diagnostic functions may lead to performance problems, or hardware-implemented redundancy and diagnostics may not be possible but must be implemented in software.

This paper sets out to describe a taxonomy for refining the high-level architectures towards more implementation-near models in a formalised way, in order to both assure conformance with the requirements implied by a given SIL, and to account for the constraints and particularities of the real-life system to be developed.

Section 2 briefly describes the architectures from IEC 61508-6. Section 3 describes the foundation for the proposed taxonomy as well as the proposed taxonomy; the taxonomy is illustrated for the 1oo2D architecture for a single and a redundant processor platform in section 4. In section 5 the modelled 1oo2D architecture for the redundant processor platform is mapped to the hardware block diagram in the context of a safety related frequency converter. Section 6 and 7 describes the related work and future work respectively and section 8 concludes the presented taxonomy.

2 Architecture Modelling Based on IEC 61508-6

IEC 61508-6 [3] describes a number of different architectures to be used when modelling safety related systems. The aim of the system architecture is to calculate PFD/PFH value and to change the properties of the architecture to fulfil the required safety integrity level (SIL). Figure 1 shows a 1 out of 2 architecture with diagnostics (1oo2D), where the channels perform the safety function of the system. The diagnostics can activate a safe state independently from the channels.

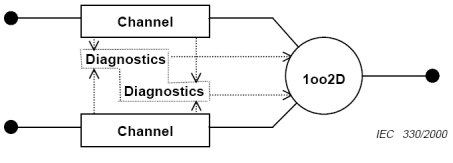


Fig. 1. 1oo2D architecture [3]

3 Taxonomy

This section describes the proposed taxonomy, which is an extension, and refinement to the way architectures are described in IEC 61508-6. The aim is to simplify the modelling of safety related architectures in relation to the required

[2] MooN(D) - M out of N with or without Diagnostic.

diagnostic techniques and measures as well as to provide an implementation oriented representation of the architecture. We will firstly analysis and describe the notation used in IEC 61508-6 to model the architecture, which was used as a basis to develop the proposed taxonomy in the paper.

The architectures presented in IEC 61508-6 are modelled consisting of elements such as channel, diagnostic and voting where the flow/relations between elements are modelled. The elements are represented using either a solid or dotted line where the solid line is used to represent elements related to the implementation of the safety function and the dotted line is used to represent elements related to the diagnostic functions. This defines the notation and functional representation used in IEC 61508-6 to model safety related architectures and will in the following be denoted as *Safe* and *Diagnostic*. Table 1 lists the foundation of the taxonomy used in IEC 61508-6.

Table 1. IEC 61508-6 architectural taxonomy

Function	**Notation**
Safe	———
Diagnostic	.....

In IEC 61508 it is not described if a hardware or software implementation has to be used to implement the architecture of the system - e.g. a watch dog timer could be either implemented in hardware or software. The 1oo2D architecture models a redundant system consisting of two channels. Realising such an architecture using a single processor system requires other measures to be applied compared to those for a dual processor system, in order to argument for the redundancy of each channel. The hardware/software platform is currently not modelled as part of the safety related architecture and it is therefore the safety engineer's responsibility to map the modelled MooN architecture onto the platform of the system. The relation between the architecture and the platform is therefore typically described using plain text and/or block diagrams. Therefore, in order to represent an element as either hardware or software, the taxonomy is extended such that elements implemented in hardware are denoted as *Physical* where elements implemented in software are denoted as *Virtual*. A colour notation is used to differentiate between *Physical* and *Virtual* elements in the architecture. The *Physical* elements are represented with a black colour and *Virtual* elements are represented with a grey colour.

Another important aspect of today's safety related systems is how to handle non-safety related functions such as normal operational data transmitted over a safety field bus such as PROFIsafe or DeviceNet Safety. IEC 61508 requires that the non-safety and safety related elements have a clearly defined interface, in order to justify the independence between these. Representing non-safety related elements, as part of the architecture is one approach to model the interface/relation between the *Safe* and *Non-safe* elements. A *Non-safe* type is therefore added to the taxonomy and represented in the architecture using a

Table 2. Safety architecture taxonomy

Function	**Implementation**	
	Physical	Virtual
Safe		
Diagnostic		
Non-safe		

dashed line. The three new identified types *Physical*, *Virtual* and *Non-safe* are added to the taxonomy derived from IEC-61508-6 and listed in Tab. 2.

The taxonomy from Tab. 2 is used to classify the elements of the safety related architecture and enables an element to be represented as either *Safe*, *Diagnostic* or *Non-safe* and are further refined as implemented in either hardware (*Physical*) or software (*Virtual*). This makes it possible to clearly differentiate between the elements of the architecture - e.g. a sensor that is part of a safety function (solid line) from a sensor used for diagnostic purposes (dotted line).

A number of typical elements have been identified and listed in Tab. 3 some of the elements are general (e.g. Sensor, Actuator, Final element actuator, Processor and Bus) where others are specific (e.g. Diagnostic, Safe channel, Fail safe sensor and Fail safe actuator). General elements can in principle be represented as one of the three functions from the taxonomy; whereas the specific elements are limited to either *Safe* or *Diagnostic*. Table 3 lists the elements that we have used with success in the development of this taxonomy and do not represent a final list of elements.

Using the taxonomy from Tab. 3 to represent a processor that is part of the safety related function is therefore represented as *Safe* (solid line) and *Physical* (black colour), where a temperature sensor used to implement a diagnostics function is represented as *Diagnostic* (dotted line) and *Physical* (black colour).

If an element of the architecture is used to represent more than one function of the taxonomy (*Safe*, *Diagnostic* and/or *Non-safe*) then the following hierarchical rule is used to determine how the element is represented: *Safe* < *Diagnostic* < *Non-safe*, meaning that the *Safe* function has higher "priority" than the *Diagnostics* and *Non-safe* functions. Therefore if a sensor implements a *Non-safe* function and the measured data is used in relation to the diagnostics of the system, the sensor is represented in the architecture as *Diagnostic* (Dotted line).

An element can encapsulate other elements of the same and different type in order to give a detailed representation of the architecture if necessary, e.g. a processor encapsulating a safe channel and a number of diagnostic techniques represented as software elements in the processor. This makes it possible to model a safety related system implemented on e.g. a System-On-Chip platform consisting of one or more processors. Figure 2 represents a System-On-Chip configuration consisting of two *Safe Physical* processors connected via a *Safe Physical* bus.

The 1oo1 architecture from IEC 61508-6 (See Fig. 3) is used as an example to illustrate how the taxonomy in Tab. 2 and the elements from Tab. 3 is used

Table 3. List of architectural elements

Element	Description
Sensor (S)	Source of signals or data (safety-related if solid, diagnostic-related if dotted, non-safe if dashed)
Actuator (A)	An interface to the environment representing e.g a switch, relay, door lock, ect.
Final element actuator (FE)	The final element actuator can, depending on the architecture of the system, activate the safety function
Processor	Contains logic and functions for safety, diagnostics, and normal operation.
Diagnostic	Implements a diagnostic function in either hardware or software
Safe channel	Implements a safety function (on a processor)
Bus	Uni and bidirectional communication between elements either hardware (e.g. network) or software (e.g. message passing) implementations
Fail safe sensor (FS)	Sensor implementing self diagnostic, or defined failure behaviour
Fail safe actuator (FA)	Actuator implementing self diagnostic

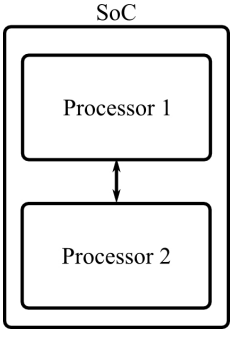


Fig. 2. System-On-Chip configuration

to model the architecture of a safety related system realised on two different hardware platforms. Figure 4(a) shows how the 1oo1 architecture is modelled consisting of a microprocessor encapsulating the safe channel implemented in software and the diagnostics of the channel implemented in hardware where Fig. 4(b) shows a microprocessor encapsulating the safe channel and diagnostics element implemented in software. Figure 4 illustrates the use of the *Physical* and *Virtual* notation to represent the elements of the 1oo1 architecture in relation to Fig. 3.

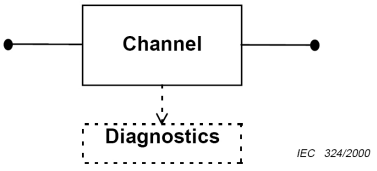
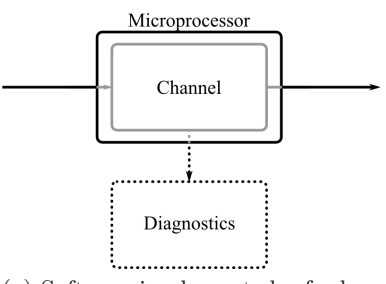
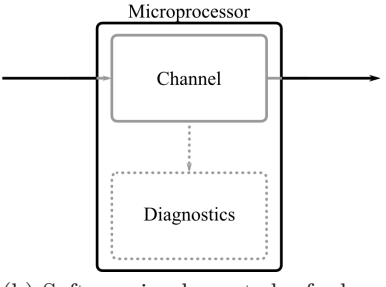


Fig. 3. 1oo1 architecture [3]



(a) Software implemented safe channel and hardware implemented diagnostics



(b) Software implemented safe channel and diagnostics

Fig. 4. 1oo1 architecture of a logical subsystem

4 1oo2D Architecture - Single and Redundant Processor Realisations

The 1oo2D architecture is used as an example to demonstrate how the taxonomy is used to model the safety related architecture of a safety related component - e.g. an emergency shutdown system. The safety related system is realised on two hardware platforms, a single and a redundant processor platform where both systems must fulfil the Safety Integrity Level (SIL) of 2.

The sensor subsystem consists of an emergency switch, a safe field bus and a reset switch. The reset switch is used to reintegrate the safety functions in case of a power cycle and/or the activation of the safety function (this does not include the activation via the safe field bus). The emergency switch and the safe field bus are used to activate the safety function. The logical subsystem consists of a non-safety related function that implements a gateway to/from a CAN network for the operational data transmitted over the safe field bus. The final element subsystem implements two independent ways to activate the safety function of the system.

Both hardware platforms are modelled using the following three steps: (1) The 1oo2D architecture from figure 1 is mapped onto the platforms as a basis for the (2) selection of diagnostic techniques and measures - (3) These are then allocated to the architecture and the relation between the elements is modelled.

4.1 Mapping the 1oo2D Architecture to the Hardware Platforms

The 1oo2D architecture (See Fig. 1) is mapped onto the single and the redundant processor platform, see Fig. 5 for the single and Fig. 6 for the redundant processor architecture. The electronic interface of the sensor subsystem for both architectures is modelled as two *Diagnostic* actuators (A) that are used to diagnose the sensors (S) and the input circuits of the processor. The *Safe* actuators (A) are used to activate the safety function and to diagnose the output circuit and final element actuators (FE). The actuators of the sensor and the final element subsystem could be implemented as a switch to remove the supply voltage for the input circuit of each sensor and likewise to remove the supply voltage for each final element actuator (FE).

The non-safety related elements for both architectures are represented using the *Non-safe* function from table 2; this clearly defines where the non-safety related functions (Gateway and Reset handler) interact with the other elements of the architecture. The safe field bus in Fig. 5 and 6 shows how the non-safety related gateway function is modelled exchanging the operational data between the two busses where the received safe data is passed as input to both safe channels.

4.2 Selection of Required Diagnostics Techniques and Measures

Figure 5 and 6 are used to select the required diagnostic techniques and measures to be implemented in order to fulfil the SIL requirements. The modelled architectures do already include elements representing the diagnostics part of the architecture in a high level abstraction where the aim of the remaining two steps is to select and represent these as part of the architecture. Based on the intermediate models of both architectures the required diagnostics techniques and measures are selected from the tables A.2 - A.15 from IEC 61508-2 [1]. The resulting set of selected measures are listed in Tab. 4 denoted with a "*" for both platforms where techniques denoted with a "(*)" are selected but not represented as part of the architecture.

