Steffen Becker
Frantisek Plasil
Ralf Reussner (Eds.)

LNCS 5281

Quality of Software Architectures

4th International Conference on the Quality of Software Architectures, QoSA 2008
Karlsruhe, Germany, October 2008
Proceedings

Springer

Lecture Notes in Computer Science 5281

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Steffen Becker Frantisek Plasil
Ralf Reussner (Eds.)

Quality of Software Architectures

Models and Architectures

4th International Conference on the Quality
of Software Architectures, QoSA 2008
Karlsruhe, Germany, October 14-17, 2008
Proceedings

Springer

Volume Editors

Steffen Becker
FZI Forschungszentrum Informatik
Haid-und-Neu-Strasse 10-14, 76131 Karlsruhe, Germany
E-mail: sbecker@fzi.de

Frantisek Plasil
Charles University, Department of Software Engineering
Malostranske nam. 25, 11800 Prague 1, Czech Republic
E-mail: plasil@dsrg.mff.cuni.cz

Ralf Reussner
Universität Karlsruhe (TH), Karlsruhe Institute of Technology (KIT)
Institut für Programmstrukturen und Datenorganisation
76128 Karlsruhe, Germany
E-mail: reussner@ipd.uka.de

Library of Congress Control Number: 2008935391

CR Subject Classification (1998): D.2.9, D.2.11, B.8, D.4.8, C.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-87878-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87878-0 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12529599 06/3180 5 4 3 2 1 0

Preface

Models are used in all kinds of engineering disciplines to abstract from the various details of the modelled entity in order to focus on a specific aspect. Like a blueprint in civil engineering, a software architecture provides an abstraction from the full software system's complexity. It allows software designers to get an overview on the system under development and to analyze its properties. In this sense, models are the foundation needed for software development to become a true engineering discipline.

Especially when reasoning on a software system's extra-functional properties, its software architecture carries the necessary information for early, design-time analyses. These analyses take the software architecture as input and can be used to direct the design process by allowing a systematic evaluation of different design alternatives. For example, they can be used to cancel out decisions which would lead to architecture designs whose implementation would not comply with extra-functional requirements like performance or reliability constraints. Besides such quality attributes directly visible to the end user, internal quality attributes, e.g., maintainability, also highly depend on the system's architecture.

In addition to the above-mentioned technical aspects of software architecture models, non-technical aspects, especially project management-related activities, require an explicit software architecture model. The models are used as input for cost estimations, time-, deadline-, and resource planning for the development teams. They serve the project management activities of planning, executing, and controlling, which are necessary to deliver high-quality software systems in time and within the budget.

The 4th International Conference on the Quality of Software Architectures (QoSA) 2008 focused on software architecture models and modelling techniques through its motto "Models and Architectures." A focus was set on models which aim at improving the predictability of the quality of systems under development. In so doing, it continued QoSA's tradition of using software architectures to develop and evolve high-quality software systems. Such architectural models contain details on the structure of a software, i.e., the components and connectors it is built of, the software's behavior, i.e., its control and data flows, and the software's deployment, i.e., the allocation of components and connectors on software and hardware environments. In line with the focus on models, QoSA 2008 also focused on the automated transformation of models using model-driven development techniques.

QoSA 2008 received 36 submissions. From these submissions, 13 were accepted as long papers after a careful peer-review process followed by an online Program Committee discussion. This resulted in an acceptance rate of 36%. The selected technical papers are published in this volume, together with a written version of the invited talk by Carlo Ghezzi. For the second time, QoSA was held as part of the conference series Federated Events on Component-Based Software Engineering and Software Architecture (COMPARCH). The federated events were QoSA 2008, the 11th International Symposium on Component-Based Software Engineering (CBSE 2008) and the Workshop on

Component-Based High-Performance Computing (CBHPC 2008). Together with COMPARCH's Industrial Experience Report Track and the co-located Workshop on Component-Oriented Programming (WCOP 2008), COMPARCH provided a broad spectrum of events related to components and architectures. By integrating QoSA's and CBSE's technical programs in COMPARCH 2008, both conferences elaborated their successful collaboration thus demonstrating the close relationship between software architectures and their constituting software components.

Among the many people who contributed to the success of QoSA 2008, we would like to thank the members of the Program Committees for their valuable work during the review process, as well as Carlo Ghezzi, Michael Stal, Thomas Dreier, Philippe Kruchten, and Florian Matthes for their COMPARCH keynotes. Additionally, we thank Alfred Hofmann from Springer for his support in reviewing and publishing the proceedings volume and Henning Groenda for his support in organizing QoSA and preparing this LNCS volume. The QoSA organizers would also like to thank the supporters of COMPARCH 2008, namely 1&1 Internet AG and sd&m AG. This conference would not have been possible without the commitment of all these people and our supporters.

July 2008

Steffen Becker
Frantisek Plasil
Ralf Reussner

Organization

QoSA 2008 (Part of COMPARCH 2008)

General Chair

Ralf Reussner, University of Karlsruhe (TH), Germany

Program Committee Chairs

Steffen Becker, Forschungszentrum Informatik (FZI), Germany
Frantisek Plasil, Charles University, Czech Republic

Program Committee

Colin Atkinson, University of Mannheim, Germany
Achim Baier, itemis AG, Germany
Len Bass, Software Engineering Institute, USA
Jan Bosch, Intuit, USA
Jeremy Bradley, Imperial College London, UK
Vincenzo Grassi, University of Rome "Tor Vergata", Italy
Wilhelm Hasselbring, University of Oldenburg / OFFIS, Germany
Christine Hofmeister, Lehigh University, USA
Jean-Marc Jezequel, University of Rennes / INRIA, France
Samuel Kounev, University of Cambridge, UK
Patricia Lago, Vrije Universiteit, The Netherlands
Nicole Levy, University of Versailles, France
Markus Lumpe, Swinburne University, Australia
Eric Madelaine, Inria, France
Tomi Mannisto, Helsinki University of Technology, Finland
Nenad Medvidovic, University of Southern California, USA
Raffaela Mirandola, Politecnico di Milano, Italy
Robert Nord, Software Engineering Institute, USA
Dorina Petriu, Carleton University, Canada
Iman Poernomo, King's College, UK
Sasikumar Punnekkat, Mälardalen University, Sweden
Andreas Rausch, Clausthal University of Technology, Germany
Matthias Riebisch, Technical University of Ilmenau, Germany
Roshanak Roshandel, Seattle University, USA
Bernhard Rumpe, University of Technology Braunschweig, Germany
Jean-Guy Schneider, Swinburne University, Australia
Michael Stal, Siemens, Germany
Petr Tuma, Charles University, Czech Republic
Axel Uhl, SAP, Germany
Kurt Wallnau, Software Engineering Institute, USA

Wolfgang Weck, Independent Software Architect, Switzerland
Murray Woodside, Carlton University, Canada
Steffen Zschaler, Technical University of Dresden, Germany

Co-reviewers

Huseyin Aysan, Mälardalen University, Sweden
Franz Brosch, Forschungszentrum Informatik (FZI), Germany
Antonio Cansado, Inria, France
Virginie Contes, Inria, France
Fabrice Huet, Inria, France
Christoph Rathfelder, Forschungszentrum Informatik (FZI), Germany

Major Supporters

1&1 Internet AG, Karlsruhe, Germany
sd&m AG, Munich, Germany

Table of Contents

Keynote

Rethinking the Use of Models in Software Architecture 1
Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola

Architectural Design Decisions and Influence on Quality

Design Reasoning Improves Software Design Quality 28
Antony Tang, Minh H. Tran, Jun Han, and Hans van Vliet

A Tool to Visualize Architectural Design Decisions 43
Larix Lee and Philippe Kruchten

Style-Based Model Transformation for Early Extrafunctional Analysis of Distributed Systems 55
Julien Mallet and Siegfried Rouvrais

Architecture and Components / Reasoning about Components

Carmen: Software Component Model Checker 71
Aleš Plšek and Jiří Adámek

MOSES: MOdeling Software and platform architEcture in UML 2 for Simulation-based performance analysis 86
Vittorio Cortellessa, Pierluigi Pierini, Romina Spalazzese, and Alessio Vianale

Designing the Enterprise Architecture Function 103
Bas van der Raadt and Hans van Vliet

Quality Prediction of Service Compositions through Probabilistic Model Checking 119
Stefano Gallotti, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli

Models and Prediction

Model-Driven Performance Analysis 135
Gabriel A. Moreno and Paulo Merson

Architectural Specification and Static Analyses of Contractual Application Properties 152
Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien

Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant 171
Andres Diaz-Pace, Hyunwoo Kim, Len Bass, Phil Bianco, and Felix Bachmann

Architecture Evaluation Processes

Middleware Architecture Evaluation for Dependable Self-managing Systems 189
Yan Liu, Muhammad Ali Babar, and Ian Gorton

Comprehensive Architecture Evaluation and Management in Large Software-Systems 205
Frank Salger, Marcel Bennicke, Gregor Engels, and Claus Lewerentz

Sharing the Architectural Knowledge of Quantitative Analysis 220
Anton Jansen, Tjaard de Vries, Paris Avgeriou, and Martijn van Veelen

Author Index 235

Rethinking the Use of Models in Software Architecture

Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola

Politecnico di Milano, Dipartimento di Elettronica e Informazione,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{ardagna,ghezzi,mirandola}@elet.polimi.it

Abstract. Models play a central role in software engineering. They may be used to reason about requirements, to identify possible missing parts or conflicts. They may be used at design time to analyze the effects and trade-offs of different architectural choices before starting an implementation, anticipating the discovery of possible defects that might be uncovered at later stages, when they might be difficult or very expensive to remove. They may also be used at run time to support continuous monitoring of compliance of the running system with respect to the desired model. This paper focuses on models that support reasoning about non-functional system properties — namely, performance and reliability. It provides a taxonomy, which tries to capture the main facets that are needed to understand, choose, and use models appropriately in the various phases of software development and operation. The paper also focuses on the roundtrip from models to reality and back. The forward path is followed in model-driven development. The backward path is instead meant to enable model calibration, with the goal of building adequate abstractions, which reflect reality and its properties in a faithful manner. Calibration may be required because of flaws in the initial model or in the process that derived the implementation, or because of changes that occurred in the environment or in the requirements. This leads to the idea that models should continue to live at run time, on-line with the running implementation. Calibrated models may drive the necessary dynamic changes that may support self-adaptation of the implemented system.

1 Introduction

Engineers use models to design artifacts. Civil engineers define the model of a bridge before constructing it. The model may be an abstract mathematical description consisting of equations that describe the static behavior of the bridge. The abstraction may view the bridge as a rigid body and may ignore deformation. By applying standard mathematical reasoning, the engineer may estimate the forces at junction points due to certain loads. By refining the model and taking into account the material used to build it, it is possible to estimate whether a certain load will be sustained. Further, models allow the engineer to reason about dynamic properties, such as the effect of changing loads, wind, or even earthquake, to foresee the behavior of the system-to-be.

Another type of widely used approach to modeling requires building a mockup of the artifact and experimentally evaluate its properties to estimate the behaviors of the future artifact. For example, wind tunnels, inspired by the achievements of Gustave Eiffel in Paris, at the beginning of the twentieth century, are now widely widespread worldwide to support aerodynamic test of mockups of vehicles or aerospace equipment.

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 1–27, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Models (and abstraction) are key to computing and play an even more important and pervasive role in software engineering [42]. In the end, the software written in a high-level language is a model of the executable code, which is generated automatically by the compiler. In addition, every application embodies a more or less explicit model of the real world with which the automated software system will interact. Indeed, the relevant part of the real world should be understood and modeled in the requirements acquisition phase, before starting the design of the system. System design also relies on models. For example, the *entity-relationship* (ER) model [22] is widely used by database designers to reason about the logical structure of data, before defining the tables used in the implementation.

A wealth of models has been proposed over time to support software engineers. They vary according to the level of formality and precision, the aspects they intend to describe, and the kind of reasoning they support. For example, a *use case* [18] provides a rather high-level and loose description of the possible interactions of actors with the system, while a *labelled transition system* [43] may provide a formal description of the dynamic behavior of a software component. The former is normally used in the requirements phase. The latter may also be used in the design phase, to reason about the dynamic behavior of a tentative high-level solution.

The most striking aspect of models in software engineering, as opposed to models in other traditional engineering fields, is that models and final artifacts are made of the same fabric: they are both *software*. This is why *model transformations* may be conceived to support the transition from model to system. Such transformations may be more or less automatic, but in any case they may be stated as precisely defined software manipulation actions, rather than informal design steps. *Model-driven development* (MDD) [7] has recently emerged as a discipline that tries to systematize model-based software development.

The role of models is rather well understood in the initial development of an application. Models represent abstractions of the system-to-be. They may abstract both the real world in which the systems will be embedded and the systems themselves. They may be used to support the engineers in the design choices they have to make. Because a model-driven development process follows a path from abstract to concrete layers, different design models are developed at each stage. In addition, at each stage, several models may be produced to support different views of the system and reasoning on different kinds of properties.

In this paper, we focus on models that support analysis of *non-functional* properties, namely *reliability* and *performance*. The models needed to reason about such properties (*analysis-oriented models*) differ from the more conventional models used by software engineers to express their design choices (*design-oriented models*), although the former may be obtained from the latter by means of certain systematic transformation patterns.

Because the models are abstraction of the real systems, the results of analysis approximate the results that will hold when the implemented system will be deployed and executed in the real world. The approximations are due both to the abstractions made on the software to-be and on the environment in which the software will be embedded. It is therefore possible that a model-driven development, because of inaccurate or wrong modeling assumptions, leads to an implementation that is not satisfactory. In particular,

the environment abstraction used at design time may prove to be inadequate: it may reflect only partially what happens in reality. For example, certain user behaviors that were assumed as occasional during design may prove to be very common. As another example, the performance or reliability profile of certain resources used by the system in practice may differ from the figures assumed at design time. Because of the inaccuracy of the data, the initial model may be flawed, and the implementation derived from the model needs to be evolved. According to the principles of model-driven development, this in turn requires calibrating the model, and then re-generating the implementation.

Recent trends in software development indicate that software increasingly needs to evolve dynamically, to adapt to changing environmental conditions that may occur as the system is running. To support the roundtrip from model to implementation and back, which is mandated by the principles of model-driven development, it is necessary that the model is kept alive at run time, and that the necessary adaptations are driven by the modifications in the model. It is also necessary that monitoring facilities should be in place to support the detection of changes in the environment which drive changes in the model.

The two aforementioned points lead to another, increasingly important, role of models, which extends from design time to also cover run time. Models must continue to exist after the end of development. They should be kept alive at run time to support the tuning of the model that may be necessary in both previous cases. And the tuning of the model may lead to a —more or less— automatic dynamic adjustments of the implementation.

The rest of the paper is organized as follows. The model-driven approach and the need for analysis-oriented models will be discussed in Section 2. Because many design-oriented and analysis-oriented models exist and it is often unclear which one to use at different stages and for different purposes, Section 3 provides a preview of a conceptual map we propose to provide guidance for software engineers. The conceptual map defines a taxonomy of models. Sections 4 through 7 justify our findings by reviewing the most relevant models supporting analysis of non-functional properties. They also discuss the stage at which each specific model is most suitable and how the analysis models may be derived systematically from the design models available at that stage. Section 8 wraps up the discussion by focusing on comparison criteria. Finally, Section 9 provides conclusions and outlines some relevant research directions, focusing in particular on run time models and run time verification.

2 Software Quality Model Driven Framework

The goal of MDD is twofold. On the one side, it aims at deriving (automatically or semi-automatically) a software implementation, starting from high-level models of the system and applying model transformation rules which refine high-level descriptions into more concrete and specific models. On the other, it aims at supporting reasoning activities on the high-level models. Through an early analysis of non-functional properties, such as performance and reliability, the software engineer can evaluate the impact of the different design choices or candidate system architectures, before they are reified into runnable code.

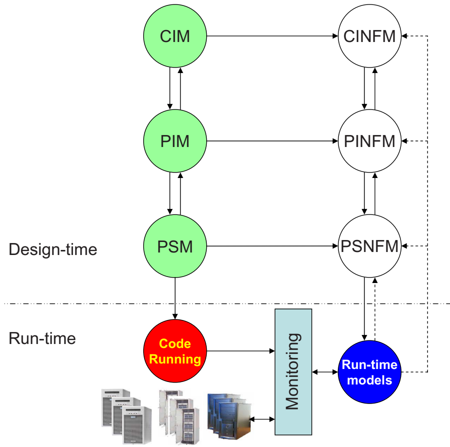


Fig. 1. Reference Framework

MDD has been actively investigated in the last years, and some taxonomies of model transformation approaches have been defined to help a software developer choosing the method that is best suited for his or her needs [28, 58, 76].

The MDD framework includes three main abstraction layers [59], shown on the left-hand side of Figure 1:

- The Computational Independent Model (CIM), whose goal is to describe the main functionalities to be implemented by the application (e.g., in the UML 2.0 framework, a use case diagram is an example of a CIM);
- The Platform Independent Model (PIM), which describes in more detail the logic flow of the system and, possibly, the user interactions, in order to achieve the functionalities (e.g., a sequence or an activity diagram of UML 2.0);
- The Platform Specific Model (PSM), which describes how application components are mapped onto the system physical resources (e.g., a UML 2.0 deployment diagram).

To reason about non-functional quality attributes of a software system, it is necessary to transform the aforementioned "design-oriented" models of the software system into "analysis-oriented" models that support the desired analysis methods [54]. In the specific case of performance and reliability attributes, the analysis-oriented model may provide a probabilistic description of the system that evolves in time and space. Possible examples are queuing networks or different kinds of Markovian models. Corresponding to the three MDD layers listed above (CIM, PIM, and PSM), in [25] three additional classes of analysis-oriented models have been identified (CINFM, PINFM, and PSNFM), where:

- CINFM (Computation Independent Non-Functional Model) represents the requirements and constraints related to a NF aspect (i.e., performance and reliability). An example of a CINFM may be a Use Case Diagram augmented with reliability requirements.

- PINFM (Platform Independent Non-Functional Model) represents the logic of the system along with estimates of NF characteristics, such as the amount of resources that the logic needs to be executed. An example of a PINFM can be a Markov Model derived from the corresponding PIM annotated with non-functional aspects.
- PSNFM (Platform Specific Non-Functional Model) contains variables and parameters that represent the software structure and dynamics, as well as the platform and the environment in which the software will be deployed. A characterization of the platform must include data on the underlying hardware architecture, such as the CPU speed or the failure probability of a network connection. An example of PSNFM is provided by a queueing network model derived from the corresponding PSM and including performance parameters.

The idea of non-functional analysis-oriented models associated with the corresponding design models proposed by [25] starts from the canonical view of the MDD approach [59] and embeds quantitative models to evaluate performance and reliability of the final application to be deployed. A conceptual extension of this framework is summarized in Figure 1. The figure shows the role of models not only at design time, which was discussed so far, but also at run time. At run time, a monitor collects the results of execution of the software system in the target environment. The data gathered by the monitor are checked against the model, to verify that non-functional requirements are satisfied. As we mentioned, this is necessary to close the loop between design time modelling and run time execution.

The transformations among models described in Figure 1 can be classified as either *horizontal* or *vertical transformations*. Following the canonical MDD framework, horizontal transformations (arrows from left to right) yield a target model at the same level of abstraction as the source model. The transformation adds annotations about non functional requirements to support reasoning about performance and reliability. A detailed overview of transformation approaches is given in [28, 54, 76].

The focus of horizontal transformation is on specifying the information that is lacking in the software architecture description but is crucial for the quantitative non-functional analysis (examples of this information can be: number of invocations of a component within a certain scenario, probability of executions, etc). Some efforts in this direction can be found for example, in the introduction of UML profiles aiming at represent performance and QoS information [60, 61].

Vertical transformations produce a target model at a different abstraction level. Vertical transformation can abstract or refine the source model (down or up arrows in Figure 1). A vertical transformation of a non-functional model is used to refine a higher level model, to improve the accuracy of the performance/reliability metrics.

According to Figure 1, the goal of performance and reliability models at design time is to evaluate the impact of multiple architectural choices by predicting the corresponding run time behaviors. Instead, at run time the goal is: (i) to track the real figures of system performance and reliability and check them against high-level requirements and constraints guarantees, (ii) to trigger re-configuration mechanisms, and (iii) to fine-tune non functional parameters used as input to higher level analysis models.

It is increasingly important that software systems behave in an *autonomic* manner; i.e., they can automatically reconfigure and adapt themselves in order to match changing execution conditions and/or user requirements. This issue is taken up later in this paper.