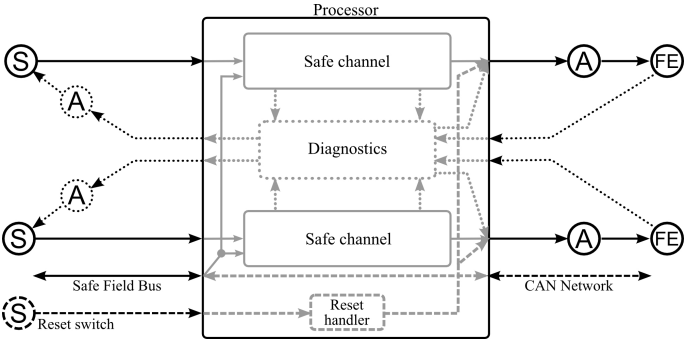


Fig. 5. Single processor 1oo2D architecture

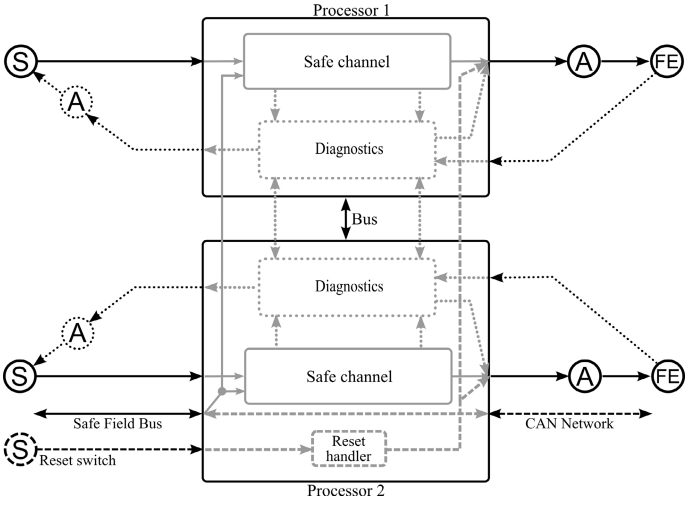


Fig. 6. Redundant processor 1oo2D architecture

4.3 Allocation of Diagnostics Techniques and Measures

The selected techniques are then allocated to the architecture of each platform as either physical or virtual diagnostic elements and the relation between elements is modelled. See Fig. 7 and 8 for the single and the redundant architecture respectively. The architectures for both platforms do not explicitly model how each diagnostics element activates the safe state of the system if a failure is detected.

The relation between hardware and software elements shows for example how the diagnostic element (A.2.5, A.6.1 and A.13.1) uses the "safe" actuator (A) to diagnose the output circuit related to the final element actuator (FE) and monitors the feedback. Other diagnostics techniques like A.4.3 (Signature of one word) and A.4.5 (Block replication) are applied to the software design and implementation of the other software elements.

The main difference between the single and the redundant processor architectures is that the safe channels of the single processor platform are represented as inverse of each other where this is not necessary in the redundant processor case. This is derived from the diagnostic techniques A.3.5, A.4.5 and A.6.5 arguing for the independence between the safe channels in the single processor case.

The safe field bus and the bus between the processors in the redundant architecture (Fig. 8) both transmits *Safe*, *Diagnostics* and *Non-safe* data, the safe field bus is modelled such that the data fans out/(in) where the communication between the two processors are modelled as *Virtual* relations across the boundary of each processor.

Table 4. Selected techniques and measures for both platforms

Table	Techniques / Measures	Maximum diagnostic coverage considered achievable	Single processor	Redundant processor
A.2 Electrical subsystem	A.1.5 Idle current principle	low	(*)	(*)
A.3 Electronic subsystem	A.2.1 Test by redundant hardware	Medium		*
	A.2.5 Monitored redundancy	High	*	*
A.4 Processing unit	A.1.3 Comparator	High		*
	A.3.2 Self-test by software: walking bit (one-channel)	Medium	*	*
	A.3.5 Reciprocal comparison by software	High	*	
A.5 Invariable memory ranges	A.4.3 Signature of one word (8-bit)	Medium	*	*
	A.4.5 Block replication	High	*	
A.6 Variable memory ranges	A.5.2 RAM test "walk-path"	Medium	*	*
A.7 I/O units and interfaces (external communication)	A.6.1 Test pattern	High	*	*
	A.6.5 Input comparison/Voting (1oo2,2oo3 or better redundancy)	High	*	*
A.8 Data path	A.7.6 Information redundancy	High	*	*
A.9 Power supply	A.8.1 Overvoltage protection with safety shut-off or switch-over to second power unit	Low	*	*
	A.8.2 Voltage control (secondary) with safety shut-off or switch-over to second power unit	High	*	*
	A.8.3 Power-down with safety shut-off or switch-over to second power unit	High	*	*
	A.1.5 Idle current principle	Low	(*)	(*)
A.10 Program sequence (watch-dog)	A.9.1 Watch dog with separate time base without time window	Low	*	*
	A.9.3 Logical monitoring of program sequence	Medium	*	*
	A.9.4 Combination of temporal and logical monitoring of programme sequences	High	*	
A.11 Ventilation and heating system (if necessary)	A.10.1 Temperature sensor	Medium	*	*
	A.10.4 Staggered message of thermo sensors and conditional alarm	High	*	*
A.12 Clock	See A.10			
A.13 Communication and mass-storage	Not relevant			
A.14 Sensor	A.1.1 Failure detection by online monitoring	Low	*	*
	A.6.1 Test pattern	High	*	*
	A.6.5 Input comparison/Voting (1oo2,2oo3 or better redundancy)	High	*	*
A.15 Final element (actuators)	A.1.5 Idle current principle	Low	(*)	(*)
	A.6.1 Test pattern	High	*	*
	A.13.1 Monitoring	High	*	*

This section has demonstrated how the proposed taxonomy is used to classify the elements of the architecture depending on their functionality and implementation. The modelled architecture can then be used in the further development of the software and hardware for the safety related system. The following section describes how the architecture for the redundant platform can be realised in the context of a safety related frequency converter.

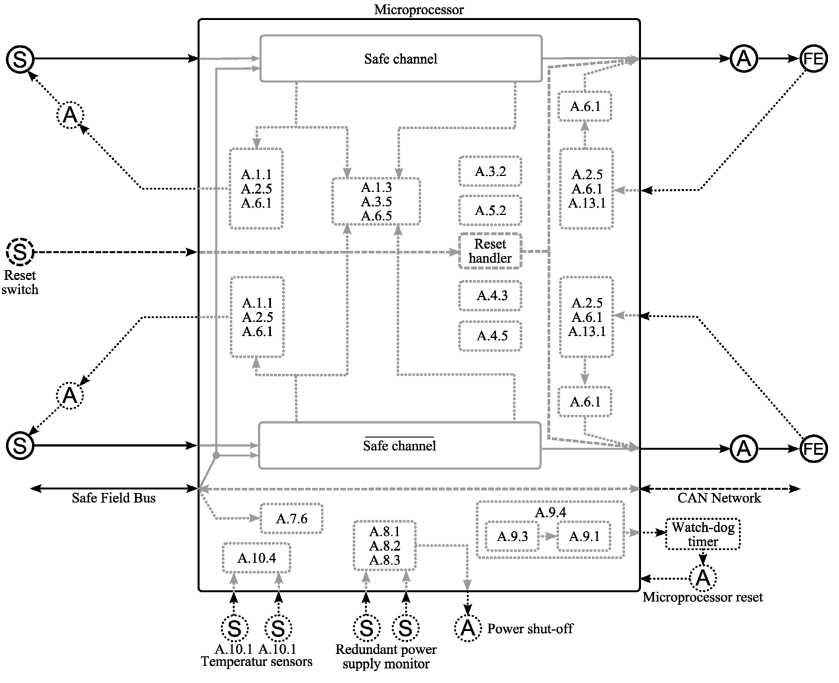


Fig. 7. Single processor 1oo2D architecture

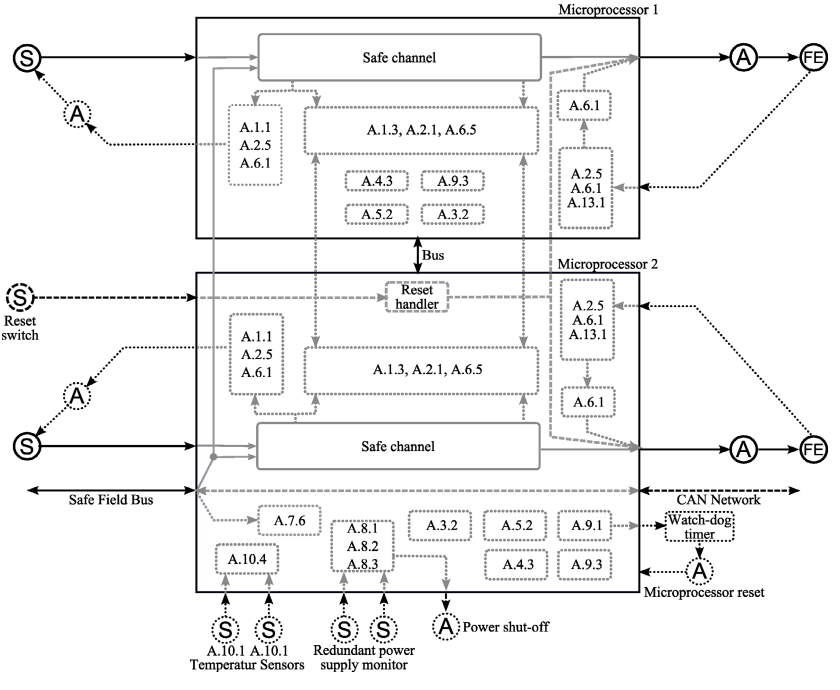


Fig. 8. Redundant processor 1oo2D architecture

5 Case Example Architecture for Safety Related Frequency Converters

This section describes how the 1oo2D architecture is realised in a safety related frequency converter. The basic functionality of a frequency converter is to control the speed of a motor by generating a PWM[3] signal to the power electronics (IGBT's[4]) that supply the motor. The frequency converter is for example used to control a conveyor belt in a production line.

The safety related function added to a frequency converter is to safely stop the rotation of the motor. Figure 9 shows a case example architecture for such a product, where the safe channels independently can activate the safety function of the device. The Emergency Stop button and the safe field bus are used to activate the safety function of the system. The architecture represents a real life implementation of the architecture described in the previous section (see Fig. 6).

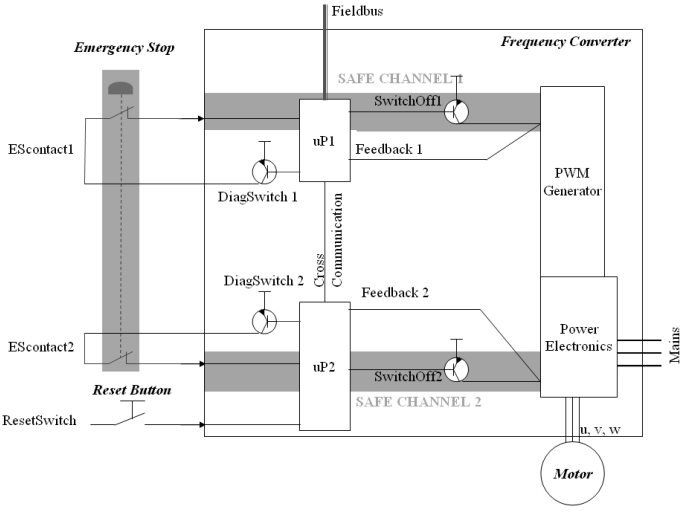


Fig. 9. Safety related frequency converter - block diagram

Mapping the architectural elements from Fig. 6 with the physical hardware components of the safety related frequency converter gives the following mapping: The two *Safe Physical* sensors (S) corresponds to the Emergency Stop button (EScontact1 and EScontact2), the sensors are diagnosed using the *Diagnostic Physical* actuators (A) corresponding to the transistors DiagSwitch1 and DiagSwitch2 and the *Physical* sensor (S) reset switch corresponds with the resetSwitch connected to uP2 - The *Safe Physical* microprocessors 1 and 2 corresponds to uP1 and uP2 - The *Safe Physical* bus (Safe Field Bus) corresponds

[3] Pulse Width Modulation.
[4] Insulated Gate Bipolar Transistor.

to the field bus connected to uP1 and the *Safe Physical* bus that implements the communication between microprocessor 1 and 2 corresponds with the cross communication between uP1 and uP2 - The *Safe Physical* final element actuators (FE) corresponds to the PWM generator and the Power electronics modules where the *Safe Physical* actuators (A) activates the safety function corresponding to the transistors Switchoff1 and Switchoff2.