3 Towards a Conceptual Map of Models for Quality Evaluation

A substantial amount of research has been devoted to devising performance and reliability prediction techniques for software systems (see for example [1, 72]). One of the goals of this paper is to provide a conceptual map where the most prominent techniques can be positioned. This may help software engineers understand where and how the different techniques may be used. The conceptual map produced by our findings is summarized in Table 1. Hereafter we anticipate a few general comments. The background material on models and analysis that led to the conceptual map is discussed in Sections 4 through 7. For each model family we will highlight the accuracy provided, the layer at which the models are usually adopted, and finally the availability of tools for automatic model derivation.

Table 1 shows that models can be classified according to several dimensions. First, they are classified according to the target quality of the analysis, i.e. *performance* or *reliability*. Models are also classified as design-time or run-time models. Design-time models are further classified according to the reference framework of Figure 1; that is, as CINFM, PINFM, or PSNFM. Run-time models are instead classified according to the different time scales they can deal with. Some performance models are based on the assumption that the system is in a *steady state*. Hence, they are appropriate for medium to long range time reasoning (e.g., half an hour) [6, 66]. In the following, they will be called *long-term models*. Vice versa, some more recent contributions from control-theory research area can accurately model the system at a finer time grain (e.g. seconds) and can also deal with system transients. They will be called *short-term models*.

The goal of performance analysis is to determine metrics, such as the *system throughput* (usually denoted with X, i.e., the rate at which requests are executed by the system), *response time* (usually denoted with R, i.e., the average time needed by the system to process users' requests), or system *utilization* (usually denoted with U, i.e., the proportion of time the system is busy). Usually models support evaluation of the average values of performance metrics, but some analyses could also provide percentile

Table 1. Quantitative Models Conceptual Mapping

		Performance	Reliability	Design Time		Run Time	
Model Family	Model			PINFM	PSNFM	Long Term	Short Term
Queueing Models	Bound Analysis	X		X	X	X	X
	Product form	X		X	X	X	
	Non-product form	X		X	X		
	LQN	X		X	X		
Markov Models	Discrete Time Markov Chains	X	X	X	X		
	Continuous Time Markov Chains	X	X	X	X		
	Markov Decision Processes	X	X	X	X		
	Stochastic Model Checking	X	X	X	X		
Simulation Approaches	Simulation Models	X	X	X	X		
Control-Oriented Models	LTI	X				X	X
	LPV	X					X

distributions. We recall that the percentile distribution of the response time R_α is defined as the value of response time such that the probability of getting a value less than or equal to R_α is $\alpha/100$. In other words, $P(R \leq R_\alpha) = \alpha/100$. The evaluation of the percentile distribution is very important since nowadays service level agreements (SLAs) between providers and their customers are typically formulated in terms of them. For example, "95% of requests have to experience a response time lower than, e.g., 5 seconds".

Several definitions of dependable systems and dependability metrics have been provided in the literature [8]. We assume that system reliability at a given time t is defined as the probability that a system has always been working properly during the time interval $(0, t)$.

4 Queueing Network Models

Queueing network (QN) models [17, 49] are a mathematical modeling approach in which a software system is represented as a collection of: (i) *service centers*, which model system resources, and (ii) *customers*, which represent system users or "requests" and move from one service center to another one. The customers' competition to access resources corresponds to queueing into the service centers.

The simplest queueing network model includes a single service center (see Figure 2) which is composed of a single queue and a single server: the queue models a flow of customers or requests which enter the system, wait in the queue if the system is busy serving other requests, obtain the service, and then depart. Note that, at any time instant only one customer or request is obtaining the service. Single service centers can be described by two parameters: the requests *arrival rate*, usually denoted by λ, and the average *service time* S, i.e., the average time required by the server in order to execute a single request. The maximum service rate is usually indicated with μ and is defined as $\mu = 1/S$. Given the request arrival rate and the requests service time, QN theory allows evaluating the average value of performance metrics by solving simple equations.

In real systems, requests need to access multiple resources in order to be executed; accordingly, in the model they go through more than a single queue. A QN includes several service centers and can be modeled as a directed graph were each node represent the k-th service center, while arcs represent transitions of customers/requests from one service center to the next. The set of nodes and arcs determines the network topology.

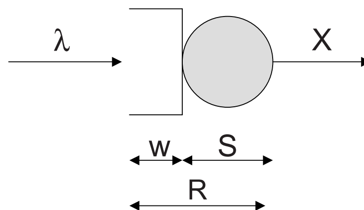


Fig. 2. Single Service Center Model. W indicates the requests' waiting time, i.e. the average time spent by requests in the queue.

Product-form QN. One of the most important result of queueing network theory is the *BCMP theorem* (by Baskett, Chandy, Muntz, and Palacios-Gomez) which, under various assumptions (see [12] for further details), shows that performance of a software system is independent of network topology and requests routing but depends only on the requests arrival rate and on the requests *demanding time* D_k, i.e., the average overall time requeried to complete a request at service center k. The average number of time a request is served at the k-th service center is defined as the number of visits V_k, and it holds $D_k = V_k \cdot S_k$.

In time sharing operating systems, as an example, the average service time is the operating system time slice, while the demanding time is the overall average CPU time required for a request execution. The number of visits is the average number of accesses to the CPU performed by a single request.

Queueing networks satisfying the BCMP theorem assumptions are an important class of models also known as *separable queueing networks* or *product-form models*. The name “separable” comes from the fact that each service center can be separated from the rest of the network, and its solution can be evaluated in isolation. The solution of the entire network can then be formed by combining these separate results [12, 31, 38, 49][1]. Such models are the only ones that can be solved efficiently, while the solution time of the equations governing non-product form queueing network grows exponentially with the size of the network. Hence, in practical situations the time required for the solution of non-product form networks becomes prohibitive and approximate solutions have to be adopted.

Open and Closed Models. Queueing models can be classified as *open* or *closed* models. In open models customers can arrive and depart and the number of customers in the system cannot be determined a-priori (and, possibly, it can be infinite). The single service center system in Figure 2 is an open model. Vice versa, in closed models the number of customers in the system is a constant, i.e., no new customer can enter the system, and no customer can depart (see Figure 3). While open models are characterized by the requests arrival rate λ, a closed model can be described by the average number of users in the system N and by their *think time* Z, i.e., the average time that each customer “spends thinking” between interactions (e.g., reading a Web page before clicking the next link). Customers in closed queue models are represented as delay centers (circles in Figure 3).

Usually Service Oriented Architecture (SOA) based systems are modeled by means of open models [6]. Closed systems can be adopted, for example, to model an Intranet with a fixed number of users or an Internet application when the concurrent number of sessions is kept constant (e.g., as a result of an overload protection mechanism [77]).

Single and Multi-Class Models. Finally, queueing models can be classified as *single class* and *multi-class* models. In single class models, customers have the same behavior; vice versa, in multi-class models, customers behave differently and are mapped into multiple-classes. Each class c can be characterized by different values of demanding

[1] The name “product-form” comes from the fact that the stationary state distribution of the queueing network can be expressed as the product of the distributions of the single queues and avoids the numerical solution of the underlying Markov chain.

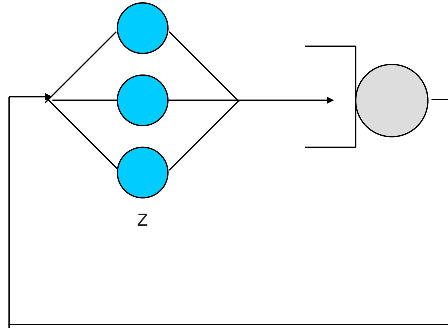


Fig. 3. Closed Model Example

time $D_{c,k}$ at the k-th service center, different arrival rate λ_c in open models, or number of users N_c and think time Z_c in closed models. Customers in each class are statistically indistinguishable. Queueing network theory allows determining performance metrics (i.e. response times, utilizations, etc.) on a per-class basis or on an aggregated basis for the whole system.

Layered QN. Layered queueing networks (LQN) models were developed as an extension of QN [69, 78]. A LQN model is an acyclic graph, with nodes representing software entities and hardware devices, and arcs denoting service requests. The software entities are also known as *tasks*. Each kind of service offered by a LQN task is modeled as a so-called *entry*, which can be further described by two alternative ways, i.e. using either *activities* or *phases*. Activities allow describing more complex behaviors, e.g. with control structures such as forks and joins and loops. Phases are used for notational convenience to specify activities that are sequences of several steps. The advantage of LQN is to introduce also software elements as resources and hence they can capture the contention to access software components.

Solution Techniques. After modelling a software system as a queueing network, the model has to be evaluated in order to determine quantitatively the performance metrics.

A first step in the evaluation can be achieved by determining the system bounds; specifically, upper and lower bounds on system throughput and response time can be computed as functions of the system workload intensity (number or arrival rate of customers). Bounds usually require a very little computational effort, especially for the single class case [21, 40].

More accurate results can be achieved by solving the equations which govern the QN behavior. Solution techniques can be classified in *analytical methods* (which can be *exact* or *approximate*) and *simulation methods* (see Section 6). Exact analytical methods can determine functional relations between model parameters (i.e., request arrival rate λ_c, number of customers N_c and think time Z_c, and requests demanding times $D_{c,k}$) and performance metrics. The analytical solution of open models system is very

simple even for multiple class models and yields the average value of the performance metrics. The exact solution of single class closed models is known as the *Mean Value Analysis* (MVA) algorithm and has a linear time complexity with the number of customers and the number of service centers of the models. The MVA algorithm has been extendend also to multiple classes, but the time complexity is non-polynomial with the number of customers or with the number of service centers and classes [20, 49]. Hence, large closed models are solved by recurring to approximate solutions, which are mainly iterative methods and can determine approximate results in a reasonable time.

Analytical solutions can determine the average values of the performance metrics (e.g., average response time, utilization, etc.) or, in some cases, also the percentile distribution of the metric of interest. Determining the percentile distribution of large systems is usually complex even for product-form networks. Indeed, while the mean value of the response time of a request that goes through multiple queues is given by the sum of the average response time obtained locally at the individual queues, the aggregated probability distribution is given by the convolution of the probability distribution of the individual queues. The analytical expression of the percentile distribution becomes complicated for large system (most of the studies provided in the literature are limited to *tandem queues*, i.e. queueing networks including two service centers [17, 39]). For this reason, some approximate formulas have been introduced. Markov's Inequality [41, 62] provides an *upper-bound* on the probability that the response time exceeds the threshold R_α. This upper-bound depends only on the average response time $E[R]$, and can be computed as $P(R \geq R_\alpha) \leq E[R]/R_\alpha$. However, Markov's inequality is known for providing somewhat loose upper-bounds. Chebyshev's inequality [41] provides a tighter upper-bound based on estimates of response time variance, $Var[R]$, in addition to estimates of the average response time $E[R]$. Chebyshev's inequality is given by $P(R \geq R_\alpha) \leq \frac{Var[R]}{(R_\alpha - E[R])^2}$.

Accuracy. Although the BCMP theorem assumptions almost never hold in real software systems, experience has shown that the accuracy of queueing network models is extremely robust with respect to violations of these assumptions [49]. Typically the deviations between the measured values in a real system and the ones obtained by the models come from an inaccurate estimate of parameter values for service demands or workload intensities. The only important exceptions are the cases where the limitations on the structure of the model imposed by the BCMP theorem inhibit the representation of aspects affecting performance (e.g., requests routing, blocking conditions, service time distributions, etc.). In these cases, basic product-form solutions can be extended for example by introducing burstiness parameters [56] or iteratively solved in order to provide more accurate results. In other words, separable models are also the basic building blocks which can be adopted for the construction of more detailed models. As an example, the solution technique for LQN is based on this idea.

In terms of accuracy, a large body of experience [49] indicates that queueing network models can be expected to be accurate in the range 5-10% for utilization and throughput estimates and within 10 to 30% for response time. Bounding techniques are usually characterized by a worst case estimate with respect to the exact analytical solution in the range 15-35% for system throughput, while for response time usually the bounds are more inaccurate. Recently in [21] geometric bounds have been proposed, a new family

of fast and accurate family of bounds for closed models where the bounding error for system throughput is in the worst case within 5-13%.

Approximate MVA algorithms for multi-class closed models provide results typically within a few percent of the exact solution for throughput and utilization, and within 10% for queue lengths and response time [49].

With respect to the use of product-form QNs as run time models, they are usually emebedded in an optimization framework and the accuracy has also been evaluated in terms of variation of the objective function to be optimized. In the preliminary work presented in [4, 27], authors have analyzed the variation of the objective function which can be obtained by adopting different solution techniques in the same autonomic framework. Results depend on the optimization framework and can not be easily generalized; anyway authors have shown that Markov and Chebichev's inequalities lead to conservative allocation decisions that are up to 20% less profitable than the optimal solution obtained by adopting an exact tandem queue model.

Model Adoption. Queueing network models can be used at design time to perform performance analyses. QN bounds can be used to obtain performance information at design time (both at PIM and at the PSM level) following the method proposed in [55]. [19, 57] use product form queueing network at the PINFM layer for the evaluation of the response time of BPEL business processes modelled by activity diagrams. Queueing models are more frequently adopted at the PSNFM layer for the capacity planning of the target hardware platform at design time.

As discussed above, product form models are also used at run time for the implementation of self-configuring autonomic computing systems for long-term control horizon.

Tools for Model Derivation. In the literature there exists a quite large set of methodologies which propose transformation techniques to derive QN-based models (both product and non-product or LQN) starting from software models. Some of the proposed methods are reviewed in [1, 10, 13].

Bounds are automatically derived starting from a description of the system behavior (PIM), and the transformation is implemented with *ad-hoc algorithms* that use imperative programming languages. Always at design time, product and non-product QN and LQN can be automatically obtained by using automatic transformations. Examples can be found in [53] for PINFMs, or in [11, 14, 26, 35, 36, 78] for the derivation of PSNFMs.

As a general consideration, in these approaches the transformations are often implemented with *ad-hoc algorithms* that use imperative programming languages. However in several transformation methodologies and tools it is possible to devise a common underlying application pattern. In this group of transformations, the source architectural model is represented by a set of UML diagrams. These diagrams are annotated with ad-hoc or standard performance annotations and then exported in the underlying XMI/XML format. The transformation is then defined from the XML document of the source model to an XML model defining the target performance model. The transformation language is often JAVA or a similar imperative language. However, in some cases the transformations are defined using XSLT or graph transformation rules [36].

5 Markov Models

This section illustrates the main Markov Models adopted in Software Engineering for prediction of non-functional properties. In this context we consider non-functional properties (performance and reliability). Markov models are stochastic processes defined as state-transition systems augmented with probabilities. Formally, a stochastic process is a collection of random variables $X(t), t \in T$ all defined on a common sample (probability) space. The $X(t)$ is the state while (time) t is the index that is a member of set T (which can be discrete or continuous). In Markov models [17, 41], states represent possible configurations of the system being modeled. Transitions among states occur at discrete or continuous time-steps and probability of making transitions is given by probability distributions. Markov property characterizes these models: it means that, given the present state, future states are independent of the past. In other words, the description of the present state fully captures all the information that could influence the future evolution of the process.

The following sections illustrate three Markov models: Discrete Time Markov Models, Markov Decision Processes, and Continuous Markov Chains.

DTMC. Discrete Time Markov Chains are the simplest Markovian model where transitions between states happen at discrete time steps.

Formally a DTMC is a 4-uple $< S, \overline{s}, P, L >$ where:

- S is the finite set of states.
- $\overline{s}$ is the initial state.
- $P : S \times S \rightarrow [0, 1]$ is the transition function, where $P(s, s')$ is the probability of reach s' from s.
- $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$
- $L \rightarrow 2^{AP}$ is the labeling function, it assign at each $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ holding in s.

A DTMC evolves starting from the initial state executing a transition at each discrete time instant. Being at time i in a state s, at time $i + 1$ the model will be in s' with probability $P(s, s')$. The transition can take place only if $P(s, s') > 0$.

When software engineers adopt DTMCs they must specify the set of states S and transitions with an associated probability distribution (probabilities of outgoing transitions in every state must sum up to one). These probability values are numerical parameters and represent, for example, the failure rate of a system component.

Figure 4 shows a simple DTMC where:

S =(a,b,c,d) and $\overline{s}$ = a
AP = (Init, Connect, Fail, Success)
L(a)=Init, L(b)=Connect, L(c)=Fail, L(d)=Success

MDP. Markov Decision Processes [64] are an extension of DTMC that allow multiple probabilistic behaviors to be specified as output from a state, considering these behaviors non-deterministically. First of all, the particular behavior to follow is selected in every state, consequently, the next state is selected using the probabilities described in

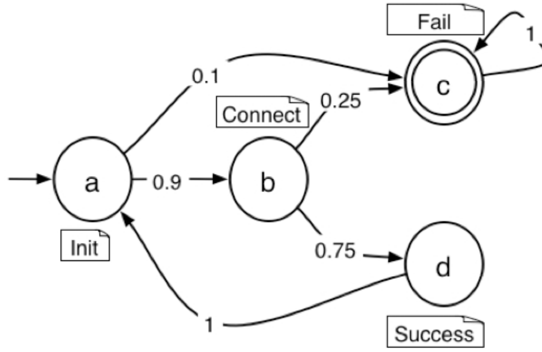


Fig. 4. An example of DTMC, c is a final state

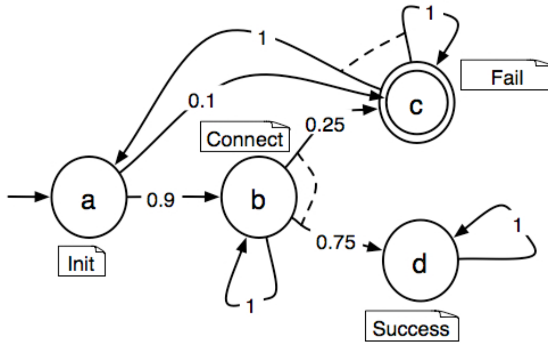


Fig. 5. An example of MDP

the previously selected behavior. Like DTMCs there is a discrete set of states representing possible configurations of the system being modeled and transitions between states occur in discrete time-steps, but in each state there is also a nondeterministic choice between several discrete probability distributions over successor states.

Formally a MDP is a 4-uple $< S, \overline{s}, STEPS, L >$ where:

- S is the finite set of states.
- $\overline{s}$ is the initial state.
- $STEPS : S \rightarrow 2^{ACT \times Dist(S)}$ is the transition function, where ACT represents the set of available non-deterministic transitions and $Dist(S)$ is the set of probability values over the set S
- $L \rightarrow 2^{AP}$ is the labeling function, it assigns at each $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ holding in s.
- for all actions a outgoing from a state s, $\sum_{s' \in S} P_a(s') = 1$

As for DTMC, transitions happen at discrete time instants. Figure 5 shows a simple MDP where:

S =(a,b,c,d) and $\overline{s}$=a
AP = (Init, Connect, Fail, Success)

L(a)=Init, L(b)=Connect, L(c)=Fail, L(d)=Success
Steps(a) = (0.9 →b, 0,1→c)
Steps(b) = (0.25 →c, 0,75→d), (1 →b)
Steps(c) = (1 →c), (1 →c)
Steps(d) = (1 →d)

CTMC. Continuous Time Markov Chains are another extension of DTMC. The model is similar to the DTMC one, but the temporal domain is continuous. This means that the time in which a transition occur is not fixed, but depends on some parameter of the model. The model is specified as a DTMC by means of state and probabilistic transition, but the value associated to the outgoing transition from a state is intended not as a probability but as a parameter of an exponential probability distribution. The probability value thus depends on the time at which the distribution is evaluated. Formally a CTMC is a 4-uple $< S, \overline{s}, R, L >$ where:

- S is the finite set of states.
- $\overline{s}$ is the initial state.
- $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is the transition function, where $R(s, s')$ is the coefficient of the distribution function $1 - e^{-R(s,s')\cdot t}$ standing for the evolving probability from s in s' within t time instants.
- $L \rightarrow 2^{AP}$ is the labeling function, it assign at each $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ holding in s.

Whenever in a state two or more output transitions are defined, a race-condition occurs. More than one transition is enabled and the output is selected with the probabilities computed given the actual time instant. First of all, the probability of leaving the state is computed as $(1 - e^{E(s)\cdot t})$ where $E(s) = \sum_{s' \in S} R(s, s')$ is the sum of all outgoing coefficients. If the transition is triggered, then the output state probability is computed as the normalized transition rate.

Updating the CTMC model $R(s, s')$ with the normalized value $P(s, s')$ computed as

$$P(s, s') = \begin{cases} \frac{R(s,s')}{E(s)} & \text{if } E(s) \neq 0 \\ 0 & otherwise \end{cases}$$

we obtain a discrete time model called *Embedded DTMC*, which represents the equivalent system at discrete time. This coincides with value of $P(s, s', t)$ that is $\lim_{t\rightarrow\infty} P(s, s', t)$.

Figure 6 shows an example of CTMC representing a generic server and its associated finite queue.