The natural next step is then to further refine the hardware block diagram from Fig. 9 according to the architecture shown in Fig. 8, with the related hardware implementations of the temperature sensors, power supply monitor/ actuator and the watch-dog timer.

6 Related Work

The Architecture Analysis and Design Language (AADL) [5] describes three categories of component abstractions, e.g. application software, execution platform and composite. The application software consists of the following components: thread, thread group, process, data and subprogram. These are used to model the architecture of the application software. The application software is then mapped onto the execution platform, which is modelled based on the following components: processor, memory, device and bus. The application software and the execution platform are encapsulated in one or more system components of the composite category.

The 4+1 view model [4] describes a methodology to derive the software architecture using 5 concurrent views to capture the concerns of the stakeholders. Two of the views are concerned with the process and physical view, similar to the application software and the execution platform in [5]. The physical view is modelled using a generic notation consisting of components and connectors. Components are used to represent "processors" and "other devices" in the physical view where as the connectors connect the "processors" and the "other devices" representing one or more networks.

Our list of elements are similar in the sense of the components that are described for the physical view [4] and the execution platform [5]. The 4+1 and AADL can be seen as a lower level abstraction representing the architecture in term of threads, their interaction and their physical realisation on processors, other devices and networks.

The taxonomy presented in this paper, provides a higher level abstraction of the system architecture representing the architectural elements as either physical or virtual in terms of their implementation as well as their functional relation. The 4+1 or AADL can be extended with the presented taxonomy or seen as a natural next step in the further refinement of the safety related architecture, mapping the identified software and hardware elements onto the components defined in the two approaches [4] [5].

The emphasis of the taxonomy presented here is to model the architecture of the safety related system consisting of hardware and software elements, focusing on the relation between the MooN architecture and the applied diagnostic techniques and measures.

7 Future Work

The taxonomy is envisioned implemented in a graphical modelling environment to support the development of safety related systems and to automate the selection of diagnostic techniques and measures based on the modelled architecture as well as the functional safety requirements implied by the required SIL. Another issue is to use the modelled architecture to identify and design a set of reusable software components encapsulating one or more diagnostics techniques from IEC 61508-2, tables A.1 to A.19 [1] to be used in future safety related development projects.

8 Conclusion

The MooN architectures from IEC 61508-6 [3] have been used to derive the foundation of our taxonomy, which describes the functional relation of the elements in the architecture. We have extended the taxonomy with a new functional relation in order to represent non-safety related functions as part of the architecture as well as to further define elements as implemented in hardware or software. This has shown valuable, because it is possible to represent the interface between safety and non-safety related elements of the architecture. It provides a clear overview on the diagnostic techniques and measures needed in order to fulfil SFF and PFH/PFD requirements; and their allocation to specific parts of the architecture in order to account for performance and implementation constraints.

References

1. IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 2: Requirements for electrical/eletronic/programmable electronic safety-related systems. International Electrotechnical Commission, IEC 61508-2 (2000)
2. IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 3: software requirements. International Electrotechnical Commission, IEC 61508-3 (1998)
3. IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3. International Electrotechnical Commission, IEC 61508-6 (2000)
4. Kruchten, P.: Architectural Blueprints - The “4+1” View Model of Software Architecture. IEEE Software 12, 42–50 (1995)
5. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis & Design Language (AADL): An Introduction. Technical Note, CMU/SEI-2006-TN-011 (2006)

Controller Architecture for Safe Cognitive Technical Systems*

Sebastian Kain[1], Hao Ding[2], Frank Schiller[1], and Olaf Stursberg[2]

[1] Institute of Information Technology in Mechanical Engineering,
Technical University Munich, Germany
{kain,schiller}@itm.tum.de
[2] Institute of Automatic Control Engineering
Technical University Munich, Germany
{hao.ding,stursberg}@tum.de

Abstract. Cognition of technical systems, as the ability to perceive situations, to learn about favorable behavior, and to autonomously generate decisions, adds new attributes to safety issues. The system can cope with heavily changing conditions but its future behavior is not known a-priori. Therefore, present software solutions to safety like a comprehensive analysis of the specification and its implementation according to e.g. the V-model are not sufficient. The paper proposes an architecture for safe cognitive controllers consisting of an operational and a strategic functional part. While the first provides certified safety, the strategic part computes safe strategies based on appropriate dynamic models, adapted sets of safety specifications, and learned knowledge about potentially safety critical scenarios. Thus, the architecture explicitly uses cognitive functions to achieve safe behavior, and it allows the application of cognitively controlled plants for safety-related tasks.

Keywords: cognitive systems, cognitive controllers, safety, hybrid systems, learning.

1 Introduction

Current research on safety in industry and academia is related to random faults, software errors, and errors in human interaction without any explicit relation to cognition. Related standards, as the generic guideline on functional safety [12], do not consider the particular requirements and possibilities that apply for cognitive systems. The term *cognition* is here used according to [22], i.e. a system is considered to be cognitive if it is able to assess its environment and to autonomously select its actions even for new situations. The non-determinism of the behavior of cognitive systems (in reaction to changing environments) poses particular challenges for designing appropriate safety concepts for controlled cognitive technical systems.

* This work was partially supported by the cluster of excellence 'Cognition for Technical Systems' (CoTeSys), funded by the German Research Foundation (DFG).

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 518–531, 2007.

As long as safety is not guaranteed for these systems, the breakthrough into application will not take place. Currently, the only connection between safety and cognition is that cognitive systems are used to model human operators in safety-critical systems to optimize the interface design and to analyze human errors and reliability. In this context, several research results are published, see e.g. [4], [9], [21], including classifications of cognitive models and techniques, the categorization of safety systems, and the evaluation of cognitive systems with regard to safety. In automation, one present challenge is seen in the development of flexible low-cost solutions without any loss of provable safety. The focus is put to such an approach to safety, i.e. its guaranty should be achieved without assumptions on the application environment, the hardware, and the software like the operating system [10], [12], [15].

Safety can only be ensured, if the safety of both the controller software and its adaptation can be proved. In addition to the technical difficulties of this complex task, the necessary reaction time in case of a fault is usually very short such that learning and adaptive algorithms are not always completely suitable in this particular situation. As these algorithms are fitted for long-term actions like learning, additional short-term mechanisms have to be installed.

With respect to control methods relevant for the scope of this paper, one has to distinguish between dedicated supervisory controllers which trigger interlocks or shutdown procedures if a critical situation occurs, and techniques which utilize concepts of perception and learning. As for the latter, intelligent and iterative learning control refers to the class of controllers which adapt plant models to varying situations or conditions (often in form of neural nets or fuzzy models) and autonomously infer appropriate control actions (see e.g. [1]). These techniques do not address, however, the safety situation in particular – in contrast, this paper proposes a two-level approach in which (a) a strategic controller increases the availability and the reliability based on cognitive functions like learning principles for derivation of control strategies (non safety-critical) and (b) an independent operational controller ensures safety. By this architecture, the cognitive system can make use of previous experiences to derive future behavior which complies with safety requirements even for heavily changing conditions of the environment, i.e. a proof of safety and the advantages of learning algorithms are combined to a powerful cognitive system which is applicable to complex industrial automation tasks.

The paper is structured as follows: in Section 2 functions and architectures of cognitive systems are presented. Safety requirements and present solutions are explained in Section 3. In Section 4, the proposed architecture for cognitive solutions to safety is presented and described in detail. Finally, the application of the presented architecture is illustrated by means of an example of a car manufacturing cell in the automotive industry.

2 Functions and Architectures of Cognitive Systems

Cognition in cognitive psychology comprises functions of perception, comprehension, memorizing and remembering, thinking and problem solving, motion

control, and the use of language. Applied to cognitive systems, both biological and technical, this may be confined to the ability of systems to represent relevant aspects of the environment and the system itself by using prior experiences. Thus memorizing, as the ability to accumulate and store knowledge, is assumed as a basic requirement. Furthermore, functions for the processing of data, as well as planning and control of actions, hare parts of a cognitive system [22].

For understanding human control behavior and design of artificial systems, several cognitive architectures have been developed [20], some of which are briefly described: The EPIC (Executive-Process Interactive Control) architecture combines cognitive and perceptual operations with procedural task analysis [14]. Different interconnected modules, called processors, operate in parallel. Control is executed by a cognitive processor interconnected to modules and interpreting so called production rules.

In contrast to EPIC, the SOAR (symbolic cognitive architecture) architecture models behavior as selection and application of operators to a state. A state represents the current situation of knowledge and problem-solving, and operators transfer knowledge from one state to another. At runtime, SOAR tries to apply a series of operators in order to reach a goal [16]. Control in SOAR refers to the conflict solution and is implemented as a deliberate and knowledge-based process, whereas in ACT-R control is regarded as an automatic process by using an automatic conflict resolution strategy [13].

The main goal of these cognitive architectures is to build artificial systems by emulating human behavior. They do not focus on ensuring safety issues in technical systems so far. However, by applying these architectures to complex automated technical systems, the availability and productivity of a system can be increased due to cognitive planning components. Thus, the objective of this paper is to propose a control architecture which is, in contrast to standard control solutions, enriched by cognitive functions but can cope with the safety challenges arising from components for adaptation and learning.

3 Safety Requirements and Present Solutions

3.1 Requirements

In this paper, the manufacturing industry will be considered as domain of application. Typical are highly automated plants, in which, though, the collaboration between human operators and automated devices cannot be avoided for certain steps of production. The interaction with human operators can possibly lead to injuries if appropriate measures are missing – obviously, one requirement for the development of safety concepts is to minimize the probability of incidents in which human operators are harmed. Likewise, the likelihood of damage of the equipment or the contamination of the environment has to be excluded. Among several existing approaches to categorize safety requirements and its measures in the automation industry, this paper focuses on the performance level 'd' according to DIN EN ISO 13849, and on safety integrity level 3 according to IEC 61508, respectively.

In addition to plain safety requirements, the reliability and availability of the plant have to be taken into account. Typically, in order to enable a safety proof, measures are better conservatively designed to react more often than indeed necessary to avoid unsafe incidents [19]. Unfortunately, safety reactions usually decrease the productivity of the plant. Thus, a combined approach to safety and reliability is favorable. Concurrently to requirements relating to these two properties, requirements derived from the actual goal of operation of the production system have to be considered in the control architecture, as will become obvious in Sec. 4.

3.2 Non-cognitive Safety Controllers

A common architectural approach for technical applications (not only in the manufacturing industries) consists of controllers for non-safety related functions and a safety components (e.g. a Safety PLC). In modern hardware solutions, both types of functions can be executed in one device based on proved software diversity. The safety controller is connected to the process by safe sensors and safe actuators. The communication among controllers, sensors, and actuators has to meet safety requirements, too. Typical solutions are based on field bus communication either by proprietary protocols or by embedding safe protocols into standard ones [11].

Present solutions to the *design of safety-related software* of controllers consist of a comprehensive analysis (with or without consideration of the environment of the software). The software development process is executed according to the V-model including testing and verification in all stages, and both techniques can address static and dynamic properties in principle. Testing is usually carried out by simulating the software for randomly or arbitrarily chosen inputs, and it is observed whether any critical state is attained. Verification, in contrast to explicit tests, analyzes safety properties for all possible inputs and evolutions, and thus can prove the absence of critical behaviors. Recent advances in verification led to algorithms which can handle very large state spaces (above 10^{20} states) for control software that is converted into state transition models [3]. Furthermore, new approaches to verification of hybrid dynamic systems allow analyzing controlled system with complex nonlinear continuous dynamics, i.e. the controller can be checked in composition with the model to be controlled [24]. As the complementary task to verification, synthesis algorithms generate safe controllers from a model of the plant to be controlled and a safety specification. Research in this field has produced first algorithms to compute safe controllers for hybrid systems (e.g. [18], [25], [26]). However, neither verification nor synthesis techniques consider the particular requirements and safety properties of cognitive systems so far.