Solving Markovian Models. The solution of Markovian models is based on the evaluation of the stationary probability π of each state of the model which can be obtained by solving a linear set of equations $\pi P = 0$ with the normalizing condition $\sum\limits_{s \in S} \pi_s = 1$.

The main problem of Markov models is the explosion of the number of states when they are used to model real systems. On the other hand, Markov chains are very general since can include as special cases other modelling approaches. As an example, under

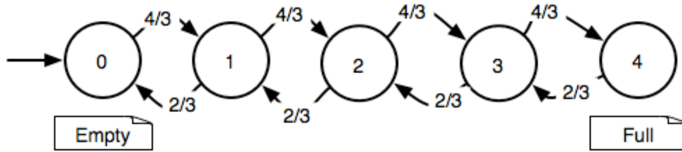


Fig. 6. An example CTMC

the assumptions of the BCMP theorem, the single service center queue of Figure 2, known also as M/M/1 queue, can be modelled by a CTMC with an infinite number of states (each state representing the number of customers in the system), and the closed formulas computed by QN theory are obtained by determining the probability stationary conditions of the underlying Markov chains. Furthermore, CTMC can be used to compute the solution of non-product form QNs. Other models that can be reduced to Markov models are the families of Stochastic Petri Nets (SPN) and Stochastic Process Algebras (SPAs) [24]. SPNs are an extension of Petri Nets and allow modelling synchronization in concurrent systems [23]. SPNs are able to model synchronization but can not model competition in the use of system resources and, with respect to this characteristic, are complementary to QN models. The advantage of SPAs is that they allow the integration of functional and non-functional aspects in a unified model. A further class of analysis supported by Markovian models is stochastic model checking, which is discussed next.

Stochastic Model Checking. Stochastic model checking is an automatic procedure for establishing if a desired property holds in a probabilistic system model. Conventional model checkers start from a description of a model and a specification (using a state-transition system and a formula in some temporal logic, respectively) and return a boolean value, indicating whether or not the model satisfies the specification. In the case of probabilistic model checking, the models are probabilistic (obtained as a variant of Markov chains) and they add a probability to the transitions between states. In this way it is possible to calculate the likelihood of the occurrence of certain events during the execution of the system. This, in turn, allows quantitative analysis about the system, in addition to the qualitative statements made by conventional model checking. Probabilities are modeled via probabilistic operators that extend conventional (timed or untimed) temporal logic (see next).

The key point of probabilistic model checking is the ability to combine probabilistic analysis and conventional model checking in a single tool. The first extension of model checking algorithms to probabilistic systems was proposed in the 1980s. However, work on implementation and tools did not begin until recently, when the field of model checking matured [46, 47]. Probabilistic model checking draws on conventional model checking, since it relies on reachability analysis of the underlying transition system, but must also entail the calculation of the actual likelihoods through appropriate numerical methods, such as those employed in performance analysis tools [46, 47].

Available model checker tools are mainly based on two probabilistic temporal logics, called Probabilistic Computation Tree Logic (PCTL) [37] and Continuous Stochastic Logic (CSL) [9].

PCTL. Probabilistic Computation Tree Logic (PCTL), adds to CTL* the P operator to specify the probability that satisfies a formula. PCTL syntax:

- state formula
  $\Phi := true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid P_{\sim p}[\Psi]$.
- path formula
  $\Psi :=$X$\Phi \mid \Phi_1$U$^{\leq k}\Phi_2 \mid \Phi_1U\Phi_2$.

where $p \in [0,1]$ is a probability, $k \in \mathbb{N}$ and $\sim \in \{<, >, \leq, \geq\}$. X, U$^{\leq k}$ and U symbols stands respectively for: next state, bounded until and unbounded until.

A PCTL formula is always a state formula; a path formula is allowed only in a $P[]$ operator.

PCTL semantics: Given $s \models_M a$ as a holds in s, semantic of a PCTL formula over a DTMC M, is as following:

$s \models_M true$	holds in each state $s \in S$
$s \models_M a$	holds iif $a \in L(s)$
$s \models_M \Phi_1 \wedge \Phi_2$	holds iif $s \models_M \Phi_1 \wedge s \models_M \Phi_2$
$s \models_M \neg\Phi$	holds iif $s \not\models_M \Phi$
$s \models_M P_{\sim p}[\Psi]$	holds iif $Pr_s\{\pi \in Path^M(s) \mid \pi \models_M \Psi\} \sim p$
$\pi \models_M$ XΦ	holds iif $\pi[1]$ is defined and $\pi[1] \models_M \Phi$
$\pi \models_M \Phi_1$U$^{\leq k}\Phi_2$	holds iif $\exists\, 0 \leq h \leq k(\pi[k] \models_M \Phi_2 \wedge \forall\, 0 \leq j < h\ (\pi[j] \models_M \Phi_1))$
$\pi \models_M \Phi_1$UΦ_2	holds iif $\exists\, h \geq 0(\pi[k] \models_M \Phi_2 \wedge \forall\, 0 \leq j < h\ (\pi[j] \models_M \Phi_1))$

where π represents a generic path, $\pi[i]$ represents the ith state in the path π, $Pr_s\{\pi \in Path^M(s) \mid \pi \models_M \Psi\}$ indicate the probability, evaluated taking into account all paths starting from s, that Ψ will hold.

PCTL is used with DTMC and MDP models, working at discrete time domain. Using PCTL over a MDP model require to extend the $P[]$ operator with min and max operators. Thus each path formula is evaluated in the best or in the worst case.

CSL. Continuous Stochastic Logic (CSL) deals with time in a continuous way. The temporal domain is extended from $\mathbb{N}$ to $\mathbb{R}_{\geq 0}$. Another operator $S[]$ is introduced, standing for steady-state. CSL sintax:

- state formula
  $\Phi := true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid P_{\sim p}[\Psi] \mid S_{\sim p}[\Phi]$.
- path formula
  $\Psi :=$X$\Phi \mid \Phi_1$U$^I\Phi_2$.

where $p \in [0,1]$ is a probability, $I \in \mathbb{R}_{\geq 0}$ is a non-empty interval and $\sim \in \{<, >, \leq, \geq\}$. X and U^I symbols indicate respectly: next state and bounded until. Being I defined over $\mathbb{R}$ the unbounded until is a special case of bounded until (U$^{[0,\infty]}$).

CSL semantics: Given $s \models_M a$ as a holds in s, semantics of a CSL formula over a CTMC M, is as following:

$s \models_M true$	holds in each state $s \in S$
$s \models_M a$	holds iif $a \in L(s)$
$s \models_M \Phi_1 \wedge \Phi_2$	holds iif $s \models_M \Phi_1 \wedge s \models_M \Phi_2$
$s \models_M \neg\Phi$	holds iif $s \not\models_M \Phi$
$s \models_M P_{\sim p}[\Psi]$	holds iif $Pr_s\{\pi \in Path^M(s) \mid \pi \models_M \Psi\} \sim p$
$s \models_M S_{\sim p}[\Phi]$	holds iif $\lim_{t \to +\infty} Pr_s\{\pi \in Path^M(s) \mid \pi^{@t} \models_M \Phi\} \sim p$
$\pi \models_M \mathrm{X}\Phi$	holds iif $\pi[1]$ is defined and $\pi[1] \models_M \Phi$
$\pi \models_M \Phi_1 \mathrm{U}^I \Phi_2$	holds iif $\exists t \in I(\pi^{@t} \models_M \Phi_2 \wedge \forall t' \in [0,t)(\pi^{@t'} \models_M \Phi_1))$

where π represents a generic path, $\pi[i]$ represents the ith state in the path π, $Pr_s\{\pi \in Path^M(s) \mid \pi \models_M \Psi\}$ indicates the probability, evaluated taking into account all the paths starting from s, that Ψ will holds, $\pi^{@t}$ represents the state over the path π at time t.

Accuracy. Markov models include as a special case QN models. Hence, for this subclass, the same considerations valid for QN networks also hold for the Markov case. If the assumptions underlying the BCMP theorem are violated, Markov models provide a higher accuracy level, but results depend on the characteristics of the software system under study and cannot be easily generalized. As a general consideration, the accuracy of Markov models depend on the precision of the state transition probability matrix, which includes, in real scenario, quite a large number of parameters.

Model Adoption. Similarly to QN models, Markovian models can be used as PINFM and PSNFM to derive performance and/or reliability metrics [32, 70]. Some recent extensions of Markov models address the problem of modelling system transients in order to study burstiness and long range dependency in system workloads. Authors in [67] introduce matrix-analytic analysis to study a Markovian Arrival Process as input and Phase Type distribution as service time in a single service center queue. Other studies (like [71]) focus on the analyses of non-renewal workloads by means on Markov models but, due to the analysis complexity, only small size models based on one or two service centers can be dealt with so far. As discussed above, in real systems Markov models suffer for high computation overhead and, hence, they are presently not suitable for run time modelling.

Tools for Model Derivation. In the literature there exists several contributions which propose transformation techniques for Markov models derivation. DTMC, for example, can be automatically derived starting from a description of the system behavior (PIM layer), using the methods and algorithms proposed in [32, 70]. At PSNFM level, Markov models and Markov decision processes are derived through ad-hoc transformations in [33, 34].

Some indirect transformations have been defined starting from software models and deriving GSPN [15, 16, 63], SPA [24] or PRISM models [29, 30]; the resulting models are then analyzed via to the numerical solution of the underlying Markov chain.

The aforementioned mentioned ad-hoc methods follow transformation patterns similar to the ones used for QN models.

6 Simulation Models

Simulation is a very general and versatile technique to study the evolution of a software system which is represented by means of a simulation model. Simulation can be adopted at design time in order to evaluate performance and reliability metrics both in steady state and in transient conditions.

Simulation requires the development of a simulation program that mimics the dynamic behavior of the system by representing the system components and interactions in terms of functional relations. Non functional attributes are estimated by applying output analysis methods to a set of observations gathered in the simulation runs.

Simulation results are obtained by performing statistical analyses of multiple runs. If the goal of the analysis is the steady state of the system, simulation output requires statistical assurance that the steady state has been reached. The main difficulty is to obtain independent simulation runs with exclusion of an initial transient period. The two techniques commonly used for steady state simulation are the "batch means method", and "independent replication" [39, 48]. None of these two methods is superior to the other in all cases. Their performance depends on the magnitude of the traffic intensity. The other available technique is the "regenerative method", which is mostly used for its theoretical nice properties; however, it is rarely applied in actual simulation to obtain the steady state output numerical results [39, 48].

Simulation output are characterized by confidence intervals which give an estimated range of values which is likely to include the performance/reliability metrics of interest for the system. The width of the confidence interval expresses uncertainty about the quality metric. A very wide interval, e.g, may indicate that more data should be collected during the simulation because nothing very definite can be said about the analysis. The confidence level is the probability value $(1 - \alpha)$ associated with a confidence interval. It is often expressed as a percentage. For example, say $\alpha = 0.05 = 5\%$, then the confidence level is equal to $(1 - 0.05) = 0.95$, i.e. a 95% confidence level.

Very often, simulation is used to evaluate performance metrics of non-product form QNs. In this case, the simulation program allows the definition of service centers and network topology and allows analyzing in detail the components behavior which violates BCMP theorem assumptions. For example, simulation allows analysing the impact of blocking conditions, or particular routing algorithms (e.g., the execution at the shortest queue among multiple parallel components). Furthermore, non-Poisson arrival rates for incoming workload, or heavy tail distributions (e.g., Pareto) for the service demands characteristics for Web systems can also be considered. Therefore the class of simulation models is much more general than the class of analytical product-form models. However, the major drawback of simulation with respect to QN models evaluation is their computational cost [39].

Accuracy. Simulation flexibility in general allows obtaining very accurate results. Indeed, the system behavior can be intrinsically captured by the simulation model and the accuracy of the results depends only on the desired confidence level. For example, some recent proposals can obtain results at an instruction cycle level precision, even for service center environments [51].

Model Adoption. Simulation models are adopted only at design time since the computational effort is significant both for the model derivation and for the computation time, especially if confidence internals are narrow. For example, simulation models have been adopted at the PIM layer in [5, 79] to evaluate the quality of service of composed BPEL processes, starting from the quality profile of the component Web services. At the PSNFM layer, simulation models are largely adopted also for the validation of the results of new bounding or approximate analytical solution of QN models [21].

Tools for Model Derivation. Ad-hoc methods have been proposed for the automatic transformation of software models into simulation models. Examples can be found in [1, 10]. Translations usually start from UML-like specification of the system and are mapped into internal representations, built on ad-hoc or general purpose simulation libraries.

7 Control-Oriented Models

The first step in the formulation of a control system design problem is the derivation of a mathematical model for the dynamics of the system to be controlled. Therefore so it is not surprising that the systems and control community can rely on a wide range of methods and tools for this task. More precisely, in this area it is common practice to distinguish between the so-called white and black box modelling paradigms. White box modelling refers to situations in which it is possible to derive the target mathematical model uniquely on the basis of first principles (e.g., conservation laws). Black box modelling, on the other hand, refers to situations in which the model for the system is derived entirely on the basis of data collected from the system itself during dedicated experiments. Various "shades" of grey box models can also be envisaged, depending on the needs of the specific application (see, e.g., [52]).

As discussed previously, genuine control-oriented modelling approaches can be used to accurately model software system transients at run time and to design control laws which can adjust a system configuration within a very short time frame. These methods are effective over fine grained control time horizons, e.g., minutes or seconds and, furthermore, they can formally guarantee both closed-loop stability and performance specifications.

The first application of system-theoretic modelling methods applied to the management of Web services are reported in [2, 3, 68, 73]. Early works focused on system identification techniques to build a Linear Time Invariant (LTI) model of a software system, e.g., the software system is described by the following set of equations:

$$\begin{aligned} x_{k+1} &= A_k x_k + B_k u_k + K_k e_k \\ y_k &= C_k x_k + D_k u_k + e_k, \end{aligned} \tag{1}$$

where k is the discrete time index, $x \in \mathbb{R}^n$ is the state vector, $u \in \mathbb{R}^m$ is the vector of control inputs, $y \in \mathbb{R}^l$ is the vector of measured outputs, and e_k is a white process noise. For example, the output could be the software system response time or utilization, the input could be the incoming workload and the state could represent the current number of request in the system. Note that, if black-box models are adopted, it might not be

possible to derive a "physical" interpretation of the state variables and the goal of the identification process is to determine a system specification which can accurately model a set of input-output observations, without any a priory system knowledge.

Identification procedures can be either performed off-line or on-line. In the former approach, ad-hoc performance tests injecting varying workload and working conditions on the software system are needed before the production deployment [74]. Vice versa, in the latter case, on-line methods can determine and adjust the values of the model parameters (matrices in equation (1)) while the software system is running [75]. In any case, dynamic models can be built and adopted only when the software is running in the target environment and hence they can be classified as run time models.

LTI models are usually employed to represent the local linear behaviour of a more complex non-linear system near a nominal operating condition. The resulting linear controller design may not suffice to allow the system to meet the target set point when the software environment is experiencing varying load conditions.

Linear Parameter Varying (LPV) models [50] have been recently proposed as a way of dealing with this kind of problem. LPV systems are linear time-varying models whose state space matrices $\{A(\delta_k), B(\delta_k), C(\delta_k), D(\delta_k)\}$ are fixed functions of some vector of varying parameters δ_k. LPV models have been adopted by Qin and Wang [65, 66] in order to identify a black-box model of a software system and implement an autonomic controller that can provide performance guarantees by dynamically changing the operating frequency of the physical servers and admission control, respectively.

Genuine control-theory techniques are very effective over fine grained time horizons. Recently, authors in [45] implemented a limited lookahead controller based on white-box models which is also effective on long-term time scales.

Accuracy. The preliminary results provided in the literature [74, 75] have shown that control-oriented models can be very accurate. In this case, the accuracy is traded-off with the time granularity (usually lower time granularity are more accurate but introduce a greater system overhead) which is also dependent on the workload intensity and variability (more variable and intense workloads require finer time grain). Authors in [74, 75] have shown LPV models introduce an average error in the evaluation of response time around 20%.

Model Adoption. Control-oriented models are adopted at run time for the implementation of closed-loop controllers with the aim to adapt the system configuration to environment changing conditions.

Tools for Model Derivation. Transformation tool in this area are still lacking and probably cannot be provided since the models are built from data measurements in preproduction/production environments.

8 Model Comparison and Discussion

The models presented in the previous sections are analyzed here according to a set of characteristics that are relevant for the system architect to drive model selection. To

complete the analysis carried out in Sections 4-7, here we provide a qualitative comparison, since the models are heterogeneous (e.g., some are oriented to performance evaluation while others to reliability) and can be used at different abstraction layers and/or at different time granularity. The characteristics are:

- *Adaptability*. Prediction techniques should support efficient performance prediction under architecture changes where: (i) components are modified, e.g. by introducing faster components, (ii) homogeneous components are added and the load is evenly shared among them, (iii) heterogeneous components are added and the load is not evenly shared (usually faster components receive higher load).
- *Cost effectiveness*. The approach should require less effort than prototyping and subsequent measurements.
- *Composability*. Prediction techniques should be able to make quality predictions based on the quality characteristics of single components, which together build the system. For example SOA systems are intrinsically structured hierarchically, hence quality prediction techniques should be able to exploit this structure in order to determine the QoS metrics of the whole systems by, possibly, exploiting the results obtained on lower abstraction layers or on the basis of the results of the analysis of single components.
- *Scalability*. Software systems are typically built either with a large set of simple components or utilize few large-grain, complex components. To predict performance attributes, analysis techniques need to be scalable to handle both cases.

The comparison is summarized in Table 2. We adopt a qualitative discrete scale for the evaluation of the above characteristics, i.e. *High*, *Medium*, and *Low* and the evaluation is justified by the following discussion.

Considering the adaptability characteristic, we have to analyse differently possible changes in the systems, i.e., modifying components, adding homogeneous components, and adding heterogeneous components, since these activities have different implications on the models. Here we consider the adaptability with the aim to determine new results by using always the same class of models and we do not consider the possibility to obtain new results through automatic transformations. In other words, the goal here is the capability to revise the model and obtain a new solution, always remaining in the same model family. Usually in the QN model family, bounding techniques

Table 2. Quantitative Models Qualitative Comparison

Model Family	Model	Adaptability	Cost Effectiveness	Composability	Scalability
Queueing Models	Bound Analysis	High	High	High	High
	Product form	Medium	High	High	Medium/High
	Non-product form	High	High	Medium	Low
	LQN	High	High	High	Medium
Markov Models	Discrete Time Markov Chains	High	High	Low	Low
	Continuous Time Markov Chains	High	High	Low	Low
	Markov Decision Processes	High	High	Low	Low
	Stochastic Model Checking	High	High	Low	Low
Simulation Approaches	Simulation	High	Medium	Medim/High	Low/Medium
Control-Oriented	LTI	Medium/Low	Low	Low	High
	LPV	Medium/Low	Low	Low	High

and non product form models have a high level of adaptivity. Vice versa, product-form models are adaptable if faster components or new homogeneous components are introduced in the system, while introducing heterogeneous components usually implies routing mechanisms which violate the assumption of the BCMP theorem. In that case, a system update requires to move from product to non-product form models or a different class of models (e.g., simulation). Considering the Markovian models, usually they provide a high level of adaptability, since a system update can be modelled simply as a faster service rate or a different probability distribution of the state space. Simulation is intrinsically adaptable since a system change can be implemented by modifying the program description of the added or updated components. Control-oriented models, vice versa, have a lower level of adaptability. In particular off-line black-box models require ad-hoc measurement which cannot be generalized, since the physical meaning of the system parameters is unknown.

Queuing and Markov models have a high level of cost effectiveness since the model solutions have a little effort if compared to the prototyping. Simulation has an intermediate level of cost effectiveness, since it requires a detailed description of system components behavior. The effort required is anyway lower than the one for the prototype development. Vice versa, control-oriented models have a low level of cost-effectiveness since they are based on the analysis of real data which have to be experimentally determined on a real implementation environment.

With respect to composability, the models which are structured hierarchically can be composed more easily. Vice versa, flat models or models based on very detailed descriptions (e.g. simulation models, if they are not structured in a modular way) or based on measurements (control-oriented models) have a lower level of composability.

Finally, with respect to scalability, bounding techniques can provide results for systems composed by several nodes or classes. Open product form models are scalable, while closed models have a lower scalability. The models adopted only at design time require considerable computation time to provide a solution and have a low level of scalability. Scalability of simulation models depends on the required level of accuracy. Indeed, the higher is the level of accuracy or narrower are the confidence intervals, the lower is the scalability. For example, some recent proposals can simulate up to 100 physical server in real service center environments, but in order to provide a solution in a small time requirement, need to be supported by 40 nodes [51]. Control-oriented models are also characterized by a high level of scalability. Usually each software component can be described by a local model. Furthermore, in autonomic system the whole infrastructure can be controlled by designing separately multiple local controllers [44, 45].