The proper *execution of safety-related software* is usually achieved by redundancy. Copies of the software run on different processors and their corresponding outputs are compared in order to detect random faults. Data exchange and general synchronization are usually enabled by fiber optic cables in industrial PLCs. If the comparison is not successful then a safe reaction has to occur with high

reliability. In many cases, a simple state like the voltage free state will be captured. The more functionality is involved in the safe reaction the more complicated becomes the proof since component reliability plays an essential role [6]. If a single processor is used, the software channels have to be diverse. This diversity is often a result of specific coding techniques and has to be proved [15]. Even solutions for one software channel executed on one hardware component are applied [7]. In that solution, only definite values of variables are permitted and processed correspondingly. In case of a fault, the probability to obtain such a fitting value is provably low.

4 Architecture for Cognitive Solutions to Safety

4.1 General Structure of Cognitive Safety Controllers

As described above, for application of cognitive algorithms to safety-critical technical systems, an adapted controller architecture is necessary. By means of this architecture, the cognitive abilities of the system should be used for achieving safety while, at the same time, safety has to be ensured. An architecture fulfilling these requirements is presented in Fig. 1.

The plant is controlled by a standard controller, connected by sensors and actuators. Safety is ensured by the Cognitive Safety Controller (CSC) connected to the plant by sensors and actuators which are safe by construction, i.e. failures of these elements can be detected and that lead to a safe reaction (e.g., so-called fail-safe states are reached).

The CSC receives information about the state of the plant by sensor and actuator signals transmitted by the standard controller. Vice versa, the standard controller obtains control information from the CSC, e.g. in case of emergency routines or if the cognitive safety controller computes new reference trajectories or parameters for the standard controller. The CSC consists of two main parts: a Strategic Controller (SC) and an Operational Controller (OC). The SC computes control strategies and actions that aim at keeping the plant away from safety-critical states, while the OC becomes active when a warning level (before

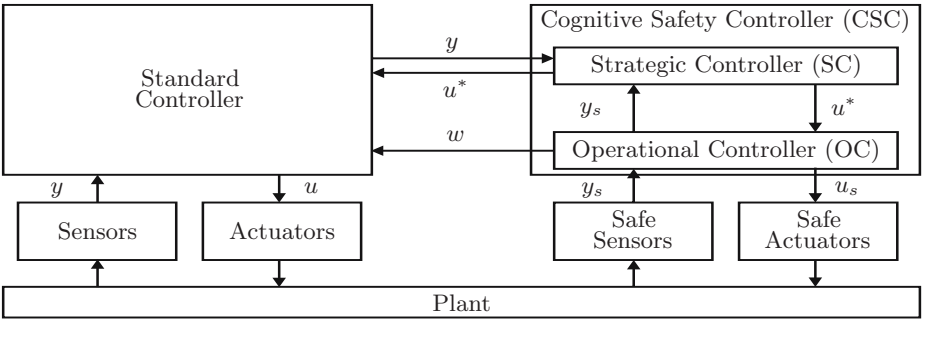
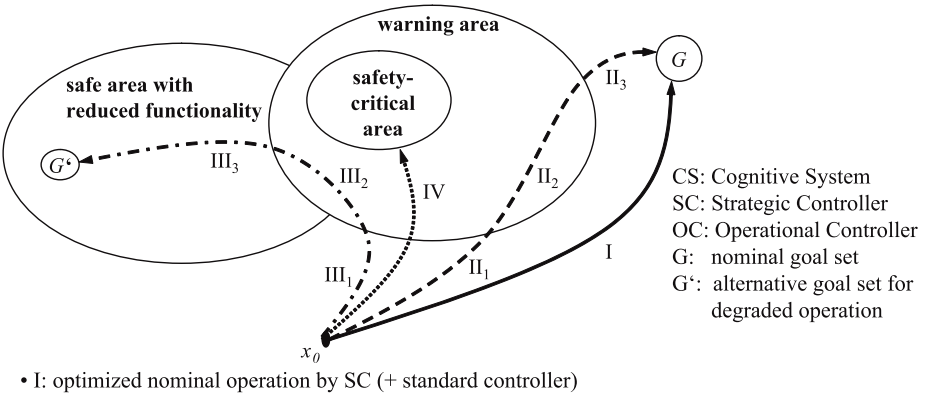


Fig. 1. Architecture of the Cognitive Safety Controller

reaching the critical state) is encountered, and the OC ensures that a critical state is never entered.

The measured variables y from the sensors are transferred to the standard controller and then to the SC. The signals y_s from the safe sensors are available in the SC through the OC, enhanced by information on the current mode of the OC. Based on these data, the SC updates the internal dynamic models of the other components, and these models are used for the optimization and learning process. The result is an optimal control action u^* which is passed to the standard controller (there executed as u) and the OC (executed as u_s), depending on the actuators which are affected. Besides checking the data y_s and u^*, the OC provides diagnostic values and information on safe reactions in form of the signal w.

The functionality of the CSC is further illustrated by Fig. 2. In nominal operation, the plant is controlled by the SC and the standard controller (I). If the system enters a warning region (due to a mismatch of the models used in the SC and the plant, e.g. in case of a device malfunction in the plant), the OC takes over control and drives the system out of this region (II), until the SC eventually resumes the nominal operation. If the nominal operation cannot be resumed due to a persistent failure, the SC and the OC drive the plant into a state of degraded operation or reduced functionality (III). An architecture without the SC and the OC could not prevent the plant from reaching the safety-critical area (IV).



- I: optimized nominal operation by SC (+ standard controller)
- II: optimized operation by SC (+ standard controller) with intermediate warning
  - II_1: CS controlled by the SC (+ standard controller)
  - II_2: by the OC (+SC)
  - II_3: by the SC (+ standard controller);
- III: degraded operation
  - III_1: CS controlled by the SC (+ standard controller)
  - III_2: by the OC (+SC)
  - III_3: by the OC (+ SC)
- IV: Unrecoverable operation without cognitive safety architecture

Fig. 2. Functionality of the cognitive safety controller

4.2 Strategic Controller

The objective of the SC is to compute a control strategy based on the current plant state, a model-based prediction of the behavior of the plant for the (near) future, and an assessment of the safety situation for this prediction. Accordingly, the SC comprises four main parts: a dynamic model of the plant, a strategy optimizer, a component for generating safety constraints, and a learning unit containing a knowledge-base. The computation of the strategy combines a performance maximization for reaching a control goal and the adherence to all constraints resulting from the safety assessment.

To obtain the strategy, an approach resembling *model predictive control*(MPC), a well-known online method to iteratively compute locally optimal controllers, is used (see e.g. [17,5]). MPC solves in any point of a discretized time axis an optimization problem which maximizes the performance for the controlled behavior based on a dynamic model and for a so-called prediction horizon (a limited time span starting from the current time). The optimization generates a (sub-)optimal control strategy, of which the part up to the next discrete point of time is applied to the plant. At any sampling time the horizon is shifted forward, and the calculation is repeated. The approach presented here extends this principle to cognitive systems in the following respects: (i) To represent the varying environment of the cognitive system, parts of the plant model are established as dynamic models with structural changes over time (see below). (ii) Safety restrictions to be obeyed by the controlled plant behavior are derived automatically from the current state of the plant model and a model of the OC. These restrictions are established as constraints of the optimization for the corresponding sampling time. (iii) The performance criterion considered for optimization is time-varying and thus adjusted to global specifications in the course of the iteration. (iv) Evaluated behavior, i.e. past evolutions of the plant and trajectories computed based on the dynamic model, is stored in the knowledge-base with a numeric quantifier expressing whether the behavior is preferable or has to be avoided with respect to safety. This knowledge-base is a crucial component of the learning unit which aims at determining suitable behavior (and thus control strategies) for situations that the cognitive system has not identically experienced before, but which have similarities to already evaluated situations and behaviors. A strategy derived by the learning unit is transferred to the optimizer as initialization. In turn, an optimal strategy obtained from the optimizer is stored in the knowledge-base, and it is transferred to the OC which checks the feasiblity of the action with respect to the current plant situation (which may deviate from the model used in the SC).

The structure of the SC is shown in Fig. 3 using the following notation: $T_p = \{t_k, t_{k+1}, \cdots t_{k+p}\}$ is a set of discrete time points considered in the prediction starting from the current time $t_k \in \mathbb{R}^{\geq 0}$; $y(t_k) \in \mathbb{R}^{n_y}$ denotes the vector of the current measured variables (the plant output); $\phi_{\sigma,k}$ is the predicted state trajectory; the state $\sigma(t)$ of the model combines information of the plant (the modeled dynamics of the cognitive system and its environment) and the OC; $\phi_{u,k}$ is the control trajectory computed by the optimizer ($\phi_{\sigma,i}$, and $\phi_{u,i}$ are intermediate results computed during the iteration of optimization and simulation

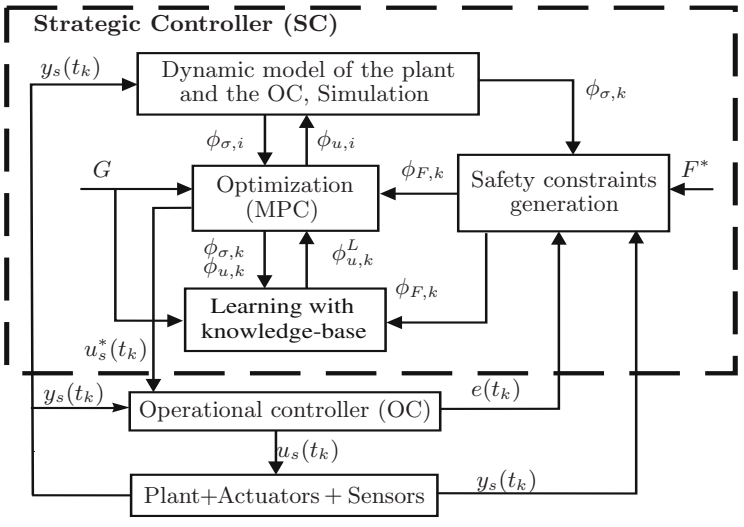


Fig. 3. Structure of the strategic controller. (The communication with the standard controller is left out here for simplicity).

carried out at time t_k). $\phi^L_{u,k}$ denotes the predicted control strategy derived from the learning block; if a notification about a safety-relevant incident is passed from the OC to the SC at time t_k, this information is encoded by the signal $e(t_k)$; this signal together with $\phi_{\sigma,k}$ and a set of generic user-specified safety specifications F^* is converted into the safety specifications to be considered in the optimization. These specifications enable the SC to foresee and thus avoid situations in the prediction, in which the OC had to interfere. $\phi_{F,k}$ denotes a sequence of state sets to be avoided. G specifies the control goal, i.e. a subset of goals into which the cognitive system should be driven. Finally, $u^*_s(t_k)$ is the control output from the OC to the plant at time t_k.

Dynamic model: Since cognitive systems operate in changing environments and have to adapt their behavior qualitatively to new situations, the use of hybrid dynamic models is appropriate, as they combine discrete (qualitative) with continuous dynamics [2]. The plant and the OC is modeled according to a hybrid automaton $HA = (X, U, Z, inv, \Theta, g, r, f)$ as in [23], where:

- $X \subseteq \mathbb{R}^{n_x}$ specifies the state space, on which the *continuous state vector* x is defined;
- $U \subseteq \mathbb{R}^{n_u}$ is the input space of dimension n_u;
- the finite set of *locations* is denoted by $Z = \{z_1, \cdots, z_{n_z}\}$;
- a mapping inv: $Z \to 2^X$ assigns an invariant set to each location $z_j \in Z$;
- the set of *transitions* is given by $\Theta \subseteq Z \times Z$;
- a mapping $g : \Theta \to 2^X$ associates a *guard* $g((z_1, z_2)) \subseteq X$ with each $(z_1, z_2) \in \Theta$;

- a *reset function* $r : \Theta \times X \rightarrow X$ assigns an updated state $x' \in X$ to each $(z_1, z_2) \in \Theta$, $x \in g((z_1, z_2))$;
- $f_z : Z \times X \times U \rightarrow \mathbb{R}^{n_x}$ defines *discrete-time continuous state update function* $x_{j+1} = f(z_j, x_j, u_j)$ in which z_j, x_j, and u_j denote the values of the location, the continuous state, and the input at time $t_j \in T_p$, respectively.

A *feasible run* ϕ_σ of HA is defined as a sequence $\phi_\sigma = (\sigma(t_0), \sigma(t_1), \cdots)$ of hybrid states $\sigma(t_j) = (z(t_j), x(t_j))$ with $z(t_j) \in Z$ and $x(t_j) \in inv(z)$. Let Φ_σ denote the set of all feasible runs. After initialization of $\sigma(t_0)$, the run is obtained by an alternating sequence of continuous evolution and transitions (see more details in [23]). The discrete part of the dynamics (i.e. the locations Z and the transitions Θ) are particularly suited to model the logic functions of the OC (see below).

Generation of safety constraints: The corresponding unit produces a sequence of *forbidden regions* $\phi_{F,k} = \{F_k, F_{k+1}, \cdots, F_{k+p}\}$ over the prediction horizon. Any $F_j \subset Z \times X$ defines for t_j a subset of the hybrid state space for which it must hold that $\sigma(t_j) \notin F_j$ for $j \in \{k, \ldots, k+p\}$. The function producing the forbidden regions according to $\phi_{F,k} = \Gamma(F^*, \phi_{\sigma,k}, y(t_k), \epsilon(t_k))$ takes the global safety specification F^*, the state trajectory obtained from simulation of the model $\phi_{\sigma,k}$, the current plant output $y(t_k)$, and the current state $e(t_k)$ of the OC into account.

Optimization: A run ϕ_σ obtained from simulation of the dynamic model and $\phi_{F,k}$ serve as constaints for an optimization to be carried out at any time t_k. The optimization problem is solved to yield the input trajectory $\phi_{u,k} = (u(t_k), \ldots, u(t_{k+p-1}))$, $u(t_j) \in U$, $j \in \{k, \ldots, k+p-1\}$ which minimizes a cost function J. The optimization problem can be formulated as:

$$\begin{aligned} \min_{\phi_{u,k}} \quad & J(\phi_{\sigma,k}, \phi_{u,k}, p, G, F) \\ s.t. \quad & \sigma(t_j) \notin F_j \ \forall j \in \{k, \ldots, k+p\} \\ & \phi_{\sigma,k} \in \Phi_\sigma \end{aligned}$$

Among many alternatives, a possible and simple version of the cost function is to minimize the distance of the last hybrid state within the prediction horizon to the goal set $G \subset \bigcup_{i=1}^{n_z}(z_j \times inv(z_j))$. For computation of the distance, appropriate measures on Z and X are defined. Depending on the specific form of J and the components of the dynamic model, a suitable solver has to be chosen.

Learning For a given *situation* in t_k, i.e. a combination of $\sigma(t_k)$, G, and $\phi_{F,k}$, the learning unit aims at inferring a proper control strategy $\phi^L_{u,k}$ and to pass the latter to the optimizer as initialization. The principle of the corresponding function $\phi^L_{u,k} = \lambda(\sigma(t_k), G, \phi_{F,k})$ is to compare the situation with entries in the knowledge-base and to select the most appropriate control strategy of those which are stored for previously encountered *similar* situations. Similarity is here defined by small distances of the quantities specifying a situation in the underlying hybrid state space. Entries in the knowledge-base are tuples $(\sigma(t_k), \phi_{\sigma,k}, G, \phi_{F,k}, \phi_{u,k}, \pi, J)$, where $\pi \in [0, 1]$ denotes a numeric safety indicator. This indicator is computed as a scaled distance of $\phi_{\sigma,k}$ (the state trajectory

starting from $\sigma(t_k)$ and arising from $\phi^L_{u,k}$) from the forbidden state sets $\phi_{F,k}$. The selection of $\phi^L_{u,k}$ by λ for a given situation is based on the associated value of the cost function J and on π.

4.3 Operational Controller

The OC provides the safety necessary for certification by independent authorities. Based on safe perception it avoids safety-critical states with safe reactions by overruling the instructions given to the plant by the SC, whenever a potential safety-critical state is detected. Safe perception means that a representation of the environment and the plant is obtained based on the measurements from the safe sensors.

The functionality of the OC is illustrated in Fig. 4. It carries out the following three actions:

- to ensure the appropriate planned safe reaction according to measured safe values supplied by the safe sensors;
- to ensure a safe behavior if the SC operates faulty or cannot reach the non-critical space for any other reason;
- to lead to safe states of the overall plant in case of:
  - erroneous sensor data,
  - erroneous data communication between OC and sensors and actuators,
  - or erroneous data processing within the OC.

Furthermore, it executes two information tasks:

- to consider information (u^*) from the SC for decision, which safe reaction should be executed, if there is a choice;
- to supply information about its behavior to the SC;

To ensure safety of the plant (including the environment), it is indispensable to measure the plant's state with respect to safety, e.g. signals which trace moving machine parts. Additionally, the correct execution of safety functions and corresponding data integrity have to be checked [12]. In case of a positive check of the data integrity (e.g. by correct checksum or corresponding values in case of redundancy), the signals can be used for selection of safe reactions.

The OC can react in different ways either to a safety-critical state or to a failure detected in the OC itself. The OC should only interfere, when the safety-critical state certainly would occur without the reaction of the OC. Safety-critical states are averted by safe reactions, e.g. deflecting or even stopping of a movement, or reducing velocities. In case of a critical state, a set of suitable safe reactions is derived from the current state what can be supported by an abstract representation. In general, there are various approaches to the design phase for the determination of a safe reaction for safety-critical states. Thus, a description of the plant is necessary, which allows a systematic deduction of safe

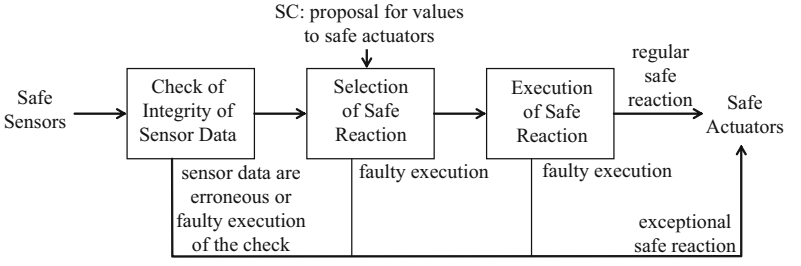


Fig. 4. The functionality of the OC

reactions. Efficient deduction is only possible, when this description is abstracted to safety-concerning attributes and their interactions. This description bases on methods like fault tree analysis, block diagram representations and Markov models, Failure Mode, and Effects Analysis (FMEA), Failure Mode, Effect, and Diagnostics Analysis (FMEDA), etc. [8]. Depending on the rules to be applied, an independent authority has to certify the resulting safety program.

Besides the correct measurement, it is necessary to have at least one safe reaction available to ensure safety in critical states. This has to be guaranteed by the design of the safe reactions. When the SC predicts the plant entering a warning area, it may preselect a possible safe reaction for the OC in addition to the avoidance measures. Independently from detected signal failures, a failed selection of reactions, or a failed execution of reactions, the checks of correctness as described above can fail itself. Then the OC's representation of the plant would deviate from reality. In this case, one must not assume that reactions selected based on this wrong representation would lead to safety.

Supervision in each check of correctness triggers safe reactions as well, see the blocks in Fig. 4. By failure detection, the system has to be transferred into a safe state (even if the availability of the overall plant may be affected).

5 Example: Car Manufacturing Cell

A manufacturing scenario of the automotive industry is shown in Fig. 5. Two human operators assemble a car supported by an automation manufacturing system, consisting of two moving portals and grippers moving in orthogonal direction. The gripper takes a part from a part supply, moves it to the car, and installs it. Simultaneously, the operators pick parts from supply 3 and mounts these to the parts installed by the grippers. Altogether the scenario includes 10 drives and approximately 40 sensors. According to the plant layout, the portals as well as the human operators share the same space the crossing of paths is necessary for completing the production. Since safety-critical situations may appear if the portals and the human operators collide, a particular safety concept is necessary. Furthermore, since the actions of the human operators are not exactly known a-priori, the automation of the manufacturing system calls for

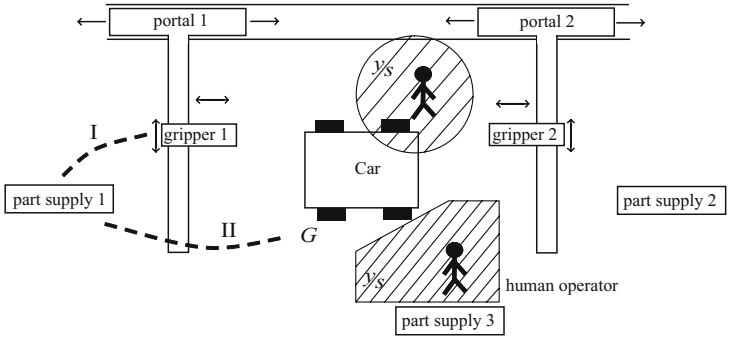


Fig. 5. SIL 3 Scenario for application of the Cognitive Safety Controller

cognitive functionality. The following safety-critical situations may arise in the manufacturing cell:

1. a human operator steps into the area of a moving object;
2. a gripper moves towards or above a human operator;
3. a gripper brings a human operator into danger by moving a wide-stretched sharp part.

The following safe sensors and actuators are available:

– a light curtain;
– safe position detectors for the human operators and portals;
– safe detectors for the parts;
– and safe drives for the portals and the grippers;

The SC optimizes the movements of the grippers including safety aspects. For example, in the procedure of mounting a part taken from supply 1 using gripper 1 requires to compute the trajectory of portal 1 and gripper 1 to the car, where the latter represents the goal set G in this case. The human operators, which may block the direct path temporarily, represent forbidden sets F. The optimization in the SC has to compute a trajectory which avoids F and minimizes, e.g., the time for the mounting procedure. Since the procedure is repeated for every car, the learning unit of the SC may, after some repetitions foresee the *typical* behavior of the human operators, and has an appropriate control strategy readily available in the knowledge-base.

The OC evaluates the signal from the light curtain and the positions of the portals. They are both implemented on a PLC whereas the OC is specifically coded based on the coding principle of [7]. The OC, the safe sensors, and the actuators communicate via the safe protocol PROFIsafe [11]. If a human operator moves into the area of moving objects, the SC changes the motion of the gripper dynamically in order to avoid a collision. If the SC fails in this, e.g. because of lacking solution from the optimization algorithm for very fast disturbances, the OC will turn off the power to the gripper immediately. If the light curtain

fails, a communication error arises, or the OC calculates an erroneous value, the same action is taken (as *last resort* safe action). The presented scenario is currently implemented and examined in a simulation environment, implementing the standard controller and the OC on a Soft PLC (Siemens WinAC) with cycle time 10 milliseconds and the SC in Matlab® with a prediction horizon in the range of 10 seconds.

6 Conclusions and Future Work

The paper proposes an architecture for cognitive safety controllers, leading to the following conclusions:

- Cognitively controlled plants can be used in safety-critical environments: the proposed architecture permits the use of any standard controller. Safety can be guaranteed also for cognitive and learning controllers, although the future behavior of the controlled plant is not known at design time.
- Cognitive algorithms can contribute to safety: controllers can actively make use of the principles of perception and learning in order to predict dangerous situations in the future, and they can align their behavior to this prediction to avoid unsafe situations.
- Cognition can increase the availability: by predicting and preventing possibly dangerous situations, the overall availability of the plant is enhanced. Furthermore, the optimization used in the SC (with an embedded model of the OC) maximizes the plant performances while considering the current safety restrictions and ensuring certified safety by the OC all the time.

The current work is focused on completing the implementation of the architecture and applying it to various scenarios of the described manufacturing example.