9 Conclusions and Current Research Challenges

In this paper, we discussed the role of models in software development. We focused on models that support software engineers reason about non-functional properties (performance and reliability). In addition, we focused on evolvable adaptive software systems, which must reconfigure their structure and behavior to respond to continuous changes in the requirements and in the environment. Systems of this kind are becoming increasingly relevant in emerging domains, such as pervasive computing applications and

modern distributed information systems for federated organizations. We stressed the fact that in these settings models are not only used at design time to guide systematic model-driven software development strategies, but they also need to live at run time.

At design time the goal of performance and reliability models is to provide quantitative predictions of run-time attributes and evaluate the impact of competing architectural choices. At run time, they continue to play an active, crucial role. First, they can be used as oracles for the implementation; that is, they can check if the behavior of the real system is correct with respect to the model. For example, the model may contain a specification of a property that at design time was proved to hold. By monitoring the real data at run time, it is possible to verify the property and detect a violation. The property might be violated because of flaws in the development process, but also because of (unexpected) dynamic changes that occurred in the environment. The violation, in turn, might trigger reconfiguration mechanisms that are performed autonomically by the running application in a self-managed manner, to achieve a self-healing behavior. These topics are currently actively investigated.

We are presently involved in research in this area. In particular, we are interested in casting the problem in the model-driven framework. We do so by collecting data at run time through monitoring, with the goal of supporting a dynamic update of the model. For example, the reliability of a certain channel whose initial estimate (contained in the current design model) may turn out to be inaccurate by examining real-world data. As another example, the probability associated with a certain transition of the model (e.g., a DTMC) may turn out to be wrong in the real world. In the examples, the data gathered by the run-time monitor may generate a feedback that provides a re-calibration of the model, and consequently the generation of a better new implementation. As a result, the model-driven process becomes a roundtrip from model to implementation and back, which progressively tunes the implementation in an attempt to provide a dynamic adaptive model-driven development strategy. Although this approach looks very appealing and some initial promising results have been obtained, further research is needed to develop a coherent approach that can solve the overall problem.

Acknowledgments

This work has been partially supported by the project Q-ImPrESS and S-Cube NoE funded under the European Union's Seventh Framework Programme (FP7). The authors are grateful to Mara Tanelli and Marco Lovera for fruitful discussions on control-oriented models. Thanks are also expressed to Elisabetta Di Nitto for insightful comments on the organization of the work.

References

1. Wosp : Proceedings of the international workshop on software and performance (1998-2007)
2. Abdelzaher, T., Lu, Y., Zhang, R., Henriksson, D.: Practical application of control theory to web services. In: Proceedings of the 2004 American Control Conference. Boston, USA (2004)

3. Abdelzaher, T., Shin, K.G., Bhatti, N.: Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. IEEE Transactions on Parallel and Distributed Systems 15(2) (March 2002)
4. Abrahao, B., Almeida, V., Almeida, J., Zhang, A., Beyer, D., Safai, F.: Self-Adaptive SLA-Driven Capacity Management for Internet Services. In: Proc. NOMS (2006)
5. Ardagna, D., Pernici, B.: Adaptive Service Composition in Flexible Processes. IEEE Transactions on Software Engineering 33(6), 369–384 (2007)
6. Ardagna, D., Trubian, M., Zhang, L.: SLA based resource allocation policies in autonomic environments. Journal of Parallel and Distributed Computing 67(3), 259–270 (2007)
7. Atkinson, C., Kuhne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (2003)
8. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
9. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time markov chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
10. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Software Eng. 30(5), 295–310 (2004)
11. Balsamo, S., Marzolla, M.: A Simulation-Based Approach to Software Performance Modeling. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 363–366. ACM Press, New York (2003)
12. Baskett, F., Chandy, K.M., Muntz, R.R., Palacios, F.G.: Open, closed, and mixed networks of queues with different classes of customers. J. ACM 22(2), 248–260 (1975)
13. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of component-based systems - a survey from an engineering perspective. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 169–192. Springer, Heidelberg (2006)
14. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP, pp. 54–65. ACM, New York (2007)
15. Bernardi, S., Donatelli, S., Merseguer, J.: From uml sequence diagrams and statecharts to analysable petri net models. In: WOSP 2002: Proceedings of the third international workshop on Software and performance, pp. 35–45. ACM Press, New York (2002)
16. Bernardi, S., Merseguer, J.: Performance evaluation of uml design with stochastic well-formed nets. Journal of Systems and Software 80(11), 1843–1865 (2007)
17. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.: Queuing Network and Markov Chains. John Wiley, Chichester (1998)
18. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1988)
19. Cardellini, V., Casalicchio, E., Grassi, V., Mirandola, R.: A framework for optimal service selection in broker-based architectures with multiple QoS classes. In: Services computing workshops, SCW 2006, pp. 105–112. IEEE computer society, Los Alamitos (2006)
20. Casale, G.: An efficient algorithm for the exact analysis of multiclass queueing networks with large population sizes. In: SIGMETRICS/Performance, pp. 169–180 (2006)
21. Casale, G., Muntz, R., Serazzi, G.: Geometric bounds: A noniterative analysis technique for closed queueing networks. IEEE Trans. Comput. 57(6), 780–794 (2008)
22. Chen, P.P.-S.: The entity-relationship model—toward a unified view of data. ACM Trans. Database Syst. 1(1), 9–36 (1976)
23. Chiola, G., Marsan, M.A., Balbo, G., Conte, G.: Generalized stochastic petri nets: A definition at the net level and its implications. IEEE Trans. Softw. Eng. 19(2), 89–107 (1993)

24. Clark, A., Gilmore, S., Hillston, J., Tribastone, M.: Stochastic process algebras. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 132–179. Springer, Heidelberg (2007)
25. Cortellessa, V., Marco, A.D., Inverardi, P.: Integrating performance and reliability analysis in a non-functional mda framework. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 57–71. Springer, Heidelberg (2007)
26. Cortellessa, V., Mirandola, R.: PRIMA-UML: a performance validation incremental methodology on early UML diagrams. Sci. Comput. Program. 44(1), 101–129 (2002)
27. Cunha, I., Almeida, J., Almeida, V., Santos, M.: Self-Adaptive Capacity Management for Multi-Tier Virtualized Environments. In: Proc. Integrated Management (IM) (2007)
28. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–646 (2006)
29. Gallotti, S., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Quality prediction of service compositions through probabilistic model checking. In: Becker, S., Plasil, F. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 119–134. Springer, Heidelberg (2008)
30. Gilmore, S., Kloul, L.: A unified tool for performance modelling and prediction. Reliability Engineering and System Safety 89, 17–32 (2005)
31. Gordon, W.J., Newell, G.F.: Closed queueing networks with exponential servers. Operat. Res 15, 252–267 (1967)
32. Grassi, V.: Architecture-based reliability prediction for service-oriented computing. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 279–299. Springer, Heidelberg (2005)
33. Grassi, V., Mirandola, R.: Uml modelling and performance analysis of mobile software architectures. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 209–224. Springer, Heidelberg (2001)
34. Grassi, V., Mirandola, R.: Derivation of markov models for effectiveness analysis of adaptable software architectures for mobile computing. IEEE Trans. Mob. Comput. 2(2), 114–131 (2003)
35. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance-reliability models of component-based systems: A model-driven approach. Journal of Systems and Software 80(4), 528–558 (2007)
36. Gu, G.P., Petriu, D.C.: From uml to lqn by xml algebra-based model transformations. In: WOSP 2005: Proceedings of the 5th international workshop on Software and performance, pp. 99–110. ACM Press, New York (2005)
37. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
38. Jackson, J.: Jobshop-like queueing systems. Management Science 10(1), 131–142 (1963)
39. Jain, R.: The Art of Computer Systems Performance Analysis–Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley-Interscience, Chichester (1991)
40. Kerola, T.: The composite bound method for computing throughput bounds in multiple class environments. Performance Evaluation 6(1), 1–9 (1986)
41. Kleinrock, L.: Queueing Systems. John Wiley and Sons, Chichester (1975)
42. Kramer, J.: Is abstraction the key to computing? Commun. ACM 50(4), 36–42 (2007)
43. Kramer, J., Magee, J.: Concurrency: State Models Java Programs, 2nd edn. Worldwide Series in Computer Science. John Wiley Sons, Chichester (2006)
44. Kusic, D., Kandasamy, N.: Risk-Aware Limited Lookahead Control for Dynamic Resource Provisioning in Enterprise Computing Systems. In: ICAC 2006 Proc. (2006)
45. Kusic, D., Kephart, J.O., Kandasamy, N., Jiang, G.: Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. In: ICAC 2008 Proc. (2008)

46. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 449–458. ACM Press, New York (2007)
47. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
48. Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis. McGrawHill, New York (2000)
49. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis Using Queueig Network Models. Prentice-Hall, Englewood Cliffs (1984)
50. Lee, L., Poolla, K.: Identification of linear parameter-varying systems using nonlinear programming. ASME Journal of Dynamic Systems, Measurement and Control 121(1), 71–78 (1999)
51. Lim, K., Ranganathan, P., Chang, J., Patel, C., Mudge, T., Reinhardt, S.: Understanding and designing new server architectures for emerging warehouse-computing environments. In: International Symposium on Computer Architecture, pp. 315–326 (2008)
52. Ljung, L.: Perspectives on System Identification. In: 2008 IFAC World Congress, Seoul, Korea. (to appear, 2008)
53. Marco, A.D., Inverardi, P.: Compositional generation of software architecture performance qn models. In: WICSA, pp. 37–46. IEEE Computer Society, Los Alamitos (2004)
54. Marco, A.D., Mirandola, R.: Model transformation in software performance engineering. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 95–110. Springer, Heidelberg (2006)
55. Marzolla, M., Mirandola, R.: Performance prediction of web service workflows. In: Overhage, S., Szyperski, C.A., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 127–144. Springer, Heidelberg (2008)
56. Menasce, D.A., Almeida, V.A.: Capacity Planning for Web Performance: Metrics, Models and Methods. Paperback (2001)
57. Menascé, D.A., Dubey, V.: Utility-based qos brokering in service oriented architectures. In: ICWS, pp. 422–430 (2007)
58. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
59. Object Management Group. OMG model driven architecture. (May 2006), http://www.omg.org/mda/
60. O.M.G. OMG.: UML Profile for Schedulability, Performance and Time 2005, http://www.omg.org/cgi-bin/doc?formal/2005-01-02
61. O.M.G. OMG.: UML Profile for Modeling and Analysis of Real-Time and Embedded Systems. ptc/07-08-04 (2007)
62. Papoulis, A., Pillai, S.U.: Probability, Random Variables, and Stochastic Processes, 4th edn. McGraw-Hill, New York (2002)
63. Pooley, R.: Software engineering and performance: a road-map. In: ICSE - Future of SE Track, pp. 189–199 (2000)
64. Puterman, M.L.: Markov Decision Processes. Wiley, Chichester (1994)
65. Qin, W., Wang, Q.: An LPV approximation for admission control of an internet web server: identification and control. Control Engineering Practice 15(12), 1457–1467 (2007)
66. Qin, W., Wang, Q.: Modeling and control design for performance management of web servers via an LPV approach. IEEE Transactions on Control Systems Technology 15(2), 259–275 (2007)

67. Riska, A., Squillante, M., Yu, S.Z., Liu, Z., Zhang, L.: Matrix-Analytic Analysis of a MAP/PH/1 Queue Fitted to Web Server Data. In: Latouche, G., Taylor, P. (eds.) Matrix-Analytic Methods: Theory and Applications, pp. 335–356. World Scientific, Singapore (2002)
68. Robertsson, A., Wittenmark, B., Kihl, M., Andersson, M.: Admission control for web server systems - design and experimental evaluation. In: 43rd IEEE Conference on Decision and Control (2004)
69. Rolia, J.A., Sevcik, K.C.: The method of layers. IEEE Transactions on Software Engineering 21(8), 689–700 (1995)
70. Sato, N., Trivedi, K.S.: Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 107–118. Springer, Heidelberg (2007)
71. Shwartz, A., Weiss, A.: Multiple time scales in markovian ATM models i. Formal calculations (1999)
72. Smith, C.U., Williams, L.G.: Performance and Scalability of Distributed Software Architectures: an SPE Approach. Addison Wesley, Reading (2002)
73. Abdelzaher, J.S.T., Lu, C., Zhang, R., Lu, Y.: Feedback Performance Control in Software Services. IEEE Control Systems Magazine 23(3), 21–32 (2003)
74. Tanelli, M., Ardagna, D., Lovera, M.: LPV model identification for power management of web service systems. In: 2008 IEEE Multi-conference on Systems and Control, San Antonio, USA (to appear, 2008)
75. Tanelli, M., Ardagna, D., Lovera, M.: On- and off-line model identification for power management of Web service systems. In: 47th IEEE Conference on Decision and Control, Mexico (to appear, 2008)
76. Tratt, L.: Model transformations and tool integration. Software and System Modeling 4(2), 112–122 (2005)
77. Urgaonkar, B., Pacifici, G., Shenoy, P.J., Spreitzer, M., Tantawi, A.N.: Analytic modeling of multitier Internet applications. ACM Transaction on Web 1(1) (January 2007)
78. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (puma). In: WOSP 2005: Proceedings of the 5th international workshop on Software and performance, pp. 1–12. ACM Press, New York (2005)
79. Zeng, L., Benatallah, B., Dumas, M., Kalagnamam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. on Software Engineering 30(5) (May 2004)

Design Reasoning Improves Software Design Quality

Antony Tang[1], Minh H. Tran[1], Jun Han[1], and Hans van Vliet[2]

[1] Swinburne University of Technology, Melbourne, Australia
{atang,mtran,jhan}@ict.swin.edu.au
[2] VU University, Amsterdam, The Netherlands
hans@cs.vu.nl

Abstract. Making justifiable decisions is a critical aspect of software architecture design. However, there has been limited empirical research on the effects of design reasoning on the quality of software design. The goal of this work is to investigate if there is any quality improvement to software design when design reasoning is applied. We conducted an empirical study involving twenty designers, the designers were asked to design a user interface and their designs were scored and compared. The results showed that the test group that was equipped with design reasoning produced a higher quality design than the control group, especially for inexperienced designers.

Keywords: Design Reasoning, Software Architecture Design, Usability.

1 Introduction

Software designers tend to base their judgments on prior beliefs and intuition rather than a logical reasoning process. This tendency is common to human thinking and has influenced the performance of many people's reasoning and decision making [1]. In this paper, we demonstrate that software design is subject to the same cognitive bias, and therefore can affect the quality of its results. Through an empirical study, we show that the design quality can be improved with a simple design reasoning approach.

Recent research, especially in the area of software architecture, has shown that design reasoning and design rationale are important [2]. Design reasoning supports designers in making justifiable decisions by explicitly modeling design rationale as a first-class entity [3]. Functional and quality requirements are considered in such methods, and decision making techniques such as trade-off analysis are used to select a solution that best suits the design criteria. It is argued that design rationale should be captured explicitly, together with the decisions taken and the resulting design. They constitute the software architectural knowledge [4].

This research explores how design reasoning affects the quality of design outcome. In exploring this issue, we have carried out an empirical study with the usability quality attribute. Usability is an important quality attribute in software architecture [5, 6]. It is concerned with whether users are able to use a system to perform their tasks effectively, efficiently and with satisfied experience.

We hypothesize that by applying a design reasoning process, designers would come up with a more usable UI. The study was conducted with two groups of designers who

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 28–42, 2008.
© Springer-Verlag Berlin Heidelberg 2008

have similar design experience. The control group carried out the design as they usually do, whilst the test group carried out the tasks using a design reasoning approach. The results have demonstrated that the use of a reasoning approach has, on average, improved design quality. Especially for relatively inexperienced designers, the improvement is noteworthy. For them, design reasoning provides a framework for deliberation and supports building up and maintaining a mental image of the ongoing design. Experienced designers have less need for such assistance, they simply "know" how to proceed [7].

The remaining of the paper is organized as follows. Section 2 discusses the concepts of design reasoning and the related work in usability design reasoning. Section 3 presents the empirical study and the findings. Section 4 discusses the lessons that we have learned from applying design reasoning to software design. We conclude the paper in section 5.

2 Related Work

2.1 Design Reasoning

Researchers in psychology have proposed that there are two distinct cognitive systems underlying reasoning. System 1 (heuristic system) comprises a set of autonomous subsystems that include both innate input modules and domain-specific knowledge acquired by a domain-general learning mechanism. System 2 (analytic system) allows reasoning according to logical standards [1]. Together they form the dual process theory that explains people's "rational thinking failure" when people rely heavily on prior beliefs and intuition rather than a logical reasoning process [8, 9]. These findings are also confirmed in a more recent study on decision making in software design where designers make use of rational and naturalistic decision making tactics [10].

In software development, design reasoning is an important process that designers use in developing a solution. Designers in the software industry often rely on intuition and experience to make design decisions. The drawback of such an approach is that the quality of decisions would heavily depend on the experience and expertise of the individuals. Since design rationale plays an important role in making design decisions [2, 11], understanding what constitutes design rationale is important to producing a good design. Rittel and Webber [12] view design as a process of negotiation and deliberation. They suggest that design is a "wicked problem" in which there is no well-defined set of potential solutions, so it is important that a designer learns how to handle and weigh alternatives.

There are different approaches to design reasoning. One approach is by way of argumentation. The basic argumentation-based representation is to use nodes and links to represent knowledge and relationships. Examples of this approach are QOC [13], DRL [14] and gIBIS [15]. A second approach is by way of using rationale template to capture design reasoning. This approach incorporates standard templates into the design process to facilitate design rationale capture. This approach is oriented towards the practical implementation of design rationale in industry. Examples are Architecture Decision Description Template [16] and Views and Beyond [17]. A third approach is a hybrid of the first two approaches. Examples of this approach are AREL [18] and Archium [4].

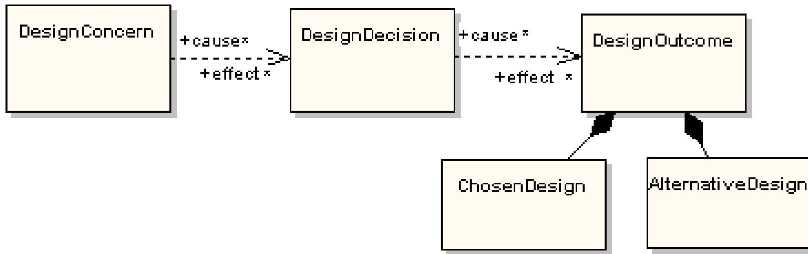


Fig. 1. AREL – A Design Reasoning Model

Whilst there are different approaches to design reasoning, the concepts presented in AREL are common to current research thinking [4, 19]. In the rest of this section, we introduce the AREL model, which concepts we used in the empirical study. In this model, there are three key elements to be considered: design concerns, design decisions and design outcomes (Fig. 1).

Design concerns are concerns that motivate the creation of a design solution. Design concerns are the causes for design decisions to be made. A design concern may be a system requirement, a business goal, a quality attribute such as usability, a circumstance that influences a design, or an existing design component that exerts some design constraints.

A design decision is made by a designer when assessing why a particular design is created or chosen. Capturing this knowledge is important because it justifies the design and explains the reasons to those who do not have intimate knowledge of the design - users, testers and maintainers. The key information contained in the design decisions are the design issues, design assumptions, design constraints, and design rationale for the selection or rejection of a design option. The links (Fig. 1) between design concerns and design decisions indicate what design concerns are considered in a decision. The results of the design decisions, i.e. design outcomes, are linked to the design decisions because the outcomes are the results of a decision.

Design outcomes are the results of a design decision: chosen design is the design that has been selected, and it contains the design elements such as components, classes and database schemas that would be implemented; alternative design is the design options that have been considered but rejected. Capturing all the design options, including the rejected alternative design, are important for three reasons: (a) a comprehensive consideration of all available options shows that the designer has not omitted any viable design option; (b) documenting design options can support design backtracking if an initial design solution is unviable when more details are considered; (c) identifying different design options allow design trade-off analysis to be performed to justify the selection of the most appropriate design option.

2.2 User Interface Usability

The quality of UI designs has been considered as an important aspect of software architecture. Research has shown that in UI design, designers are required to make decisions on selecting design options in one way or another [20-24]. However, existing UI design lifecycle models provide very limited guidelines that help

designers reason about design options and make justifiable design decisions. UI design reasoning is something that has always been assumed and intuitive.