References

1. Antsaklis, P.J, Passino, K.M.: An Introduction to Intelligent and Autonomous Control. Kluwer Academic Publishers, Dordrecht (1993)
2. Balluchi, A., Benvenuti, L., Engell, S., Geyer, T., Johansson, K.H., Lamnabhi-Lagarrigue, F., Lygeros, J., Morari, M., Papafotiou, G., Sangiovanni-Vincentelli, A.L., Santucci, F., Stursberg, O.: Hybrid Control of Networked Embedded Systems. European Journal of Control 11, 1–31 (2005)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proc. 5^{th} IEEE Symp. on Logic in Comp. Science, pp. 1–33 (1990)
4. Boy, G.: Cognitive Function Analysis for Human-Centered Automation of Safety-Critical Systems. In: SIGCHI Conf. on Human Factors in Computing Systems (1998)
5. Carlos, D.M.P., Garcia, E., Morari, M.: Model Predictive Control, Theory and Practice - a Survey. Automatica 25, 335–348 (1989)

6. Exida, L.L.C.: Safety Equipment Reliability Handbook, Exida, Sellesville, USA (2005)
7. Forin, P.: Vital Coded Microprocessor - Principles and Application for Various Transit Systems. In: IFAC Conf. Control, Comp., Comm. in Transp., pp. 79–84 (1989)
8. Goble, W.M.: Control Systems Safety Evaluation and Reliability. In: ISA (1998)
9. Grant, S.: Safety Systems and Cognitive Models. In: 5th Int. Conf. on Human-Machine Interaction and Artificial Intelligence in Aerospace (1995)
10. Humphrey, D.W., Spada, S.: Siemens' Safety Integrated Adds Value to Automation Applications. ARC Advisory Group (2005)
11. Humphrey, D.W., Grundmann, U.: PROFIsafe – Networked Safety for Process and Factory Automation. ARC Advisory Group (2006)
12. International Electrotechnical Commission: Functional Safety of Electrical Safety-related systems. IEC Standard No. 61508 (2001)
13. Johnson, T.R.: A comparison of ACT-R and SOAR. In: Schmid, U., Krems, J., Wysotzki, F. (eds.) Mind modeling, pp. 17–38, Papst Publisher (1998)
14. Kieras, D.: EPIC Architecture – Principle of Operation, Univ. of Michigan (2004)
15. Krosigk, H.: Functional Safety in the Field of Industrial Automation. Computing & Control Engineering Journal, 13–18 (2002)
16. Laird, J., Congdon, C., Coulter, K.: The Soar User's Manual Version 8.6.3. University of Michigan (2006)
17. Mayne, D.Q., Rawlings, J.B., Rao, C.V., Scokaert, P.O.M.: Constrained model predictive control: Stability and Optimality. Automatica 36, 789–814 (2000)
18. Moor, T., Raisch, J., O'Young, S.D.: Discrete Supervisory Control of Hybrid Systems based on L-Complete Approximations. Journal of Discrete Event Dynamic Systems 12(1), 83–107 (2002)
19. Schiller, F.: The Relation between Safety and Reliability in Automation from the Safety Perspective (Plenary Talk). In: 11th Int. Symp. on System-Modelling-Control, Poland, pp. 13–19 (2005)
20. Schultheis, H.: Distribution and Association: Modeling Two Fundamental Principles in Cognitive Control. In: Proc. German Cognitive Science Conf., pp. 177–182 (2005)
21. Sträter, O.: Cognition and Safety. Habilitation, Institut of Ergonomics, Technical University of Munich (2006)
22. Strube, G.: Modeling Motivation and Action Control in Cognitive Systems. In: Schmid, U., Krems, J., Wysotzki, F. (eds.) Mind modeling, pp. 89–108. Pabst Publisher (1998)
23. Stursberg, O., Panek, S.: Control of Switched Continuous Systems based on Disjunctive Formulations. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 421–435. Springer, Heidelberg (2002)
24. Stursberg, O., Lohmann, S., Engell, S.: Improving Dependability of Logic Contr. by Algor. Verification. 16th IFAC World Congr., ID: Mo-E17-TO/6 (2005)
25. Stursberg, O.: Supervisory Control of Hybrid Systems based on Model Abstraction and Refinement. Journal on Nonlinear Analysis 65(6), 1168–1187 (2006)
26. Trontis, A., Spathopoulos, M.P.: Supervisory Target Control for Hybrid Systems. Int. Journal of Control 76(11), 1142–1158 (2003)

Improved Availability and Reliability Using Re-configuration Algorithm for Task or Process in a Flight Critical Software

Ananda Challaghatta Muniyappa

Aerospace Electronics & Systems Division, National Aerospace Laboratories, Bangalore, India
ananda_cm@css.nal.res.in

Abstract. Traditionally in avionics, Federated Architecture (FA) is used where each function has its own independent, dedicated fault-tolerant computing resources. FA though has the advantage of inherent fault containment but envelops a potential risk of massive use of resources resulting in increase in weight, increase in looming, cost and maintenance. Integrated Modular Avionics architecture (IMA) is successful, as it has an efficient and effective management of hardware and software computing. Most of the applications designed on IMA currently do not have dynamic reconfiguration. The paper presents a new method for re-configuration of tasks or a process in an embedded avionics application. The proposed algorithm works based on four control parameters: re-configurability Information factor, Schedulability Test/TL/UF, Context Adaptability/suitability and Context Flight Safety. The algorithm is data centric and interfaces system health as control input and initiation of the re-configuration is only after successful evaluation of the parameter metrics. It enhances the availability and reliability of the system under failed conditions by efficient selection and procedural re-configuration with safe state exit. The advantage of the new approach over the non-configurable systems is the increased availability of flight critical applications under failed conditions. It also preserves the advantages of non-Reconfigurable systems over federated architecture. Invalid failure of control parameter brings the system to safe state. The scheme, algorithm and the control parameters metrics and their validation approach are described. The algorithm provides very good availability of the system even under failures.

1 Introduction

The avionics systems and software architecture of federated era was no doubt very good in terms of the fault containment, fault tolerant and a sort of fool proof architecture. However, this has disadvantages like, increased weight, redundant computer resources in each Line Replaceable Unit (LRU), higher looming volume, electrical interfaces complexity and physical maintenance.

The advances in computer technology encouraged the avionics industry to utilize the increased processing and communication power and combine multiple federated applications into a single shared platform [1]. The Integrated Modular Avionics (IMA) was developed for integrating multiple software components into a shared

F. Saglietti and N. Oster (Eds.): SAFECOMP 2007, LNCS 4680, pp. 532–545, 2007.

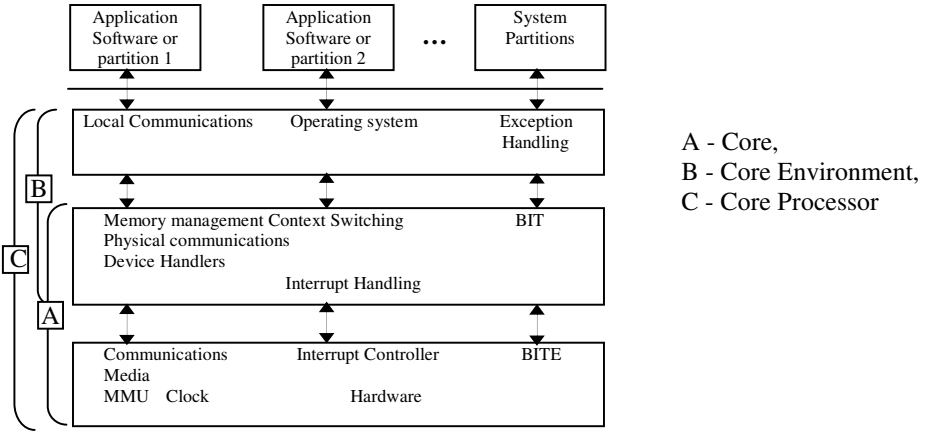


Fig. 1. System Architecture

computing environment [2]. This is powerful enough to meet the computing demands of multiple applications using common hardware and system resources.

The IMA integration has the advantage of lower hardware costs and reduced level of spares inventory. Typical system architecture for such IMA [2] is shown in Fig. 1.

1.1 Motivation and Related Work

Existing mechanism of system behavior in the event of a task failure is to declare system failure resulting in non-availability of either part or full partition functionality. Here the failure recovery, by removing the faulty task or replacing by a new task is not exercised. However, all the failures cannot be re-configured due to the safety and criticality of the avionics applications.

Proposed algorithm has the desirable feature of reconfiguring the critical tasks or removal from the schedule to enable continued functionality of the non-faulty partition. The novelty of the proposed algorithm is of reconfiguring the critical tasks or removal from the schedule to enable continued functionality of the non-faulty partition using control metrics [3]. This aspect has motivated to propose a new approach of re-configuration to attain higher system availability and improved reliability. This re-configuration algorithm is based on rule based decision-making approach and control metrics coupled with state and condition matrix. The algorithm is described for a typical multi partitioned multiple process task based architecture and finds its use in important phase of flight like cruise very easily. The other crucial phases of flight like take off and landing requires more validation, as these phases are very critical.

2 Organization of Task or Process Scheduler in a Typical Aerospace Avionics Application

Typical aerospace IMA applications employ multiple functionalities with the same hardware and system software resources.

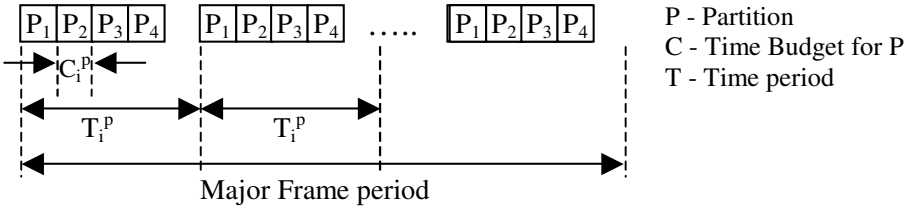


Fig. 2. Static table schedule diagram with partition period and process execution duration

This uses the concept of major frames, multiple partitions and each partition having multiple processes to schedule the tasks.

Fig. 2 shows the set of partitions [4], which are scheduled across a major frame M consisting set of partitions and each partition having set of tasks/process. Typical integrated avionics ARINC 653 based applications have a major frame. Major frame consists of number of partitions with each partition having set of process and each process having set of tasks. In some applications task and process are interchangeably used, but it is better to use process and task separately under the multi-process operating system.

Consider a major frame M having a set of partitions $Pt_i..Pt_n$ based on functionalities. Each partition Pt_i consists of a set of process $Ps_i..Ps_n$ based on the applications sub functionalities. The number of partitions and number of processes in each partition is a trade-off to get the real time response based on the capabilities of the hardware and software together. The representation of Major frames, partition, processes and each process with number of tasks represented as (1).

$$\text{Major Frame} = \begin{bmatrix} Pt_1 \\ Pt_2 \\ Pt_3 \\ . \\ Pt_n \end{bmatrix} = \begin{bmatrix} Ps_{11} & Ps_{12} & Ps_{13} & . & Ps_{1n} \\ Ps_{21} & Ps_{22} & Ps_{23} & . & Ps_{2n} \\ Ps_{31} & Ps_{32} & Ps_{33} & . & Ps_{3n} \\ . & . & . & . & . \\ Ps_{m1} & Ps_{m2} & Ps_{m3} & . & Ps_{mn} \end{bmatrix} \quad (1)$$

Each process P_s consists of set of tasks $\tau_1...\tau_n$ and the sequence of tasks are predefined and priorities are fixed as per static table scheduling mechanism is

$$\begin{bmatrix} \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n \\ \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n \\ \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n \\ & & \\ \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n & \tau_1,\tau_2,\ldots,\tau_n \end{bmatrix} \quad (2)$$

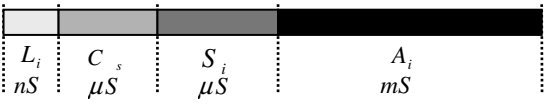
Each task τ_i has definite timing characteristics:

$C_i \leq D_i \leq T_i$, where C_i - Task Worst Case Execution Time, D_i - Task Deadline and T_i - Period

Also each task τ_i have other timing characteristics, which are critically examined for real-time capabilities like worst-case blocking, worst-case partition delay, worst-case process jitter and OS overheads. During the execution of a process, Worst Case Execution Time (WCET) and Worst Case Process Jitter (J) are the two important timing characteristics to be considered for realistic estimation of execution time.

Worst Case Process Jitter quantifies the maximum difference of the response time with the execution times for each period [5]. Jitter depends on the kernel overheads and partition jitter. Typical jitter measurements were carried out using embedded target to study the jitter timings (refer 3.4). These timing measurements help to characterize the delays and execution non-linearity in the algorithm.