Recent studies have pointed out that the outcome of UI design includes both the resulting interface itself and a rationale for why the interface is chosen. Howard's exploratory study [22] identifies key elements, including environment, focus and agents, which are often taken into account when designers make tradeoff decisions. Howard's study also models a behavioral process of which designers make an argument in choosing between two key elements. MacLean et al. [24] examine a representation of design rationale. They have developed a semi-formal notation that allows designers to represent explicitly design options and reasons for choosing from amongst them. However, to the best of our knowledge, there is no empirical evidence concerning how design reasoning influences UI design quality.

3 An Empirical Study

The objective of this study was to explore the effects of design reasoning on the quality of design. To do so, we asked the participants to design a user interface for a commercial system. The case was carefully chosen and simplified to make sure that it makes sense to the participants and it was not so simple that it could be designed without careful considerations.

We have been working with an automotive company to develop a Web-based system to monitor test vehicles of a fleet. The key function of the system is to allow car engineers to monitor electronic signals collected from the electronic control units (ECUs) in a vehicle. Through a UI, an engineer is able to prepare a monitoring schedule which specifies the information to be monitored. Each monitoring schedule must contain a minimum of 1 request and a maximum of 99 requests (i.e. requirement R_1). Engineers would search and select the electronic signals from a search list of 1700 signals when specifying requests (i.e. requirement R_2). A request specifies what electronic signals need to be monitored and monitoring start and stop conditions (i.e., requirement R_3). Other requirements such as the insertion and deletion of monitoring conditions and signals were given to the participants but we do not describe them in detail here.

3.1 Participants

The study involves twenty participants. We used a convenient sampling method to invite practitioners in the software industry and academia to participate in the study. The participants were allocated randomly to two groups: *test group* and *control group*. The average design experience of the test group and the control group are 8.95 years and 8.40 years, respectively.

3.2 Experiment Procedure

Both the control group and the test group were asked to design a UI for the monitoring system. The groups were given the following: (a) a set of requirements for the design; (b) usability requirements; and (c) UI controls that can be used in the design. We carried out the experiment with each participant individually.

The control group carried the design as they usually do. The test group was briefed about the design reasoning process and they were asked to apply the principles of

design reasoning. In the briefings, we described the design reasoning principles of AREL without referring to its formal model. During the experiment when the participants reach a design decision point, they need to explain their design options and issues, and justify why they chose a particular design option over other alternatives. As the participants justified their design decisions, the interviewers did not give any hints on how to design nor engaged in discussing the quality of the participants' design. However, the interviewers would ask the following questions to ensure that reasoning was applied: "What are the issues in the decision?" and "What are the options to deal with the issues?"

Participants in both the test and control group were asked to use a think-aloud protocol [25] to describe their design strategies. At the end of the design session, the participants were interviewed for their comments. In the interview, we used the Retrospective Think Aloud (RTA) technique [26] to gather participants' comments on their own designs after they had completed the tasks. We timed each design session, starting from when the participants commenced the design process until they completed the design, excluding the briefing and the interview.

Both quantitative and qualitative data were collected in the study. The quantitative data included details of the participants' experience, duration taken to complete their tasks, the levels of their satisfaction and confidence in their own designs and the quality scorings of their designs. The qualitative data was collected from the participants' think-aloud process, our assessment of the participants' design, our observation of the participants' design process, and the participants' comments.

3.3 Findings

We analyzed the test results from three perspectives: the quality of the design outcomes, the design process and the participants' feedbacks.

3.3.1 Design Outcomes

With each UI design, we assess the quality of the design based on three (out of ten) UI design heuristics proposed by Nielsen [27]: (a) consistency; (b) flexibility; (c) accessibility. For instance, we assess design consistency by inspecting if participants used UI controls (e.g. scroll-bars, buttons) consistently across the design. For the purpose of this study, we have selected only three most relevant usability heuristics.

For each participant's design, we rated the usability based on the selected heuristics. For each heuristic, we used a 5-point Likert scale ranging from 0 to 4, with 4 being the best design. Thus, the top score of a design is 12 and the worst score is 0. The rating process was done by comparing how well each design conforms to the heuristics. When scoring the designs, we scored and compared the designs irrespective of which group they come from to ensure that the scorings were consistent and unbiased.

We hypothesize that the test group who equipped with design reasoning would produce better quality design than the control group. If *m1* is the average score of the test group and *m2* is average score of the control group, the hypotheses are:

$$\mathbf{H_0}: m1 = m2$$

$$\mathbf{H_1}: m1 > m2$$

Table 1. Test and Control Group Design Quality Scores

	n	Mean Score	Std. dev.	Wilcoxon Test
Test Group	10	9.10	1.52	$p = 0.02$
Control Group	10	7.10	1.91	

The null hypothesis $\mathbf{H_0}$ states that the quality of the UI design created by the control group is the same as the test group. The average scores of the control group is 7.10, and that of the test group is 9.10. The one-tailed Wilcoxon[1] [28] test shows that there is a significant difference in design quality between the test and control groups with $p < 0.025$[2] (see Table 1). Since the one-tailed Wilcoxon test is significant, we reject $\mathbf{H_0}$ and accept the alternative hypothesis $\mathbf{H_1}$. The conclusion is that the application of a design reasoning process has improved the quality of a UI design.

In order to understand what quality aspects of the design are different between the two groups, we analyze each participant's design. There are two key design issues involved and both of them are concerned with usability:

- (R_1) catering for between 1 and 99 monitoring requests; and
- (R_2) determining how to search for 1700 vehicle signal(s) and selecting them for monitoring and monitor triggering.

Test Group. All participants in the test group used the rationale-based approach to design for the given requirements. In designing for requirement R_1, the participants selected either a scrollable tab or a side-located expandable list to display and select a request. The participants had initially considered different design options such as a pop-up list, a list-table in the center of the page and a dropdown list. These options had been discarded after the participants considered the usability issues as part of the design reasoning process.

In designing for requirement R_2, all participants except one in the test group used a pop-up window to specify the monitoring signals. Although other design options such as a dropdown list and a button-control had been considered by the participants, they decided that only one compromised solution is viable. This is because of the combination of constraints that are present: limited screen real-estate, the need to copy specified signals to multiple controls, and reusable programs. Some participants explored each branch of an option in detail and backtracked when the options became unviable. Overall, all participants have created a similar design. They have articulated similar issues related to usability. Most of them have identified a similar set of design options with minor variations. These minor variations are the placements of controls such as "buttons" and "tables".

There was one exception in the group. This design used a menu driven approach. Although the UI was still usable, it did not conform to the Nielsen UI design heuristics and therefore the scores of this design were lower.

[1] Wilcoxon's test is a non-parametric test suitable for comparing ranked data that makes no assumption about their distribution .

[2] We used the standard test of significance at 0.05. For one-tailed test of $\mathbf{H_1}$ in our case, the significance is at 0.025.

Control Group. In comparison to the test group, the control group produced much more diverse and less usable designs. First, as for designing UI to handle multiple requests (i.e., requirement R_1), four varieties of design were proposed, including:

- *A textual list*: Requests are organized in the form of a hyperlink list on one side of the screen, whilst the rest of the screen estate displays details of a request.
- *Graphical icons*: Requests are represented as graphical icons numbered from 1 to 99. In this design, it takes the entire screen estate to show 99 icons.
- *A textbox*: There is no list to show an overview of all requests. Users retrieve details of a request by entering a unique identification of the request into the provided textbox. This design saves screen estate, but requires users to remember identifications of all requests.
- *A dropdown list*: A dropdown list shows all requests. This design saves screen estate, but scrolling down a long list of requests could be inconvenient.

Amongst these design variations, only the textual list is usable, and four out of ten participants in the control group ended up with this design. In the other six designs where the usability was low, the participants designed for multiple requests after they had finished designing for an individual request. In these cases, they did not reconsider if their individual request design fits multiple requests.

Second, for searching and selecting signals to specify a request (i.e., requirement R_2), the control group derived three discrete designs, including:

- *Pop-up window*: Buttons are included in different sections of a request (e.g., start condition, monitoring list and stop condition). When users click on a button, a pop-up window is displayed allowing users to select one or more signals to add to a particular section of the request.
- *Sequential pages*: Users must first search and select signal(s) that they want to monitor, and move to different pages to paste the information.
- *Tabbed windows*: Tabs are used to provide different functionalities. For example, the first tab shows a grid of all available signals from which users can choose, the second tab shows a list of users' selected signals, and the third tab shows start and stop monitoring conditions. Users click between them to copy signal information.

Out of these three designs, the pop-up window mechanism is the most suitable for the system, because it allows users to select signals with minimal mouse clicks and errors. Only five out of ten participants of the control group used a pop-up window as a means of searching and selecting signals.

3.3.2 Designers' Experience and Design Quality

Using the quality scores of the participants in the test group and the control group, we analyzed the relationships between their quality scores and their design experience. The results are shown in Fig. 2. The solid diamonds depict those who used design reasoning, and the hollow squares depict those who did not use design reasoning.

We notice that above the 5 year experience mark, the quality scores of the two groups are similar. They mostly score between 8 and 10 with some outliers. However, there is a noticeable difference in quality between the two groups of participants

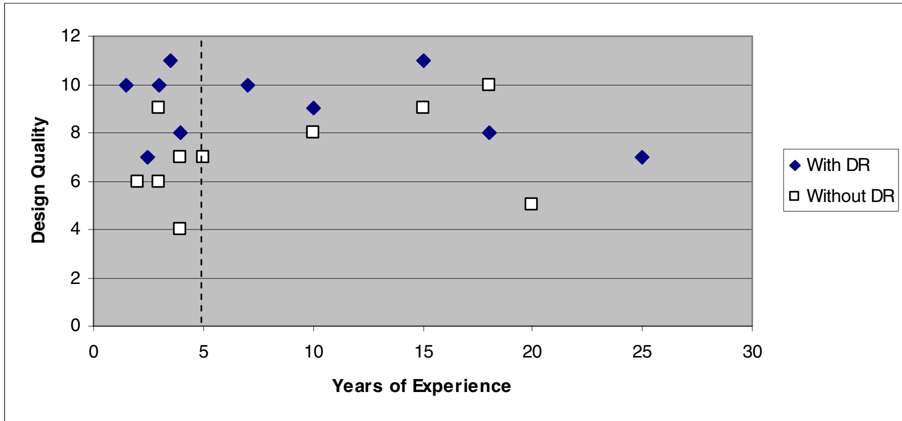


Fig. 2. Design Quality Scores and Years of Experience

having less than 5 years of experience. The majority of the participants in the control group in this category scores between 6 and 8, and the majority of the test group in this category scores between 8 and 10. These results indicate that design reasoning helps less experienced designers to design better.

It is interesting to observe that the two most experienced designers did not produce the best designs. Both entered into the information technology field in the era of batch processing systems, way before graphical user interfaces (GUI) became an issue. Their design knowledge was formed in that pre-GUI era, and seems not changed all that much thereafter. This was confirmed by their actions and thinking-aloud during the experiments.

3.3.3 Design Process

We compare the duration that the two groups took to finish their tasks. On average, the test group took 39.30 minutes and the control group took 29.40 minutes. The Wilcoxon test shows no significant difference between the time spent by the test and the control group with $p > 0.05$ (see Table 2). That means both groups took a similar amount of time to finish their tasks.

Table 2. Test and Control Group Design Time

	n	Mean Time (min)	Std. dev.	Wilcoxon Test
Test Group	10	39.30	10.86	$p = 0.113$
Control Group	10	29.40	10.08	

In addition, we considered the design processes of the two groups as described below.

Test Group. During the study, the test group was required to state their design options and design issues explicitly, and to justify why they made a particular decision at every design point. For example, when they explained the issues of the

initial design, they would say something like: "*how do I organize the UI to show 99 requests when they cannot all be displayed at the same time*" or "*how do I copy data from the signal search screen in a way that is easy to use*" or *"how many clicks are required to get the job done*". They contemplated their initial design and reasoned about what it could and could not do. In some cases, participants believed that the initial design was adequate. However, when the participants consciously tried to find more design options, they often came up with alternative designs. There were many cases in the experiments in which such alternatives became the final design. However, there were cases when the design alternatives had helped to reinforce that the initial design was more appropriate when all alternative designs seem to be inferior.

After verbalizing the issues, the participants of the test group seemed to have a mental picture of those issues. The participants often considered how these issues conflict or work with each other in the design. The explicit verbalization of design issues and options helps the thought process when the participants started to formulate design options and to backtrack when certain design issues cannot be resolved. For instance, one participant explored the design issue of listing the 99 requests first and then he examined the issue of displaying and editing a single request, and finally the searching of signals. At every decision point, he would backtrack to assess if the chosen design options would work. This backtracking and verification of design options seemed quite natural to the participants of the test group when the issues and options were explicitly stated.

Control Group. The participants of the control group were not asked to state design issues and justify their decisions. We observed different design behavioral patterns depending on the experience of the designers. The first pattern was that most participants' initial design became their final design, especially for designers who are less experienced. Unlike the test group, the control group's design approach was based heavily on their intuition and on their first impression. The participants appeared to adhere to their initial designs, from where they continued to design for additional requirements. After each decision point, the inexperienced participants especially, spent little efforts on reassessing the consequences of additional changes to the initial design.

The second pattern was that even though the control group was aware of the usability guidelines, most of them did not consider the usability requirements at every decision point. We realized that the participants talked about the usability requirements initially but they became less conscious of them as the design progressed. The more experienced designers in this group were the exceptions.

3.3.4 Participants' Feedbacks

In the follow-up interviews after the design session, all the participants were asked to comment on two things:

(a) "*What are your key considerations for the design*?"
(b) "*Do you have any comments on the design process you went through in this exercise*?"

In response to question (a), all participants of the test group mentioned that the usability of the UI was a key issue because of the complexity of the requirements and

the limited screen real-estate. Most participants said that the ease of use and understanding was another key issue and argued that minimal user clicks and minimal UI screens should be provided. Some of the participants also considered that the design should be reusable, particularly the signal search function.

Similar to the test group, the participants of the control group also commented that usability requirements were key to their design. In fact, a list of important usability concerns drawn from the control group was very similar to that of the test group. A convergence in the participants' comments on question (a) indicated that even though both groups of participants were aware of the usability requirements, they had very different approaches to tackling them. The participants in the test group were asked to explicitly state their design issues and design options, and their final designs were more consistent and more usable. Whereas with the control group, the participants carried out the design the way they normally do, the results among the participants were less consistent and showed an inadequate level of usability.

In response to question (b), some participants of the test group commented that well stated design issues helped them think through the design. Nine out of the ten participants in this group mentioned that the exploration of design options had helped their design. When they were asked why this was so, the general suggestion was that the design options allowed them to assess what would and would not work. In the control group, the inexperienced participants had very little comments on their design process, whilst experienced designers in this group were able to describe their requirement analysis and design process.

After the participants had completed their tasks, they were asked to rate their *satisfaction* with their own designs, using a seven-point Likert scale where 1 is not satisfied and 7 is fully satisfied. The test group reported an average of 5.7 and the control group reported an average of 4.9. When the participants were asked how *confident* they were on the usability of their design, using a seven-point Likert scale where 1 is not confident and 7 is fully confident, the test group reported an average of 5.1 and the control group reported an average of 5.5.

Although the test group was more satisfied with their design, they were slightly less confident about it than the control group. We cannot offer any explanations for this outcome, except by showing that the differences between the two group's ratings are statistically insignificant. The Wilcoxon test results show that there is no significant difference between the two groups' ratings on either the satisfaction or confidence of their design, $p = 0.142$, and $p = 0.34$ respectively. This finding shows that participants from both groups were similarly confident and satisfied with their own designs despite the differences in the design qualities (as reported in Section 3.3.1).

4 Discussions of the Findings

The primary objective of this study is to analyze how design reasoning influences the quality of design. From the findings of the experiment, we have made a number of observations on how the participants tackled the design.

4.1 Discussions

Participants of both groups studied the requirements and the usability guidelines before creating their design. Analyses have shown that the test group has produced better quality UI designs than the control group in general. The following is a summary of the differences between the test group and the control group:

Reasoning awareness. By stating the design rationale, the participants of the test group have made explicit the reasoning underlying their designs. This imposed justification process had made the participants more cognizant of whether their decisions were correct. For instance, after spelling out that usability issues that concerned them, they had to find ways to ensure that their design was reasonable in dealing with the usability issue. As such, the test group was more aware of the usability requirement in the design compared with the control group. This result could reflect on the importance of reasoning of quality requirements in software architecture design.

Additionally, the reasoning approach induced the participants of the test group to explicitly reason about their design in a structured manner. Therefore, they were probably more careful in assessing their solutions in order to provide reasonable justifications. This contrasts with the participants of the control group who mostly used their intuition and knowledge to design. The control group's objective was to complete the design and satisfy the requirements without having to justify them.

Usability awareness. The participants of the test group identified usability as a key issue to be addressed in the design, and they consistently revisited this issue in the reasoning process. On the other hand, the participants of the control group considered usability in the early part of the study and then they were less conscious about it towards the end of a design session. It implies that an explicit design reasoning can help make designers aware of quality requirements throughout the design process.

Initial design impression. We observed that participants from both the test and control group formed initial impressions of a design solution initially. After exploring the design issues and options, participants in the test group may shift from the initial impressions of the design based on their design reasoning. On the other hand, the initial design impression played a more dominant role in the control group, especially with inexperienced designers, the initial design often became their final design. This result indicates that the initial design impression can be dominant, but a reasoning approach would help designers consider the design issues and options more carefully, allowing the designers to move away from the dominant belief to a design that is more appropriate.

Design backtracking. The participants of the test group backtracked their design and reconsidered their previously made decisions much more often than those of the control group. We suggest that it is because the test group was asked to explicitly state their issues, options and design rationale, and such acts forced them to address the issues that have been outlined, thereby achieving a level of systematic design reasoning. The control group generally did not reconsider previously made decisions as they built their design. Individual requirements were addressed but issues that arose from the conflicting requirements were not identified.

Level of satisfaction and design quality. The level of satisfaction and the level of confidence between the test group and the control group were not significantly different. This is despite that the test group had carried out the design process more thoroughly and produced higher quality designs. We also did not observe differences between the more experienced and less experienced designers. The results have shown that the level of satisfaction and the level of confidence of a designer on his/her design are not good indicators of the design quality.

Design time. There is no significant difference between the average time it takes for the test group and the control group to complete their tasks. It implies that the effort (in terms of time) spent by the test group is not significantly higher than that of the control group. Thus, we have not found evidence to indicate that using reasoning in design adds significant overhead to the design process.

Experience level and design quality. We have examined the design experience of the two groups and have found them to be similar. However, as shown in Table 1, the test group performed better on average than the control group. This is due to the higher scores achieved by less experienced designers in the test group (see Fig. 2).

In the test group, the design outcomes of experienced (i.e. over 5 years) and inexperienced (i.e. equal or less than 5 years) participants do not show much difference. They both produced good quality design. The inexperienced participants on average took longer to complete their design. In contrast to the test group, there is an observable difference in design quality between the experienced and inexperienced participants in the control group. Put differently, in the group of inexperienced participants, those that used design rationale consistently performed better than those that did not (see Fig. 2).

These results suggest that by using a design reasoning approach, less experienced designers could benefit from it to achieve a better quality designs. All designers, inexperienced and experienced, were briefed equally of the first principles of usability. Design reasoning has helped inexperienced designers in the test group to apply these principles successfully, and achieve what expert designers can do from mere experience, but design reasoning has shown little difference between experienced designers in the two groups. This suggests that experienced designers have the intuitions and insights to look for the right issues and options, as demonstrated by [7].

4.2 Limitations

This experimental study was based on a sample of twenty designers with industrial experience. We used a convenient sampling method to find the participants, i.e. the participants are the people whom we had access to and they were not randomly selected. The sample size in this study is small, and so there are limitations on the interpretations of the results.

The participants of the test group were explicitly required to state their design issues, design options and reasoning. Such reminders may have directed them to think more thoroughly, as such one could argue that the presence of the interviewers may bias the results. However, the interviewers did not provide any design hints, and the design decisions were deliberated entirely by the participants based on their knowledge

and reasoning abilities. Hence, we argue that the test results from the experiments are valid.

There are different experimental variables in such empirical studies that cannot be strictly controlled, e.g. participants' familiarity with the technologies involved. To overcome this limitation, we analyzed qualitatively what the designers have done and said about their designs to ensure that these variables do not affect the validity of the results. As for the quantification of the scores, the limitation is the bias the researchers may introduce. To overcome this, we have cross-checked all the designs to ensure that there is a consistency scoring across all designs.

5 Conclusions

Recent research works have argued that the explicit representation of design rationale is useful and can lead to better design outcomes. Yet there has been limited research to examine design reasoning's impact on design quality. Using usability as a software architecture quality attribute, we have studied how design reasoning influences the design quality, especially differentiating between experienced and inexperienced designers.

We have used an empirical study to examine the design quality of two groups of designers, one equipped with design reasoning and one without. The results of the experiment have shown statistically that a design reasoning approach improves the quality of design.

Designers who explicitly reason about their design decisions produce on average better designs. Furthermore, design reasoning appears to help inexperienced designers more than they do help experienced designers. The designers who do not use explicit design reasoning produce diverse results, and some of the designs have low usability, especially in the case of inexperienced designers. Therefore, we conclude that design reasoning helps inexperienced designers to better apply first design principles and to deliver a better design, by providing them with a deliberation framework and a mental image of the ongoing design.

These findings have provided encouraging empirical results to support further investigation into incorporating design reasoning in the software architecture design process.

Acknowledgments. This work is supported in part by the Australian Collaborative Research Centre for Advanced Automotive Technology and the Swinburne University of Technology Visiting Professor Award Scheme 2008.

References

1. De Neys, W.: Implicit conflict detection during decision making. In: Proceedings of the Annual Conference of the Cognitive Science Society, vol. 29, pp. 209–214 (2007)
2. Tang, A., Barbar, M.A., Gorton, I., Han, J.: A survey of architecture design rationale. Journal of Systems and Software 79(12), 1792–1804 (2006)

3. Bosch, J.: Software Architecture: The Next Step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004)
4. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Proceedings 5th IEEE/IFIP Working Conference on Software Architecture, pp. 109–120 (2005)
5. Bass, L., John, B.E.: Linking usability to software architecture patterns through general scenarios. The Journal of Systems and Software 66(3), 187–197 (2003)
6. Golden, E., John, B.E., Bass, L.: The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 460–469 (2005)
7. Cross, N.: Creative Thinking by Expert Designers. The Journal of Design Research 4(3) (2004)
8. Epstein, S.: Integration of the cognitive and the psychodynamic unconscious. American Psychologists 49, 709–724 (1994)
9. Evans, J.S.: In two minds: dual-process accounts of reasoning. Trends in Cognitive Sciences 7(10), 454–459 (2003)
10. Zannier, C., Chiasson, M., Maurer, F.: A model of design decision making based on empirical results of interviews with software designers. Information and Software Technology 49(6), 637–653 (2007)
11. Bratthall, L., Johansson, E., Regnell, B.: Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution. In: Second International Conference on Product Focused Software Process Improvement, pp. 126–139 (2000)
12. Rittel, H.W.J., Webber, M.M.: Dilemmas in a general theory of planning. Policy Sciences 4(2), 155–169 (1973)
13. Maclean, A., Young, R., Bellotti, V., Moran, T.: Questions, Options and Criteria: Elements of Design Space Analysis. In: Moran, T., Carroll, J. (eds.) Design Rationale - Concepts, Techniques, and Use, pp. 53–105. Lawrence Erlbaum, Mahwah (1996)
14. Lee, J., Lai, K.: What is Design Rationale? In: Moran, T., Carroll, J. (eds.) Design Rationale - Concepts, Techniques, and Use, pp. 21–51. Lawrence Erlbaum, Mahwah (1996)
15. Conklin, J., Begeman, M.: gIBIS: a hypertext tool for exploratory policy discussion. In: Proceedings of the 1988 ACM conference on Computer-supported cooperative work, pp. 140–152 (1988)
16. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE SOFTWARE 22(2), 19–27 (2005)
17. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2002)
18. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
19. Ali-Babar, M., Gorton, I., Jeffery, D.R.: Capturing and Using Software Architecture Knowledge for Architecture-Based Software Development. In: Proceedings of the Quality Software International Conference (QSIC 2005), pp. 169–176 (2005)
20. Carroll, J.M., Rosson, M.B.: A case library for teaching usability engineering: Design rationale, development, and classroom experience. Journal on Educational Resources in Computing 5(1), 1–22 (2005)

21. Mayhew, D.J.: The usability engineering lifecycle: a practioner's handbook for user interface design. Morgan Kaufmann Publishers, San Francisco (1999)
22. Howard, S.: Trade-off decision making in user interface design. Behaviour & Information Technology 16(2), 98–109 (1997)
23. Norman, D.A.: Design principles for human-computer interfaces. In: Proceedings of the SIGCHI conference on Human Factors in Computing Systems, pp. 1–10. ACM Press, New York (1983)
24. MacLean, A., Young, R.M., Moran, T.P.: Design rationale: the argument behind the artifact. In: Proceedings of the SIGCHI conference on Human factors in Computing Systems, pp. 247–252. ACM Press, New York (1989)
25. Erikson, T.D., Simon, H.A.: Protocol Analysis: Verbal Report as Data. The MIT Press, Cambridge (1985)
26. Guan, Z., Lee, S., Cuddihy, E., Ramey, J.: The validity of the stimulated retrospective think-aloud method as measured by eye tracking. In: Proceedings of the SIGCHI conference on Human Factors in Computing Systems, pp. 1253–1262 (2006)
27. Nielsen, J.: Ten Usability Heuristics. (2007), `http://www.useit.com/papers/heuristic/heuristic_list.html`
28. Walpole, R.E., Myers, R.H.: Probability and Statistics for Engineers and Scientists. Macmillan Publishing Co., Inc, Basingstoke (1978)

A Tool to Visualize Architectural Design Decisions

Larix Lee and Philippe Kruchten

University of British Columbia
{llee,pbk}@ece.ubc.ca

Abstract. The software architecture community is shifting its attention to architectural design decisions as a key element of architectural knowledge. Although there has been much work dealing with the representation of design decisions as formal structures within architecture, there still remains a need to investigate the exploratory nature of the design decisions themselves. We present in this paper a tool that should help improve the quality of software architecture by enabling design decision exploration and analysis through decision visualization. Unlike many other design decision tools which acquire, list, and perform queries on decisions, our tool provides visualization components to help with decision exploration and analysis. Our tool has four main aspects: 1) the decision and relationship lists; 2) decision structure visualization view; 3) decision chronology view; and 4) decision impact view. Together, these four aspects provide an effective and powerful means for decision exploration and analysis.

1 Introduction

Software design is derived from making many decisions; capturing the most significant of these decisions would help convey significant insight and rationale behind the different aspects or features of the system architecture and design. However, the architectural knowledge provided by a simple enumeration of design decisions is often dry and difficult to peruse. If decisions can be browsed or visualized in an effective manner, the amount of time spent on communicating the software design with others can be reduced.

Defining software architecture to be a set of important design decisions [16] suggests that we need to effectively capture, browse, and exploit such design decisions. We addressed improving decision capture in an earlier paper [22], but for decision perusal and analysis, we propose that visualizing design decisions using different perspectives may improve the quality of decision information conveyed to the software architect, designer, or developer, which results in making better system architecture.

We have built a tool that assists the architects in decision exploration and analysis by employing different aspects to visualize a set of design decisions. This paper describes our tool and its various components as well as discusses how the tool contributes to architectural design decision exploration.

The paper is structured as follows: Section 2 provides some background into the scope of our research. Section 3 describes the previous work in visualizing and exploring software architectural design decisions, while section 4 describes our tool in detail. Sections 5 and 6 describe our experiences and our future work with the tool.

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 43–54, 2008.
© Springer-Verlag Berlin Heidelberg 2008

2 Background

During the software design process, we make many kinds of decisions. Some decisions can be traced directly to some element in the design or in the code. Other decisions are more difficult to track down to any concrete artifact as they span over many different elements or specify certain properties of the design/system. There are also decisions related to business/organizational issues. People frequently change their minds on decisions and the shift to agile software development methods imply that decision changes will be more common in occurrence. A single decision change could significantly affect the entire architecture of the software system being developed, implying the need to clearly document and model the role of design decisions and knowledge into software architecture.

2.1 Design Decisions and Software Architecture

What differentiates architectural design decisions from other design decisions is that the former are decisions that cross-cut multiple components and connectors and intertwine with other design decisions [14]. If one of the architectural decisions is changed, then the components dependent upon the changed decision would be impacted; furthermore, the change may affect other decisions that are intertwined with or related to the changed decision. Representing decisions as explicit, formal decision entities within architecture and their descriptions may assist in determining the severity and scope of a change. Decision entities make interdependencies more apparent and helps identify the set of decisions and architectural components that cause design rule or constraint violations.

The current shift is towards making design decisions and assumptions explicit within architectural descriptions [11, 14, 17]. Since these decisions intertwine and cross-cut architectural components, a decision view into software architecture [11] is fitting and we can model software architecture involving the use of formal decision entities. Software architectural models that use explicit decisions or assumptions include Lago and van Vliet's assumptions meta-model [19], the Archium metamodel [14], the ADDSS metamodel [9], and the architectural decision ontology described by Ackerman and Tyree [2].

2.2 Design Decision Representation

Many architectural models identified that the decision itself should have a model of its own to represent the architectural knowledge it carries. There are two main streams in the capture and representation of architectural knowledge as design decisions. The first stream is to use an argumentative approach via design rationale and the second stream is to use structured decision entities.

2.2.1 Design Rationale

Design rationale documents the analysis history of why particular artifacts or features are chosen [21]. They can also refer to non-functional requirements and constraints imposed on the general nature of the system. Design rationale commonly employs the use of an argumentation structure, which improves the capturing process as the

knowledge can be expressed in familiar forms such as issues, alternatives, questions, and options [12]. In essence, they take the perspective that decisions are context dependent, so the context and background must be captured in detail. Although the decisions can be captured using comfortable and relatable structures, it is difficult to extract the architectural decisions from the rationale as the decisions are embedded within the justification texts of the design rationale. The more expressive form of using design rationale makes decision referencing more difficult.

2.2.2 Explicit Decisions

The second stream is representing design decisions explicitly, viewing the choice as primary and the context and justification secondary to that decision. Although research in the design rationale community has dealt with representing decisions and assumptions explicitly, such as SIBYL [20], the software architecture community developed this area significantly due to the software architectural shift towards making design decisions and assumptions explicit. We introduced an approach that makes decisions first-class citizens by representing decisions using a decision ontology model [17]. Jansen and Bosch's architectural model [14] as well as Lago and van Vliet's work [19] also makes contributions to what is considered part of the decision model.

Though there are multiple approaches in representing design decisions and other architectural knowledge, the definitions are starting to converge. An attempt to define relevant architectural knowledge [8] by generalizing the principles of what would constitute architectural knowledge brings together other definitions of architectural knowledge, such as our decision ontology model [17], Bosch's four decision aspects [5], and Tyree and Ackerman's decision description template [24]. A recent literature survey by de Boer and Farenhorst [10] collected and synthesized definitions of architectural knowledge to conclude that all authors considered design decisions to be a significant part of the knowledge, and more definitions will come as the concept matures.

3 Decision Exploration and Visualization Tools

The formalized structure of explicit design decision representation in software architecture offers high decision analysis and exploitation potential. However, the analytical and exploitative capabilities of architectural design decision representation are bound by the way the information is organized or rendered. If the decisions are not effectively conveyed to the software architects, designers, and developers, then we question the usefulness of capturing those decisions; consequently, architectural design decision tools implementing explicit decision representation would need to adequately support decision exploration.

Design decision visualization facilitates understanding of the architecture and allows a kind of "walkthrough" of the designers' intents. Visualization is one of the five requirements listed in the decision view of software architecture [11]. These requirements are: multi-perspective support, visual representation, complexity control (in terms of scalability and navigation), groupware support, and gradual formalization of design decisions. The five requirements were described later to apply to all tools that utilize or manage design decisions [9].

3.1 Current Design Decision Tool Support

The recent interest in design decisions stimulated the development of several decision-based architectural tools. There are a number of tools created recently for the exploration and analysis of design decisions; some are from the design rationale community, some are from the architecture community. We will briefly look at a few of these tools in the context of decision visualization and exploration.

Although it is a design rationale tool, the SEURAT tool is an Eclipse development environment plug-in utility that captures and utilizes design rationale by linking its software code [7]. Decisions are visualized as part of the rationale in hierarchical tables displayed in Eclipse "views". Since the goal of SEURAT is to assist in software maintenance, the application to software architecture is not explicitly made. Another rationale-based tool, Sysiphus, is a toolset that assists in the capture of various system models for system development activities [6]. It supports rationale-based design decisions and links them with system models. However, with the focus being on collaboration support and on multiple system models, decision relationships and states are not investigated in detail.

The Archium tool is an architectural design decision tool which primarily focuses on how design decisions can be traced to the requirements and to the architectural components of a software architecture [15]. Although the tool's decision visualization component uses a graphical view of decision relationships, Archium regards design decisions as a "change function" with a single parameter [14], and the visualization of the decision entities themselves are light. Decisions are visualized with a dependency graph and the properties of the decision are listed in a table of attributes. Each decision can be linked to a graphical presentation of the architectural model, showing the components and connectors that relate to the design decision.

The ADDSS tool is a web-based tool to capture and document architectural design decisions for immediate browsing [9]. The tool lists the system requirements, the decisions and the requirements it addresses, and user-uploaded picture files representing the architectural products in a table format. Currently, the uploaded picture files from ADDSS approach graphical representation of architecture; however, another version is being developed to address the requirements stated in their follow-up paper [8]. Another web-based design decision tool, known as PAKME, focuses on general architectural knowledge capture and management of scenarios, patterns, design options, and decisions for the software architecture process [3, 4]. All these components, including decisions and their relationships, are displayed in tables and are retrieved by query-based mechanisms.

Although not officially designed to be tool-based work, there was a case study that focuses specifically on the ontological visualization of design decisions [18]. Design decisions followed the decision ontology model [17] and applied a visualization framework that visually clusters decision entities together depending on the query. The case study revealed that many architectural knowledge use cases can be supported by the clustering tool, but further tool extension is needed to explore the ideas of inter-decision relationships and the amount of information that relationship analysis can provide.

IBM research has also developed a tool, called the Architect's Workbench, which acknowledges the importance of decisions, but like many other tools, it has no

obvious ways to visualize webs of decisions [1]. There are other successful knowledge visualization tools that document decisions such as the Compendium tool [23], which documents and visualizes the flow of knowledge and design rationale during interactive team meetings. However, these tools are of a general scope and do not apply well to the interrelated, dynamic nature of software architectural design decisions and the multiple perspectives these decisions can have.

In many of the tool examples above, decision visualization would enable the attention to be directed towards specific areas of interest where useful conclusions may be drawn; yet, the majority of the tools did not significantly investigate the concept of decision visualization as a separate decision representation view or component. Visualizing the decisions using different perspectives may improve the quality of the software design by helping software architects, designers, and developers understand the nature and impact the design decisions they made.

4 Tool Implementation

We implemented a tool that visualizes software architectural design decisions separately from the software architecture in which they are referenced. The purpose of this tool is to facilitate both decision browsing and detailed decision analysis.

Although there are various models to represent software architectural design decisions, we excluded rationale-based decision models since those models detract attention away from the core decisions. We are primarily interested in how architectural design decisions can be exploited, so a simpler, broader model that addresses software design decisions in general is preferred. Decision models that architects and designers can use more easily during software development are ideal. We decided to adopt Kruchten's decision model [17] because the model is simpler, more process-focused, and it is the only model that explicitly represents decision *relationships*.

The tool captures design decisions and stores them into a file or a database for later retrieval. Moreover, the decisions can be imported or exported using XML across multiple computer workstations to ease decision capture and distribution. The user can create, modify, remove both decisions and their interrelationships, and visualize the decisions in several ways to support decision perusal and analysis. The tool utilizes the Prefuse visualization framework [13] for the visual representation of design decisions. The tool has four main views for decision visualization and information display. The first is a simple tabular list of decisions and their relationships, while another view visualizes the decisions using decision-graphs to display the decision structures and relationships. The tool can also visualize the decisions in a chronological order. The fourth view displays decisions from an impact perspective.

4.1 Decision / Relationship List

This view, a *decision table*, is the most common in the decision tools, and almost every tool mentioned in Sect. 3 supports this view. The decision / relationship list simply lists the design decisions in a table, showing a selection or all the attributes of a design decision. Decision relationships are also listed in another table that references the decision list. A screenshot is depicted in Fig. 1. The purpose of this view is

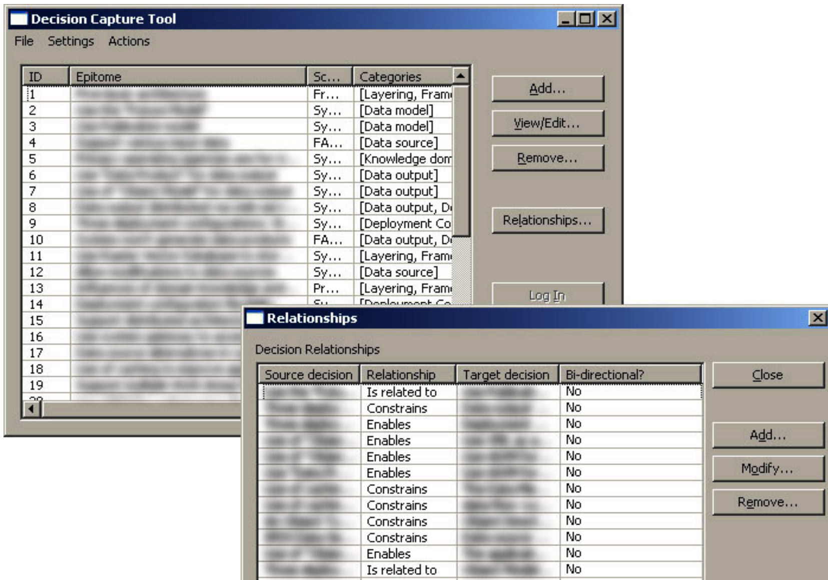


Fig. 1. Decision and relationship lists showing the current set of design decisions and their relationships: the user can create, view, modify, or remove design decisions directly from the tool. In this figure, the "relationships…" button has been pressed, bringing up the relationships dialog that is currently displayed in the foreground. The decision epitome has been intentionally concealed to protect the intellectual property of the organization that provided the dataset.

to supply a quick and effective way to browse and retrieve information from design decisions. The textual representation of the decisions facilitates decision querying and simple decision entry. However, it is difficult to trace decision relationships and quickly assess decision properties when the decision set gets large.

4.2 Decision Structure Visualization

With large decision sets, an effective way to sort and analyze decision information is to represent the decisions graphically. In this view, we visualize *decision structure* as graphs, in which decisions are represented as nodes and the relationships are the edges. Figure 2 depicts a decision graph that represents the decisions and their relationships. Decisions and relationships can be created, selected, viewed, modified, and removed from this view. The advantages of graph visualization are apparent, such that an observer can see relationships and their associated decisions more quickly than from a list.

Besides the view's graphical visualization, there is a high degree of interactivity to communicate information. Using a force-directed layout for the visualization of the decision graph, the tool represents decisions of a less mature state as being physically lighter in the layout model and visually smaller than more mature decisions. Decision maturity refers to the decision states in Kruchten's decision ontology model, where a decision can be an idea, tentative, decided, approved, challenged,

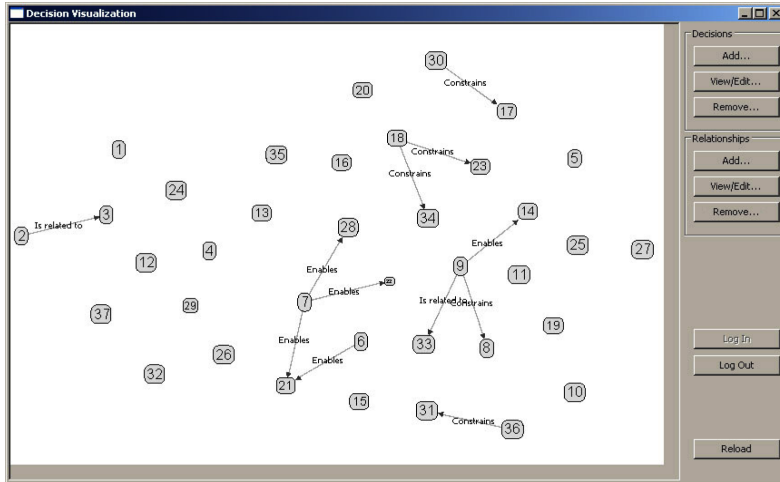


Fig. 2. The visualization of design decisions and their relationships as a directed graph: the nodes represent the decisions and the directed edges represent the decision relationship to another decision. The node-size depends on the state of the decisions. Not shown in this figure are the semantic zooming properties that can list decision properties like the decision's epitome, and the interactivity provided visualization where decisions at less mature decision states can be moved around the visualization more easily than other decisions.

rejected, or obsolete. Visually, the maturity of a design can be assessed from the number of small or large nodes in the graph. However, when the user interacts with a decision node or a cluster of nodes, the user can quickly assess the maturity from how quickly the decision can be moved around the screen. For example, more mature decisions are heavier and more difficult to move, so the decisions behave like heavy objects.

Depending on the zoom level, the decision nodes can show more or less information about the decision. When a user zooms towards a decision, the decision's properties will appear inside the node. When a user zooms away, decision information gets hidden. Viewing the decision or relationship details can also be performed without zooming simply by selecting a decision.