However, the response time of a task or a process encompasses the various delays and execution times and they are



Where L_i – Interrupt latency time, C_s – Context save time, S_i – Schedule time
A_i – Process Time

Therefore the response time is expressed as

$$R_i = L_i + C_s + S_i + A_i.$$

3 Proposed Algorithm

3.1 Control Parameters for the Proposed Algorithm

A new re-configuration algorithm using critical control parameters is introduced using control parameter metrics. The re-configuration algorithm is implemented based on four major metrics, which are the heart of the algorithm in re-configuration. The Re-configurability Information-Factor, Schedulability Test/TL/UF, Context Adaptability/ Suitability and Context Flight Safety Factor are the efficient decision-making control parameters defined and used in the algorithm. Based on these control metrics, the re-configuration GO/NO-GO is decided.

3.1.1 Re-configurability Information-Factor (RI)

Re-configurability Information Factor (RI) is defined as the ratio of re-scheduled Task or Process Functional Credit Point (FCP) to the original scheduled task or process FCP. The FCP or Credit Point is the measure of the functionality characteristics in terms of its requirement and weight age of the task or process to accomplish the

defined system accomplishments. Credit point is represented in the range of 0 to 1 and hence FCP is the ratio of credit points.

For every selected critical task (τ_s) in a Major frame consisting of number of scheduled lists, there can be at least one configurable task (τ_r). The selection of replaceable task is based on the RI i.e., a process P_s or task τ_s can be re-configured by a process P_r or a task τ_r if and only if the RI of new process P_r or task τ_r should be at least equal to or greater than the RI of the faulty process P_s or task τ_s and is expressed as

$$(\tau_s = \tau_r) \leftrightarrow (RI(\tau_s) \geq RI(\tau_r)) \text{ or } (Ps = P_r) \leftrightarrow (RI(P_s) \geq RI(P_r)) \tag{3}$$

For every task (τ_s) or (P_s) there is exactly one task (τ_r)

(Denoted by E!) Or process (P_r) such that
[$RI(\tau_s) \geq RI(\tau_r)$] Or [$RI(P_s) \geq RI(P_r)$]
$((\forall \tau_s)(E!\ \tau_r)(RI(\tau_s) \geq RI(\tau_r))) \text{ Or } ((\forall P_s)(E!\ Pr)(RI(P_s) \geq RI(P_r)))$

FCP is derived based on the type of task, criticality of the task and phase of application envelope. For all critical tasks task τ in a process P_s scheduled in a partition P_t, has a defined FCP. Every element of (2) has corresponding credit point matrix as denoted in (4).

FCP elements in (4) are derived from the system requirements, design limits and Failure Mode Effect Analysis and Testing guidelines.

$$\begin{bmatrix} f_1, f_2, \ldots f_n & f_1, f_2, \ldots f_n & \cdot & \cdot & f_1, f_2, \ldots f_n \\ f_1, f_2, \ldots f_n & f_1, f_2, \ldots f_n & \cdot & \cdot & f_1, f_2, \ldots f_n \\ f_1, f_2, \ldots f_n & f_1, f_2, \ldots f_n & \cdot & \cdot & f_1, f_2, \ldots f_n \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ f_1, f_2, \ldots f_n & f_1, f_2, \ldots f_n & \cdot & \cdot & f_1, f_2, \ldots f_n \end{bmatrix} \tag{4}$$

3.1.2 Schedulability Test (Time Loading TL or Utilization Factor UF)

Schedulability Test is the standard method of testing the time loading or utilization for a task to be scheduled

$$(\tau_s = \tau_r) \leftrightarrow \left(WCET(\tau_s) \leq WCET(\tau_r) \right)$$
$$(\tau_s = \tau_r) \leftrightarrow \left(\left(\sum_{i=1}^{s_n} \frac{C_{s_i}}{T_{s_i}} \leq \sum_{i=1}^{r_n} \frac{C_{r_i}}{T_{r_i}} \right) \right) \tag{5}$$

Similarly for a process, the faulty process shall be replaceable if and only if Schedulability test as per (5) passes for a process.

For all cases of task phasing, a set of n tasks will always meet their deadlines [4] if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq U(n) = n(2^n - 1) \quad \leq 0.69 \tag{6}$$

And (5) and (6) is strictly enforced in algorithm for static computation of time loading in each schedule table of every partition.

Execution time or utilization is the important data resulting in efficient selection of a task or process to re-configure. Each task is benchmarked with the execution times and the same is used in real time for the algorithm and the corresponding matrix as per (2) is

$$\begin{bmatrix} t_1, t_2 \ldots . t_n & t_1, t_2 \ldots . t_n & . & . & t_1, t_2 \ldots . t_n \\ t_1, t_2 \ldots . t_n & t_1, t_2 \ldots . t_n & . & . & t_1, t_2 \ldots . t_n \\ t_1, t_2 \ldots . t_n & t_1, t_2 \ldots . t_n & . & . & t_1, t_2 \ldots . t_n \\ . & . & . & . & . \\ t_1, t_2 \ldots . t_n & t_1, t_2 \ldots . t_n & . & . & t_1, t_2 \ldots . t_n \end{bmatrix} \tag{7}$$

For selected critical tasks, reference execution time dataset is compiled and generated in accordance with (1) and (2). Re-configurable algorithm, which uses (7) as one control parameter input is tested using the data captured from a live flight critical project. The algorithm checks this reference dataset for task selection criteria.

3.1.3 Context Adaptability and Suitability (CAS)

Context Adaptability and Suitability metric decides acceptability of the faulty task replacement in real time. This involves checking the state table and condition table to decide whether the re-configuration is permissible. Hence the context of the scenario is verified and validated for the functionality and context suitability of the task.

Context Adaptability and Suitability (CAS) is defined as

(CAS=TRUE) ↔ (Re-scheduled Task or Process Context Flag is equal to Original Task or Process Context Flag)

And is expressed as

For every task (τ_s) or process (P_s) there can be a replaceable task (τ_r) or process (P_r) such that (8)

$$(CAS\ (\tau_s,\ \tau_r)\ is\ TRUE)\ \ Or\ (CAS\ (P_s,\ P_r)\ is\ TRUE)$$

$$((\forall \tau_s (CAS\ (\tau_s,\ \tau_r)\ is\ TRUE))\ \ Or\ ((\forall P_s (CAS\ (P_s,\ P_r)\ is\ TRUE))$$

Every task in a process and partition has the CAS flag dictating the function's use at that point of time using task reference dataset condition table. However, each task can

have more than one suitable tasks depending on the prevailing scenario (phase of flight) in real time. The CAS condition table used in the algorithm is derived based on the system functionality and inter system re-configuration dependencies based on Failure Mode Effect Analysis (FMEA) and Failure Hazard Analysis (FHA) along with System Safety Assessment (SSA).

3.1.4 Context Flight Safety Factor (CFS)

It is very vital in aerospace flight critical applications to check the safety of the system before and after re-configuration. After validating the above three control parameters, the system is checked for safe state to initiate re-configuration. For aircraft systems in closed loop control, a wrong function being re-configured can lead to catastrophic failure. Hence any action carried out in real time is verified and validated thoroughly by all the control parameter artifacts along with the system information.

Context Flight Safety Factor (CFS) is defined as

$$(CFS = TRUE) \leftrightarrow ((\text{Re-scheduled task or process Safety Factor / Original scheduled task or process Safety Factor}) \geq 1.0)$$

Also process or task is replaceable only if (9)

$$(P_s = P_r) \leftrightarrow (CFS(P_s) \geq CFS(P_r))$$
$$(\tau_s = \tau_r) \leftrightarrow (CFS(\tau_s) \geq CFS(\tau_r))$$

And is described as

$$\text{For every critical task } (\tau_s) \text{ or process } (P_s) \text{ there can be a replaceable task } (\tau_r) \text{ or process } (P_r) \text{ such that}$$

$$(CFS(\tau_s, \tau_r) \geq 1.0) \;\; Or \;\; (CFS(P_s, P_r) \geq 1.0)$$
$$((\forall \tau_s (CFS(\tau_s, \tau_r) \geq 1.0)) \;\; Or \;\; ((\forall P_s (CFS(P_s, P_r) \geq 1.0))$$

A process P_s or task τ_s can be re-configured by a process P_r or a task τ_r if and only if the safety factor of new process P_r or task τ_r should be at least equal to or greater than the safety factor of the faulty process P_s or task τ_s and is expressed as in (9).

CFS is derived from both RI and the Safety Units (Su) based on the Failure hazard Analysis (FHA), Failure Mode Effect Analysis (FMEA) and System Safety Analysis (SSA)[6]. Every element of (2) has corresponding Safety Unit matrix, which will be used by (9). Su is a measure of margin of system safety to re-configure a task with the prevailing dynamic context of the flight.

In case of failure of task after re-configuration, the system considering the safety issues can have Degraded mode for limited functionality. The extent of degraded performance allowed in such safety critical systems is decided based on CAS and degradation factor derived specific to the application objectives and functionalities as described in (10) for both process and task.

$$
\begin{aligned}
&\textit{If } (\tau_{CAS} \textit{ is TRUE) then} && \textit{If } (P_{CAS} \textit{ is TRUE) then} \\
&\quad (\tau_{CFS} = \tau_{RI}) && \quad (P_{CFS} = P_{RI}) \\
&\textit{Else} && \textit{Else} \\
&\quad \tau_{CFS} = \tau_{RI} * \textit{Degradation Factor} && \quad P_{CFS} = P_{RI} * \textit{Degradation Factor}
\end{aligned} \tag{10}
$$

The Degradation Factor (DF) is the measure of allowed degraded performance or functionality in selected envelope of the system being re-configured. If degraded functionality is not allowed then the degradation factor is 1. The reference data set of DF is captured from functionality requirements under dynamic pre-defined scenarios. Each scenario will be analyzed statically and the functionality is simulated for varying degraded functions and finally the data is compiled for each functional requirement as what is the extent of degradation allowed.

3.2 Condition, Status and State Information

Input reference dataset for the control parameters used in the algorithm depends on the information of the system are captured by System Design and analysis, Sample Implementation on typical platform, Aircraft Level Failure Hazard Analysis (FHA), system level Failure Mode Effect Analysis (FMEA), FAA/TSO requirements for aerospace flight critical systems and System Safety Analysis (SSA). The dataset for each of the control parameters are captured from live projects of flight critical in nature during the design and integration phase. Each control parameter will have dataset captured with varying real time scenarios.

3.3 Re-configurable Algorithm

A non-Reconfigurable system either shuts down or performs a partial degraded functionality in the event of a task failure. In some cases, this may lead to infinite loops or crash of the application leading to serious failure. Here the fault is not resolved rather the system enters failed state.

The proposed algorithm overcomes above fault scenario by re-configuration of faulty task resulting in recovery of fault in complete or partial. The identification of a task failure is by the monitors and is called *System Monitors.* System Monitor continuously monitors the state and status of critical tasks with reference to the schedule sequence with control parameters in context. The algorithm replaces a faulty process or task by a compatible, suitable and safe substitute after extensive check and validation. The re-configured task or process performs the required operation without any safety impact to the system and aircraft.

After careful design and definition of control parameter metrics for re-configurable decision-making as described in section 3.1, the following algorithm is proposed for re-configuration of a task or a process. The algorithm has the fail off path in case the algorithm enters the fault loop with multiple re-configurations without effective output. This is handled by a re-configuration counter, which avoids the repetitive reconfiguration for the same failure.

Proposed Algorithm

- ***If a task/job fails***
  - *Capture the task (τ_s) status, functionality, priority, criticality to identify the faulty task*
  - *Identify the most suitable substitution task (τ_r) after validating the following metrics for feasibility*
    - *Re-configurability I-Factor (RI)*
    - *Schedulability Test/TL/UF (TL)*
    - *Context Adaptability/suitability (CAS)*
    - *Context Flight Safety Unit (CFS)*
    - *Re-configure the task table or Process (Ps) before the next major frame after system assessment of functional state of the partition.*

- ***If re-configured task fails,***
- ***If the system can run in de-graded mode***
  - *Revert all tasks to its original state*
  - *Identify set of tasks which needs to be removed from the schedule*
  - *re-schedule the task set with de-graded performance using dead task removal techniques (all the failed tasks are removed from the task set)*
  - *The rest of the task set continues to run provided no safety impact after re-schedule*

- ***In case de-graded mode is not feasible,***
  - *Declare failure*
  - *Shutdown the system*

The algorithm is well suited for an open architecture multiple process and multiple schedule static table mechanism. The algorithm plays the role of high-level real time monitor software continuously monitoring the status of the running tasks of a schedule table. The algorithm is explained in the following five phase.

3.3.1 Phase I: Status Capture

The algorithm starts with continuous monitoring the status and health of a task execution. The data capture is part of the application software and the algorithm receives the information on function call or through global shared resources. When the algorithm detects a task failure or not performing as per its functionality, then the algorithm initiates the Phase II execution.

3.3.2 Phase II: Control Parameter Validation

The main objective of phase II is to identify the most suitable task using the results of control parameter validations. In any case, if any of the control parameter fails to comply with limit values, the algorithm returns to the system without any action resulting in resumption to the original state or transits to phase IV. On successful completion of control parameter checking and validation, the algorithm initiates the Phase III execution.

3.3.3 Phase III: Re-configuration

During the re-configuration process, the global state of the system, process or partition is not altered. Only the selected task or process gets altered for their respective state variables. On successful re-configuration, the re-configured task start execution in the next major frame.

3.3.4 Phase IV: De-graded Performance

If the re-configured task fails again in the next major frame, then the algorithm reverts back to its earlier state by reverting the re-configured task. If the degraded performance or functionality is allowed for that particular function, then the algorithm removes/alter order of the faulty task or process from the schedule table and allows it to continue. In this case, the system continues to execute without the functionality of removed task. This is still an improved mechanism instead of totally shutting down the application with many others tasks in good state.

3.3.5 Phase V: Fail Off Procedure

During the execution of the algorithm if the degraded functionality is not allowed then the algorithm behaves similar to the normal process of entering failed state.

The re-configuration algorithm is applied only for the critical task or process, which improves the availability as an effect of re-configuration. The algorithm is simulated on a test platform using predefined known states and the work is in progress to simulate using the realistic dataset. The simulated results under defined conditions show significant improvement from reliability numbers from 1E-05 to 1E-07 under failed conditions for critical task. The reliability numbers is for the system under critical function failure followed by recovery and not for the task. Configuration of a task or a process in aerospace flight critical system is a crucial event with the safety of the aircraft and availability of systems. Hence to identify a task for re-configuration requires severe judgment, methodical analysis, extensive cross checking across various relative parameters in real-time. The control parameters are checked for their states, status and validation before re-scheduling the sequence of tasks. Failure data collection for various scenarios is based on standards and equipment life cycle and quality control data management [7].

3.4 Simulation and Experimental Data

A sample schedule partition is simulated in Matlab Simulink using the state machines to check the time loading and execution scenarios. Even though the target is not that of a real environment, this gives a platform to study the timing behavior of system under varying external reactive interfaces. Experiments using low scale target hardware shows an average of ± 0.1 ms, ±0.14 ms, ± 0.2 ms and ±0.2 ms execution and jitter timing for 10 ms, 30 ms, 40 ms and 50 ms interrupt intervals for various configurations under multiple code composition in terms of the control and data flow deviations. The scenarios were simulated to capture the worst-case timings with constructs simulating the single and multiple failures.

The measurements were carried out with an embedded target system working at 20.0 MHz clock and Fs/4 internal clock frequency. Measurements show an average value of 20 to 30 μsec for the interrupt jitter or interrupt arrival interval time.

However these numbers vary across target to target with different clock frequencies and architectures. In this experiment an effort was made to study the interrupt interval time variation with varying clock frequencies and dynamic external reactive inputs for the same target. Context switching time is very vital in determining the response time of a task. An effort was made to capture typical switch time between an interrupt and a task entry / exit-using RISC based Micro-controller as target. These measurements were used in algorithm simulation.

Experiments showed the interrupt and function call timing measurements in entry and exit conditions based on simulated task execution are 2.52 μsec and 7.62 μsec for 11.056 MHz and 4.0 MHz clock respectively. The complete partition schedule was simulated using Matlab Simulink and the time loading aspects were studied with varying time loading or utilization of 0.32 to 0.7. Also simulation was done with varying fault scenarios and the results of varying task timings are as shown in Fig. 3 and Table 1.

The tasks used in the simulation were first scheduled as per the fixed priority scheduler and the sequence of tasks with execution times were simulated using Time Optimization of Resources, SCHEduling (Torsche) toolbox [8]. Torsche results were used to sequence the tasks in Matlab Simulink as table driven fixed priority scheduling for implementing the algorithm.

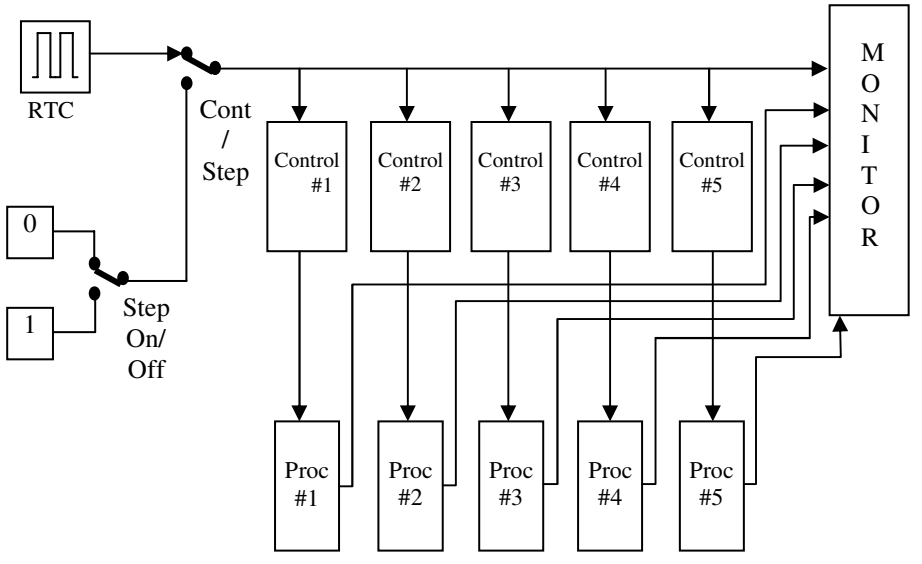


Fig. 3. Simulation of multiprocessing multitasks

TORSCHE is a MATLAB-based toolbox including various scheduling algorithms, which are used for various applications as high-level synthesis of parallel algorithms and optimized production of manufacturing lines. Using the toolbox, one can easily and quickly obtain an optimal code of computing intensive applications running on specific hardware architectures

Table 1. Timing for a varying fault scenario of a set of tasks

Tasks	Trail #1(ms)	Trail #2(ms)	Trail #3(ms)	Trail #4(ms)	Trail #5(ms)
IO Proc	4.77	4.57	3.50	4.64	4.11
Sensor Valid	3.33	3.23	4.44	3.12	3.46
LMS Filter	0.03	0.03	0.032	0.031	0.034
Cross Comp	1.27	1.25	1.26	1.28	1.29
RT Comp	3.9	2.86	3.90	3.86	3.75
Data Format	3.9	4.10	3.50	2.66	2.54
Output Proc	0.03	0.03	0.034	0.031	0.030
Fault Mngt	0.03	0.03	0.031	0.032	0.031
Time Loading (%)	**69.17**	**64.50**	**66.88**	**62.73**	**61.06**

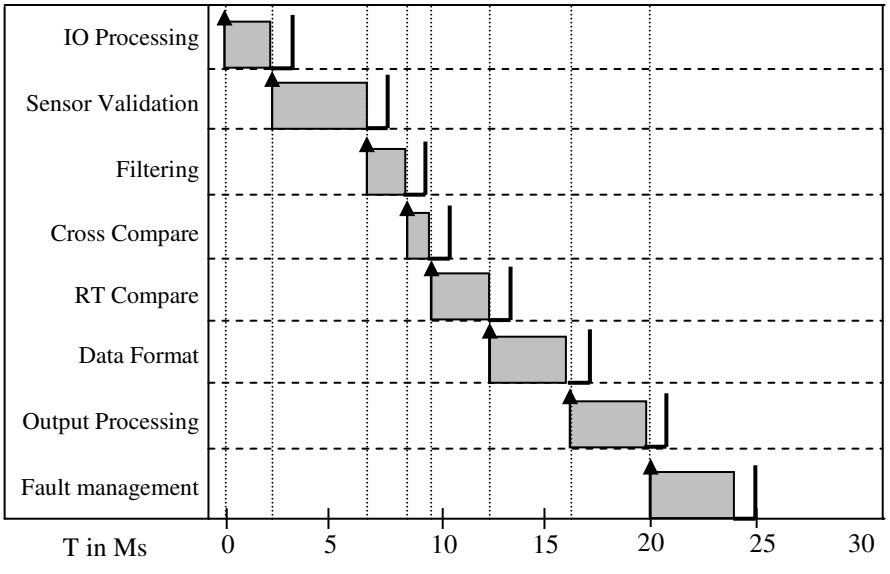


Fig. 4. Torsche scheduler task sequence optimization

The tool can also be used to investigate application performance prior to its implementation and to use these values in the control system design process performed in Matlab / Simulink. Torsche supports number of scheduling algorithms, but the fixed priority scheduler is selected for modeling optimized task sequence for our algorithm as shown in Fig. 4. The total frame time of 40 ms was used and corresponding tasks executed with utilization of up to 24 ms.

4 Conclusion and Future Work

Consequent to research and study, the algorithm along with the control parameter metrics were designed and defined. Identification of suitable task as a substitute for a

faulty task is very crucial and difficult task. The effectiveness of the algorithm is purely dependent on the system information available at the instance of failure in real time. Also the fault models of the dynamic environment should be considered for simulation for a realistic behavior of the algorithm. The simulation is being carried out using True Time [9] plug-in to Matlab Simulink with various fault scenarios. However the data generation for the control parameter metrics is very crucial for the full-fledged algorithm simulation. The preliminary data generation activity is completed and the second level data generation is in progress.

The solution to the problem of software complexity is not to avoid complexity rather to develop reliable protection and safety mechanisms to handle such scenarios. At the same time the implementation overheads should be maintained to the least possible for effective resource management. The re-configurable algorithm described offers the benefits of higher availability with the state of the art techniques. The algorithm uses re-configurability Information factor, Schedulability Test/TL/UF, Context Adaptability/suitability and Context Flight Safety for efficient and safe re-configuration for effective failure handling. Bench marking of all these control parameter reference dataset is not covered in this paper.

Work is being done in optimization of the control parameter validation process for task selection and compiling the required reference dataset for the algorithm using flight critical open architecture platform. Also the algorithm fault scenarios are being evolved and studied using true time, Torsche and neural network model using Matlab Simulink.

Acknowledgment. Author thanks Dr. SV Narasimhan, Deputy Director, National Aerospace Laboratories for his guidance and motivation from time to time. Author also thanks Prof. Y Narahari, CSA IISC, Prof. S Govindarajan, SCRC, IISC, Dr. BS Adiga, Dr. MR Nayak, Head ALD and Dr. AR Upadhya, Director NAL.

References

1. ARINC report 651, Design Guide for Integrated Modular Avionics, Published by Aeronautical Radio Inc., Annapolis, MD (November 1991)
2. ARINC Specification 653-1, Avionics Application Software Standard Interface, Published by Aeronautical Radio Inc. (October 2003)
3. Ananda, C.M.: Avionics for general aviation light transport aircraft: An insight into the avionics architecture and integration. In: AIAA Southern California Aerospace Systems and Technology Conference, May 2007, Santa Anna, California, USA (2007)
4. Audsley, N., Wellings, A.: Analyzing APEX Applications. In: IEEE Real Time Systems Symposium RTSS (1996)
5. Briand, L.P., Roy, D.M.: Meeting deadlines in Hard Real-Time Systems The Rate Monotonic Approach. IEEE Computer Society, Los Alamitos (1999)
6. IEC 60812, Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA), IEC 60812 Ed. 1.0 b:1985 (1985)

7. Dhillon, B.S.: Design Reliability: Fundamentals and Applications, CRC London New York Washington D.C (1999)
8. Miloslav, S., Michal, K.: Torsche scheduling toolbox: ListScheduling. In: 7th International Scientific – Technical Conference – PROCESS CONTROL 2006, June 13–16, 2006, Kouty and Desnou, Czech Republic (2006)
9. Benitez-Perez, H., Garcia-Nocetti, F.: Re-configurable Distributed Control. Springer-Verlag London Limited, London (2005)

Author Index