4.3 Decision Chronology Visualization

The tool supports a time-based view of design decisions, the *decision chronology*, to show the evolution of design decisions and provide the ability to quickly determine created or changed decisions during a specified time interval. This view is shown in Fig. 3. A user can select a subset of these decisions to view more closely, such as the decisions within a cluster, and can create, view, or modify decisions. This feature is especially valuable when there are periodic reviews of the architecture: it saves time and effort for the reviewers who are already familiar with the system, who may only want to know, "What has changed since last time?"

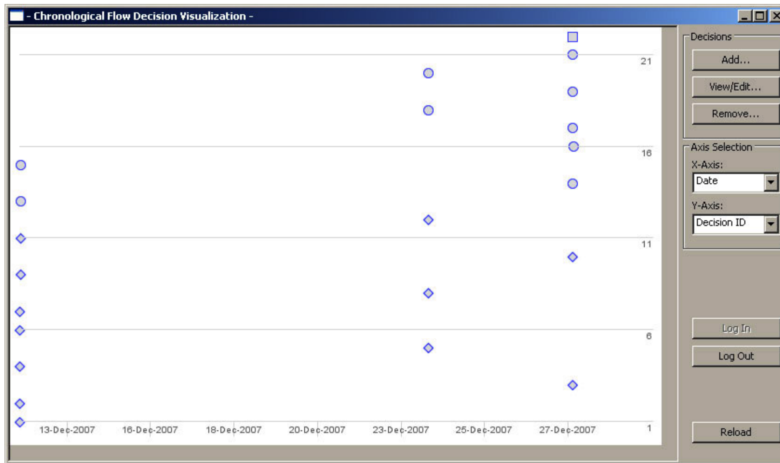


Fig. 3. The chronological view of a set of design decisions: this example shows three decision creation or activity sessions over a two-week interval. The state of the decisions is denoted by the shape: Diamonds are idea, circles are tentative; and squares are decided.

This view initially displays all the decisions created and modified during the project in a timeline, with the date on the x-axis and a user-selectable field for the y-axis. Decisions that are closely spaced denote a decision capture or activity session. A user can quickly identify the state of a decision by its shape in the view.

A particular area of interest is in the user-selectable y-axis. The tool currently allows categorization of the y-axis by decision ID or decision author. If the decision ID is used for the y-axis, one can view decision changes in a global perspective (as the decision ID is implemented as an increasing number). If the author is used for the y-axis, we can determine which decision-makers are most active and which changes they have made. Categorizing by author include the ability to find both subversive and critical stakeholders who can potentially damage the system if they change their minds [18]. With other category types for the user-selectable y-axis, the tool can be a powerful way to exploit hidden knowledge within design decisions.

4.4 Decision Impact Visualization

The fourth view of design decisions that this tool supports is the *decision impact*. Shown in Fig. 4, this view provides a visualization of decisions that can be potentially impacted by a change of a decision. This view is very valuable when radical changes are about to be made to a system, and the impact of certain changes may not be obvious in the architectural design or the code. Although the decision structure visualization supports visualization of decision relationships, there are related decisions that are associated by attributes, such as author, scope, and categories. The tool provides an entry-point into the large matrix of potential impact-relationships by visualizing it.

The decisions are laid out using a radial layout, where all other decisions surround the selected center decision. Selecting a different decision brings that decision into the center and all other decisions surround it. Resting a mouse cursor on a decision would

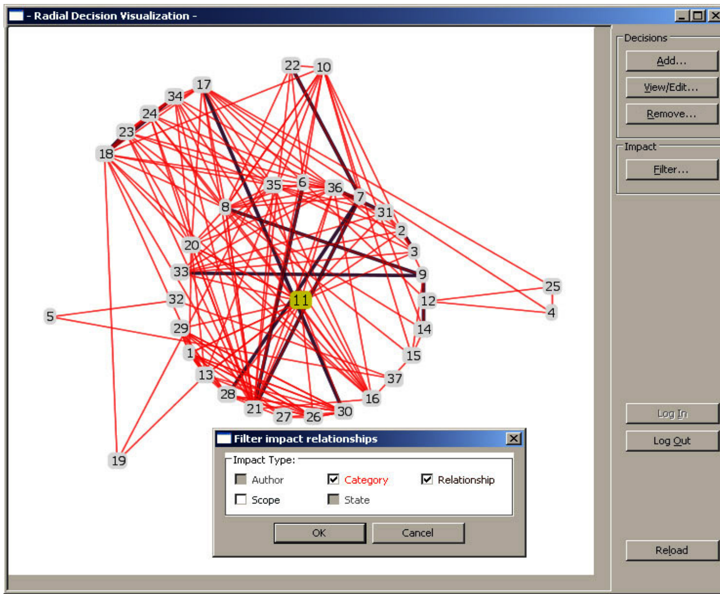


Fig. 4. The decision impact view of design decisions: the nodes represent design decisions while the colored lines represent the impact-relationships between them. Thick edges are the decision relationships in our decision ontology model [17], thin edges are impact-relationships.

highlight neighboring decisions associated with an impact-relationship. The impact relationships can be filtered according to different criteria, such as category, scope, or relationship. Currently, the tool links decisions that share a common criteria value with an impact-relationship, though the tool can be modified to support different criteria values, ranges, and thresholds.

5 Experience with the Tool

We were able to use industry datasets to test the practicality of the decision visualization aspects. We demonstrated and used the tool with industry participants and obtained feedback on the feasibility of the visualization tool.

One group of industry participants is from a large technology corporation that is both process and documentation heavy. The participants are involved in a multinational project to develop an elaborate spatial modeling system, but the system is constrained by many domain-specific guidelines and procedures. As the project is very large, we reduced the scope and focused the decision capturing on the deployment configurations and the data model used for this system.

After demonstrating the tool to the participants using their own decision dataset, we asked the participants what their initial impressions are. They found that the decision and relationship lists were acceptable, but for the graphical decision structure visualization, the participants stated that the decision identifier used in the default zoom-level is not fully intuitive, as it can be hard to mentally map decisions details to the decision identifiers. The participants mentioned that the decision-relationship

graphs are informative, but they reported that the explicit decision relationships are difficult to elicit and categorize, partly due to the various relationship definitions and the tacit nature of defining these relationships. The participants appreciate the ability to see decision sessions in the decision chronology view, and they found the "author" criterion for the user-selectable y-axis to be an interesting application. Although the participants felt that implementing a fine-grained filtering mechanism would improve usability, all the participants agree that the decision impact view can be effective in identifying potential, indirectly-impacted decisions.

From the initial feedback, we were able to validate the feasibility of the visualization tool with industry datasets in a laboratory setting. However, we would like to validate the tool in industry, allowing software designers, architects, and developers to use the tool firsthand and investigate the exploratory and analytical capabilities of the visualization tool.

6 Future Work

The four aspects that the visualization tool supports provide the user with a powerful means to explore and analyze decisions with. This user can be a software architect, a designer, a developer, or any other individual who would like to learn and explore the decisions for a project. We are continuing to develop the tool further, improving the user interface and providing more useful features to support the exploration and analysis of decisions. Full query-support is a feature we would like to implement. Another useful feature is to link decisions across the four different views. Selecting a decision in one view should select the same decision in another view. This way, the user can maintain continuity between views and may discover new associations about the design decisions. We would also like to implement fine-grained impact querying as suggested by the participants, after which we could implement multi-level impact visualization to improve usability.

Although we have identified four aspects, there may be other visualization techniques that can reveal more information provided by the set of decisions. Ontological visualizations [18] can be added to assist in decision retrieval and categorization, while support for visualizing software and organizational artifacts may provide insight by associating decision capture with the software development process. Furthermore, visualizing the links between artifacts and design decisions may improve traceability between requirements, architecture, and developed software code.

7 Conclusion

The tool we presented here has been developed and used with industry datasets from several software organizations. The screenshots displayed in the preceding figures demonstrated early results on the feasibility of the visualization tool as applied to industry, but further evaluation is needed to identify additional use cases in which the visualization tool may be useful for industry practice. The decision sets acquired from industry show some encouraging early results, but further detailed case studies with industry participants may be necessary to apply other visualization techniques to better capture, represent, and relay software architectural knowledge.

References

1. Abrams, S., et al.: Architectural thinking and modeling with the Architects' Workbench. IBM Systems Journal 45(3), 481–500 (2006)
2. Akerman, A., Tyree, J.: Using ontology to support development of software architectures. IBM Systems Journal 45(4), 813–825 (2006)
3. Babar, M.A., Gorton, I., Jeffery, R.: Capturing and Using Software Architecture Knowledge for Architecture-based Software Development. In: 5th International Conference on Quality Software (QSIC), Melbourne (2005)
4. Babar, M.A., Gorton, I., Kitchenham, B.: A framework for supporting architecture knowledge. In: Dutoit, A.H., et al. (eds.) Rationale Management in Software Engineering, pp. 237–254. Springer, Heidelberg (2006)
5. Bosch, J.: Software Architecture: The Next Step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, Springer, Heidelberg (2004)
6. Bruegge, B., Dutoit, A.H., Wolf, T.: Sysiphus: Enabling Informal Collaboration in Global Software Development. In: First International Conference on Global Software Engineering, Costao do Santinho, Florianopolis, Brazil (2006)
7. Burge, J.E., Brown, D.C.: Rationale-based Support for Software Maintenance. In: Dutoit, A.H., et al. (eds.) Rationale Management in Software Engineering, pp. 273–296. Springer, Heidelberg (2006)
8. Capilla, R., Nava, F., Duenas, J.C.: Modeling and Documenting the Evolution of Architectural Design Decisions. In: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent, IEEE Computer Society, Los Alamitos (2007)
9. Capilla, R., et al.: A web-based tool for managing architectural design decisions. SIGSOFT Software Engineering Notes 31(5) (2006)
10. de Boer, R.C., Farenhorst, R.: In Search of 'Architectural Knowledge'. In: Third Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent, IEEE Computer Society, Germany (2008)
11. Duenas, J.C., Capilla, R.: The Decision View of Software Architecture. In: 2nd European Workshop on Software Architecture, Morison, Italy (2005)
12. Dutoit, A.H., et al.: Rationale Management in Software Engineering: Concepts and Techniques. In: Dutoit, A.H., et al. (eds.) Rationale Management in Software Engineering, pp. 1–48. Springer, Heidelberg (2006)
13. Heer, J., Card, S.K., Landay, J.A.: Prefuse: a toolkit for interactive information visualization. In: SIGCHI conference on Human factors in computing systems (2005)
14. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Working IEEE/IFIP Conference on Software Architecture (WICSA) (2005)
15. Jansen, A., et al.: Tool support for architectural decisions. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), Mumbai (2007)
16. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd edn. Addison-Wesley, Boston (2003)
17. Kruchten, P.: An Ontology of Architectural Design Decisions. In: 2nd Groningen Workshop on Software Variability Management, Rijksuniversiteit Groningen, NL (2004)
18. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
19. Lago, P., van Vliet, H.: Explicit Assumptions Enrich Architectural Models. In: International Conference on Software Engineering (ICSE 2005), ACM Press, USA (2005)

20. Lee, J.: SIBYL: a tool for managing group design rationale. In: ACM conference on Computer-supported cooperative work (CSC1990), Los Angeles (1990)
21. Lee, J., Lai, K.-Y.: What's in Design Rationale? In: Design Rationale: Concepts, Techniques, and Use, pp. 21–51. Lawrence Erlbaum Associates, Inc, Mahwah (1996)
22. Lee, L., Kruchten, P.: Customizing the Capture of Software Architectural Design Decisions. In: 21st Canadian Conference on Electrical and Computer Engineering, IEEE, Los Alamitos (2008)
23. Selvin, A., et al.: Compendium: Making Meetings into Knowledge Events. In: Knowledge Technologies, Austin, TX (2001)
24. Tyree, J., Ackerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22(2), 19–27 (2005)

Style-Based Model Transformation for Early Extrafunctional Analysis of Distributed Systems

Julien Mallet and Siegfried Rouvrais

Institut TELECOM; TELECOM Bretagne
Technopole Brest-Iroise, CS 83818, 29238 Brest Cedex 3, France
{julien.mallet,siegfried.rouvrais}@telecom-bretagne.eu

Abstract. In distributed environments, client-server, publish-subscribe, and peer-to-peer architecture styles are largely employed. However, style selection often remains implicit, relying on the designer's know-how regarding requirements. In this paper, we propose a framework to explicitly specify distributed architectural styles, as independent models of the application functionalities. To justify feasibility and further benefits of our approach, we formally define three classical distributed architectural styles in a process calculus. Our proposal then opens up the way to a systematic composition of functional models with architectural style models as an endogenous transformation. Comparative analysis of extrafunctional properties could then be proposed at the early design stages to guide the architect in stylistic choices.

1 Introduction

Architectural styles build up conventional structures for designing large systems at a software architecture level. Different architectural styles enforce different quality attributes for a system [1]. Within distributed systems, an application often relies on an architectural style which defines connections between application components (*e.g.* simple message interaction models, client-server, publish-subscribe, peer-to-peer). Most often, style selection remains implicit and tacit [2], relying on the architect's know-how regarding requirements. Choosing an inappropriate architectural style can lead to major impacts on the properties of a system or application [3]. Moreover, extrafunctional properties such as security, performance, reliability or scalability are not easily grasped at an abstract description level. Such concerns thus tend to be forwarded to the end of the design process lifecycle, though they are rough to manage once a style has been selected and a system designed. They are however the critical selection criteria to better manage the development process, regarding system's internal and external properties.

Specifying a distributed system's software architecture classically requires to model architectural components and connectors, and some of their extrafunctional properties. However connectors, as communication mediums, are parts

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 55–70, 2008.
© Springer-Verlag Berlin Heidelberg 2008

of distributed styles having their own comprehensive, intrinsic, and emergent properties. To manage extrafunctional properties at early design stages, we propose to specify distributed architectural styles independently of the functional model. By separating concerns in a framework, we then propose a model transformation corresponding to a composition of an abstract functional model with styles predefined in a repository. Functional and especially extrafunctional analysis could then be investigated to compare models and guide the architect faced with several distributed design alternatives. To justify the approach, we restrict to three common distributed styles descriptions in this paper, using structure diagrams and process calculus: client-server [4], publish-subscribe [5] and peer-to-peer [6]. A distributed version control system with its functional model is proposed as a case study.

The remainder of this paper is organised as follows. Section 2 introduces our framework and proposes some common distributed architectural styles and their specifications using a process calculus. Section 3 presents an independent functional model and proposes a specification example on the version control system case study. The systematic composition of a functional model with an architecture style model is described in section 4 through the application example. Section 5 addresses extrafunctional properties integration in the framework, while section 6 presents related work. Finally, section 7 concludes this paper with a summary and an outline of further research.

2 A Framework with Distributed Architectural Styles

The motivation of the proposed framework presented in figure 1 is to guide the architect in choosing the right distributed architecture style in conformance with extrafunctional requirements. It follows a model driven engineering approach [7] and addresses the quality of the target system's software architecture for early design decisions. First, the designer specifies system functionality using

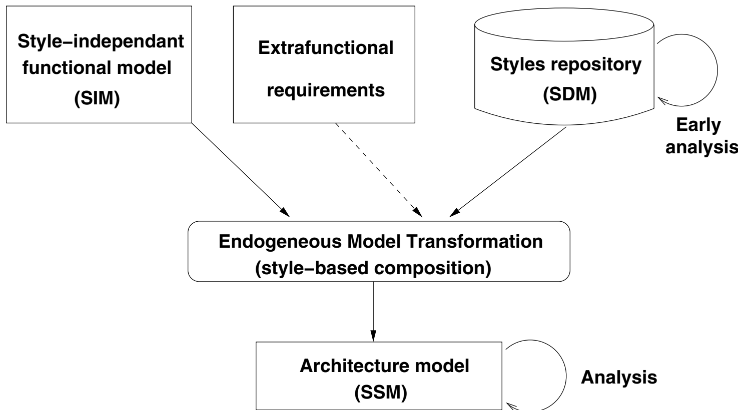


Fig. 1. The overall framework

a style-independent model (SIM). This model is purely functional: the services are provided in an ideal structureless-environment. However, application components in a distributed system require some interaction mechanisms for coordination and communication. As early proposed in the software architecture community [8], architectural styles provide conventional structures for building large systems [9,10]. Such structures are the primary models for distributed interactions. Connectors, as interaction mechanisms, are the principal structural elements for a boxology [3] of distributed architectural styles. Within distributed systems, the client-server architectural style [4], based on the request-reply protocol, is still predominant. With Web generalisation, related styles like service-oriented or *Representational State Transfer* (REST [11]) architectures are now taking a major position. So far, mechanisms for coordination and communication based on push-like models (*e.g.* message sending) or pull-like models (*e.g.* request-reply) are structural elements of architectural styles. Thus, we propose to gather distributed architecture styles in a repository of style definition models (SDM). Those models, having their own extrafunctional characteristics, can be early analysed in light of extrafunctional requirements.

Extrafunctional properties (*e.g.* security, performance, scalability) are key elements to guide the design decisions. Our transformation model consists in composing the required functional services with a given distributed style available in the repository. A distributed architectural style candidate can first be selected in front of quality attributes. The resulting style specific model (SSM) represents a possible system's architecture model which can be compared with other SSMs using functional and extrafunctional analysis. The framework makes it possible to choose, at upper stages, possible styles thanks to the extrafunctional properties. After analysis, if a proposed style specific model does not guarantee the extrafunctional requirements, the architect may modify or weaken some of the requirements until finding an appropriate style, or introduce specific mechanisms (*e.g.* patterns) in the specified system's software architecture to satisfy requirements.

2.1 Three Classical Distributed Architecture Style Models

Client-server, publish-subscribe, and peer-to-peer styles are largely used for distributed applications. Such styles encourage reusability, system comprehension, and analysis by using well-known interaction mechanisms. These mechanisms predominantly rely on push or pull models. Style variants exist, but they share common characteristics at an abstract level. In the classical client-server style, a client component requests a service through a remote invocation to a server component. Often synchronous, this interaction mechanism follows a pull model based on a request-reply protocol. In the publish-subscribe style, components are either announcers or listeners of events. By registering through an event manager, listeners are asynchronously informed of events most often through a push model. In the decentralised peer-to-peer style, where a peer represents a component, overlays, as logical networks, are dynamically constructed or maintained. For instance, by using a pure pull model between peer neighbours or by

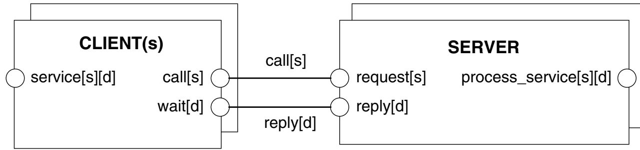
combining the pull with a push model restricted in depth, this style is much adopted in mobile or ubiquitous environments.

The distributed architecture styles have emergent extrafunctional properties. For example, the publish-subscribe and peer-to-peer styles are mostly known to be scalable and reliable. Moreover, the publish-subscribe style generally guarantees the anonymity of the announcers. However, these intrinsic properties mainly arise from empirical studies and not from systematic evaluations on style models.

2.2 Modelling Architectural Styles

Magee and Kramer [12] provide elements of style specification using the Finite State Processes (FSP) process calculus. For our purpose, we expand their style examples to create a first repository of distributed styles, independent of any application functionalities. Other formal approaches could have been addressed, but FSP process calculus, with its associated LTSA tool (*i.e.* model-checker), is suitable for a comprehensible demonstrator for a model transformation. For the sake of clarity, the three classical distributed styles addressed in this paper are shown hereafter as structure diagrams of processes (*i.e.* components as boxes in figures) and ports or events (*i.e.* bullets in figures). Note that the structure diagrams are only graphical representations of the FSP expressions as defined in [12] (*e.g.* the notion of provided/required ports does not exist). Elements of syntactical FSP expressions for client-server and publish-subscribe styles can be found in [12]. We detail the FSP expression only for the client-server style.

Figure 2 presents the generic client-server style, through the structure diagram, where several clients call services from the server and obtain the associated result. The clients are introduced by the processes CLIENT (stacked boxes represent identical processes). A given client will always request the same service




```
set Data = {d1,d2,d3,d4}
set ServiceId = {s1,s2}
set Clients ={c1,c2,c3}
CLIENT(S='si)
  = (call[S] -> wait[d:Data] -> service[S][d] -> CLIENT)+{call[ServiceId]}.
SERVER
  = (request[s:ServiceId] -> process_service[s][d:Data] -> reply[d] -> SERVER).
||CS_EX
  = (c1:CLIENT('s1) || c2:CLIENT('s2) || c3:CLIENT('s1) || Clients:SERVER)
      /{forall [c:Clients]{[c].call/[c].request,[c].reply/[c].wait}}.
```

Fig. 2. Structure diagram and FSP expression of a client-server style

(introduced by its parameter s). Furthermore, each CLIENT process is statically linked with one distinct SERVER process that responds to its request (there are as many SERVER as CLIENT processes). For model transformation, in order to further compose the style-independent and style definition models, the clients provide a service[s][d] event corresponding to an external call to the service s that returns a result d, and the server offers a process_service event corresponding to an external service computation.

In addition, figure 2 presents the corresponding FSP expression and an example of an instanced client-server style (process ||CS_EX). Three sets are introduced: Data as the possible results, ServiceId as the service names offered by the server and Clients as the identifiers of the client processes. A CLIENT process calls the service (event call[S]), then awaits synchronously the result (event wait), forwards it through the service event to external components and iterates. Similarly to [12], the CLIENT process uses the alphabet extension operator (noted +) in order to ensure a suitable synchronisation between the clients and the server and takes the service name as parameter (its default value is si). The SERVER process awaits a service request (event request), then requests the result computation (event process_service) and returns the result (event reply). Finally, the ||CS_EX client-server style example is the parallel composition of the clients (in our case, three clients: c1, c2 and c3) with as many server processes prefixed by the identifier of the corresponding client (expression Clients:SERVER). Corresponding to architectural attachments, the mapping between client and server events associates, respectively, the call and wait events of the client with the request and reply events of the corresponding server.

Note that the definition of the client-server style is generic : we just have to define the Data, Client and ServiceId specific sets in order to instanciate the style to a client-server application.

Relying on [12], figure 3 presents the publish-subscribe style where one announcer (process ANNOUNCER) publishes events to zero or more listeners (process LISTENER). The EVENTMANAGER process carries out the event broadcasting. A listener can register his/her interest in a particular pattern p with the event manager through the register[p] event. Each time the announcer produces the pattern, only the registered listener is notified. Finally, a listener can deregister himself through the deregister event. In order to specify this style in FSP, one event manager process is introduced per listener. When an event announcement is produced, the event managers forward it to their associated registered listeners.

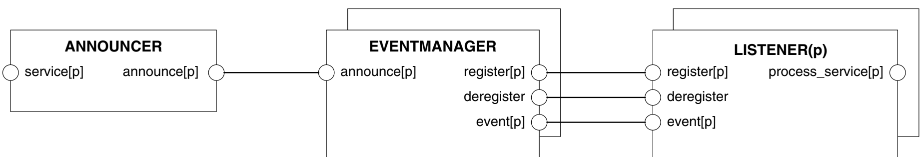


Fig. 3. Structure diagram of a publish-subscribe style

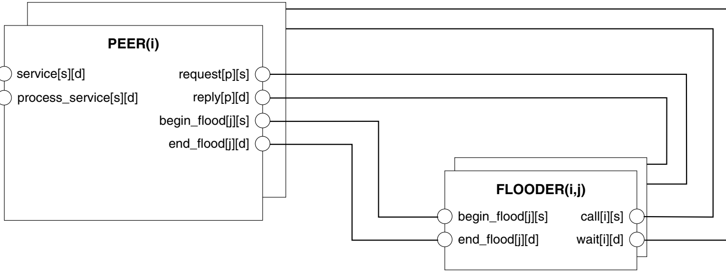


Fig. 4. Structure diagram of a pure peer-to-peer style

Finally, figure 4 presents the peer-to-peer style in its flooding version where peers collaborate through a pull model (*e.g.* like in the Gnutella scheme). Since many variants of peer-to-peer styles exist, we limit our example to the simple flooding version. Each peer is represented by a PEER process taking its number as parameter. It offers two services as interface: service[s][d] for requesting a service s returning the result d and process_service[s][d] for requesting external components. A FLOODER process per peer is associated with each link between peers. In the figure 4, the peers of number i and j are neighbouring. The FLOODER sends the requests to the peer neighbours (event call) then awaits the response from the latter (event wait). Therefore, the topology of the considered peer-to-peer network is modelled by the FLOODER and PEER links. At the time of a service[s][d] event, the PEER process initiates the flood request to its neighbours with the begin_flood event. The FLOODER process carries out the flood, then obtains the result d thanks to the wait[i][d] event. Finally, a PEER process receives the result through the end_flood event and transmits it to the requester.

The styles presented above and contained in the repository are distributed architecture styles rather than communication ones in the sense that they describe interaction between components. Due to the style transformation (presented section 4), the intended interaction mechanisms will be introduced into the functional model using the selected style model.

3 Functional Model

In the framework, the designer first specifies his system in a style-independent model, with a pure functional point of view set in an ideal non-constrained environment: extrafunctional properties are not taken into account, and architectural stylistic elements are abstracted.

Thereafter, we exemplify this principle on a version control system as a classical distributed case study. A version control system (*e.g.* CVS) allows several users/developers to modify a set of shared files concurrently. Each developer has a copy of the files (most often in a repository) which he/she can modify locally, using a write command. The local modifications are spread to other developers

with a commit command, using the versioning system. Update command brings local copies up-to-date with the last shared version.

3.1 Pure Functional Model of a Versioning System

The style-independent functional model of a distributed application is defined by introducing, for each user, the three following components (defined by one or more processes as the application requires):

– **User:** operations from the user's point of view (*i.e.* update, write, and commit in the example). It introduces the available services thanks to shared events;
– **RemoteState:** operations for data information obtained from other users due to distribution (*i.e.* future components for interaction mechanisms);
– **Safe:** business rules specified through authorised sequences of events in the system, to guarantee application requirements in functional terms.

Our abstract level approach is general and can be applied to other distributed applications as long as they could be decomposed into the three previous components. For our case study, each preceding component is specified by one FSP process (*i.e.* User, RemoteState, Safe) as presented in the structure diagram of figure 5. Each User process of the system takes a unique user number I as parameter. It holds and updates the state of its local copy of the repository (*i.e.* either it is identical to the reference repository or it contains some update). Any user can modify the file locally (event write), update his/her local copy with the repository (event update) or update the repository with his/her local copy (event commit). Moreover, the User process offers a service to read its local state thanks to the localState event. In the structure diagram, the User(I) and User(J) are similar except for their number. We distinguish them in order to present the specific shared events between User and RemoteState processes.

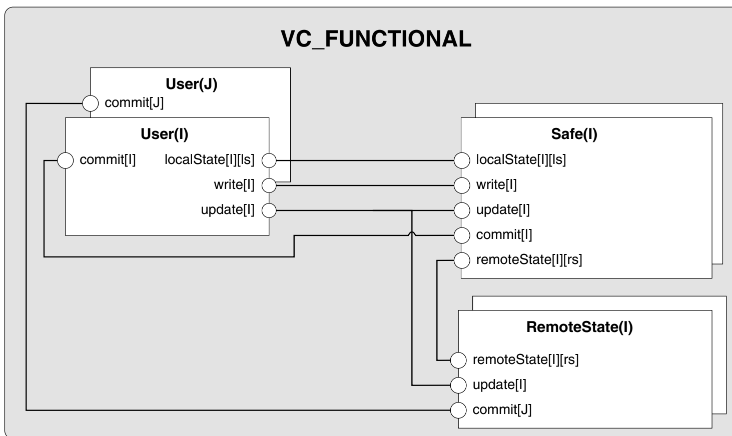


Fig. 5. Functional structure diagram of a source control management system

The RemoteState(I) process abstracts the overall remote state for the ith User process. The remote state is either remotely non modified (other users have not modified the repository since the last local update) or remotely modified (one or more other user has modified the repository). So, when another User(J) process, distinct from User(I), performs a commit, the remote state associated with I becomes remotely modified. When User(I) produces an update event, the remote state becomes remotely non-modified. Furthermore, RemoteState provides a service allowing its state to be read (event remoteState).

Finally, Safe(I) processes ensure the right update policy (*i.e.* each User(I) process has to obtain a local copy that is consistent with the repository before updating the repository with its changes). These processes define the functional properties that the whole system has to ensure by specifying the authorised sequences of events.

There are several implementations of versioning systems, introducing architectural styles more or less implicitly (*e.g.* CVS and Subversion rely on the client-server style, SVK uses the peer-to-peer style). As we can see in our case study, the functional model does not imply any style.

3.2 Facilitating Functional Model Generation

We propose to formally specify style-independent models at an abstract level using a process calculus. A functional model is a composition of processes having remote states definitions. However, specifying processes could be difficult for the designer unfamiliar with formal methods. Tools supporting functional model elaboration have been designed in the process calculus community. For instance, based on scenarios specified as sequence diagrams or message sequence charts, FSP expressions could be generated to assist the architect [13].

4 Functional and Style-Based Model Transformation

Once a functional model of the system has been specified by the designer, it can be related to a certain architectural style model taken from the repository. Thanks to process calculus specifications, the transformation is achieved by process composition and event renaming. This model transformation is endogenous, *i.e.* the source and target models are still FSP processes. The generic transformation process is based on the following steps:

1. Choice of an architectural style in the repository (note that the designer could be egged on a choice due to early analysis on style models);
2. Selection of functional model event(s) in order to introduce the style;
3. Definition of event relabelling between functional model and style.

These above steps are devolved to the designer. Then, based on them, a systematic transformation can be applied in order to obtain the style specific model. The target style specific model is simply a process parallel composition, where events have been relabelled. We detail two generic transformations on our case

study, respectively to produce client-server and publish-subscribe style specific models. The transformation will link the external events of a style (*e.g.* service and process_service for the client-server style) with shared events of the functional model (*e.g.* commit, update and remoteState for the case study). These later events can be seen as join points for the style introduction.

4.1 Two Style Specific Models of the Case Study

A Client-Server Versioning System. The client-server style is introduced through the remoteState, update and commit events shared by the User, Safe and RemoteState processes from the functional model detailed in section 3. The events are transformed into a request-reply interaction between clients and a server. The resulting structure diagram is given in figure 6. Three kinds of components are identified: CS_GEN, VC_CLIENTS, and VC_SERVER. CS_GEN is the client-server process described in section 2. VC_CLIENTS represents the n clients of the distributed versioning system, each one maintaining the state of the local copy of a user and ensuring the access policy to the repository. Finally, VC_SERVER is the system server that holds and maintains the remote state of each user.

Figure 7 presents the FSP expression (for simplicity, only the components previously described are given). The processes from the functional model are unchanged (*i.e.* User, Safe and RemoteSate). In this case, the system is composed of three users identified by u0, u1 and u2. ServiceId is the set of events used as join points in order to introduce the style. The ||CS_GEN component contains a CLIENT process per event and user. Thus, for the remoteState event, there are three CLIENT processes, prefixed by u1.remoteState, u2.remoteState and u3.remoteState respectively. The ||VC_CLIENTS component is the parallel composition of the User and Safe processes. Events are relabelled in order to

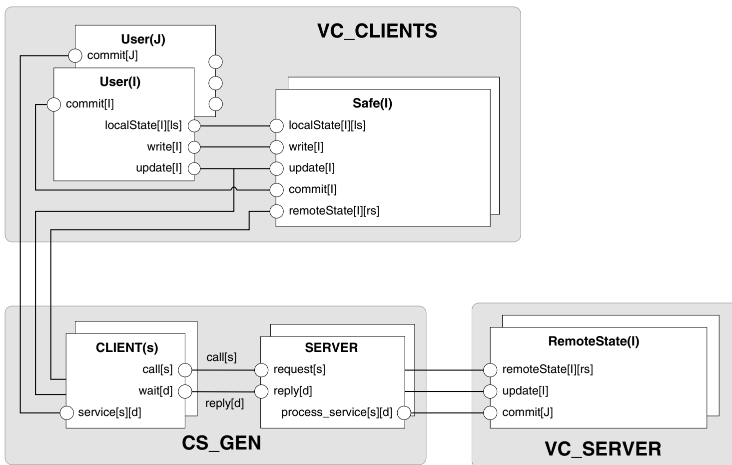


Fig. 6. Structure diagram of the client-server specific model

```
set Users = {u0,u1,u2}
const NUser = #Users
range U =0..NUser−1
set ServiceId = {remoteState,update,commit}
||CS_GEN
    = (forall[c:Users](forall[s:ServiceId]([c].[s]:CLIENT(s)))
        || Users[s:ServiceId]:SERVER)
       /{forall [c:Users]{forall [s:ServiceId]{
                          [c].[s].call/[c].[s].request,[c].[s].reply/[c].[s].wait}}}.
||VC_CLIENTS
   = forall[i:U](User(i)||Safe(i))
     /{forall[j:U]{[@(Users,j)].remoteState.service['remoteState]/remoteState[j],
                        [@(Users,j)].update.service['update]/update[j],
                        [@(Users,j)].commit.service['commit]/commit[j]}}.
||VC_SERVER
   = (forall[i:U](RemoteState(i)))
     /{forall[j:U]{[@(Users,j)].remoteState.process_service['remoteState]/remoteState[j],
                        [@(Users,j)].update.process_service['update]/update[j],
                        [@(Users,j)].commit.process_service['commit]/commit[j]}}.
||VC_CS =(VC_CLIENTS || CS_GEN || VC_SERVER).
```

Fig. 7. Case study client-server FSP expression

match the style events. For instance, the remoteState[0] event of User(0) is relabelled u0.remoteState.service['remoteState] (in FSP, the expression @(Users, j) denotes the jth element of Users). The ||VC_SERVER component composes the RemoteState processes and relabels events equally. Finally, the ||VC_CS final system is the parallel composition of the three previous components.

A Publish-Subscribe Versioning System. For the publish-subscribe style, we choose the commit event shared by the User and RemoteState processes in order to introduce the style. This event is transformed into a push mode interaction: each User notifies all the others that he/she has modified the repository. The resulting structure diagram is given in figure 8. Three kinds of component are introduced: VC_ANNOUNCERS, VC_LISTENERS and PS_GEN for the style. The first one contains the User processes that act as announcers of commit events. The VC_LISTENERS component contains the RemoteState processes acting as listeners of the same events. At commit time, the User(J) process announces the commit[J] event to the event manager, which broadcasts it to all the RemoteState processes. The users communicate through the event manager to notify repository updates.

We have shown throughout this example that a functional model could be systematically composed with different styles provided in a common repository. Nowadays, the transformations are handmade; we have not yet an automatic tool that produces the target style specific model but we plan to describe formally the style transformation as a FSP expression transformation. Indeed, the transformation consists in composing the FSP expression of the functional model with

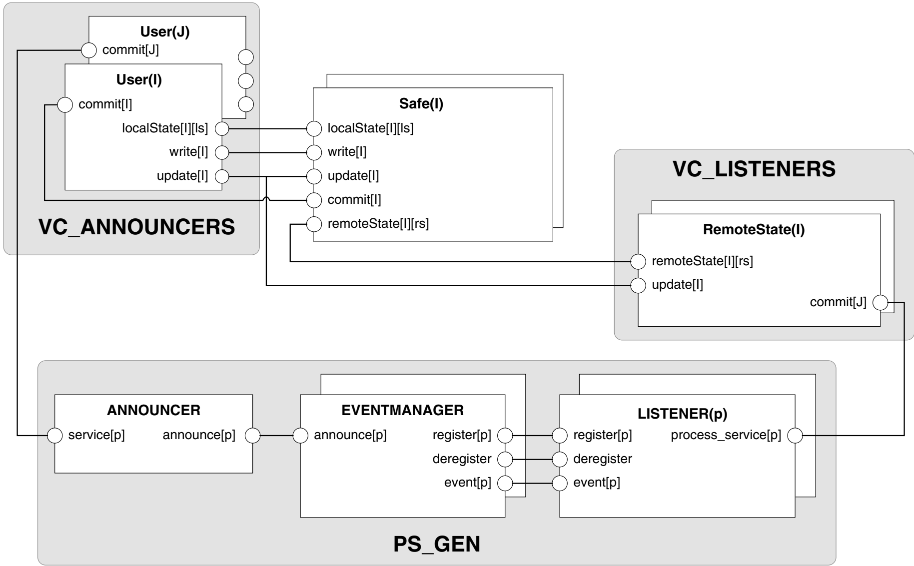


Fig. 8. Structure diagram of the publish-subscribe specific model

the chosen style one and relabelling the events in order to match the functional model ones with the external events of the style.

Further, style variants are also to be considered. In fact, RemoteState could be distributed either locally on a User or within a particular component. A range of style interaction models can then be made available according to the required distribution. After composition, a model checker is the primary tool used to verify conformity with functional requirements. A generated model can be verified through a LTSA model checker (*e.g.* liveness, progress). For the versioning system example, we can check that the update policy is preserved after the style introduction (*i.e.* there is no deadlock due to the SAFE processes). But extrafunctional properties should be the key concerns for selection.

5 Preparing Architecture Quality Analysis

For the moment, our framework focuses mainly on the functional services and properties of the system's software architecture. However, extrafunctional properties are also to be taken into account in the style definition models and in style-specific models so as to compare architectural choices. Some styles are intrinsically known to meet extrafunctional requirements more easily. In particular, peer-to-peer distributed systems are well recognised for scalability and reliability characteristics; publish-subscribe systems support extensibility, anonymity of actors and dynamicity of incoming/outgoing participants; client-server systems are often considered better for flexibility but worst for availability due to single points of failure.

Using our approach, some security properties such as message authenticity, confidentiality and integrity may be early verified, *e.g.* based on earlier formal work of Schneider [14]. Pursuant to this proposition, the system is specified in the CSP process calculus (quite similar to FSP) and includes an additional enemy process in order to model potential security attacks (*e.g.* message leakage, message alteration). The security properties are then introduced as properties on event traces. For example, for message confidentiality, it can be stated that each message received by the enemy process must have been sent to it before. This ensures that the message can only be accessed by the component which was intended to receive it. Security analysis on a style model could then warn anonymity weaknesses in face of strong security requirements. The designer could then have a look at other styles or investigate the introduction of security mechanisms after model transformation. In order to preserve architecture quality after refinement by introducing such mechanisms, cross-cutting concerns between extrafunctional properties need to be addressed. Extrafunctional properties influence each other [15] and can lead to conflicts in face of requirements (*e.g.* a security mechanism impacts performance issues).

On the other hand, extensions to process algebras allow to introduce a timed interpretation, *i.e.* to specify time performance properties as computing time or message transfer time. These existing works seem a good starting point to introduce extrafunctional properties into the framework. Their integration and the extension to other properties (*e.g.* scalability, dependability) are under investigation.

6 Related Work

An application can rely on several styles. As an early example, Garlan and Shaw [8] have defined a collection of architectural styles showing how different architectural solutions for a same problem offer different benefits. For example, they outline four distinct architectural designs for the *Key Word In Context* (KWIC) system (*i.e.* shared data, abstract data types, implicit invocation, and pipes and filters). However, their early proposal does not open up the way to a systematic transformation of functional models with different styles. In a distributed system, selected interaction mechanisms impact locally on extrafunctional properties of a point-to-point interaction. But the choice of the right architectural style also broadly depends on the emergent properties addressed by the overall structure. To guide the selection, a formal specification of common distributed styles is a prerequisite for early analysis. Moreover, styles could be combined [16] to meet specific requirements, encouraging analysis assistance. Our proposal tends to go one step further in this direction.

In the last ten years, a number of architecture description languages have been proposed to represent software architectures (*e.g.* [17,18,19]). More recently, development and deployment of large distributed systems also conduce to rely on component models with dedicated languages (*e.g.* CCM, Fractal, GCM). Some of those architecture description languages open up the way to analysis by

incorporating formal specifications (*e.g.* CSP or pi-calculus, Z, OCL, types, graph grammars or chemical abstract machine). However, their usage does not directly dissociate styles from system models and therefore limits the definition of a fixed repository of styles independent of the architect's know-how.

To the best of our knowledge, three mature frameworks provide some architecture stylistic guidance. Morisawa and Torii [20] have restricted their exploration to the client-server style alternatives and propose to evaluate them under some of the ISO 9126 quality issues (*e.g.* data security, reply to user for time performance). Metrics with maximum range are fixed on properties. The target style may be selected thanks to size and distance functions regarding requirement criteria. It is worth noting that several extrafunctional characteristics are not taken into account at the level of early design choices. By separating concerns, the ISO 42010:2007 recommended practice [21] now provides some elements within a conceptual framework for describing and analysing complex architectures in terms of architectural viewpoints. However, there is still a recognised gap between requirements and architectural description phases.

The NFR framework [22] considers extrafunctional goals to guide the designer and cover a far-reaching area of extrafunctional requirements. A nonfunctional requirement is defined in a tree description as a combination of lower level requirements (*e.g.* security is a combination of confidentiality, integrity and availability) and pattern-mechanisms to meet them (*e.g.* confidentiality can be ensured by using authentication and access matrix). Informal positive and negative contributions between mechanisms and requirements are elaborated by an expert to guide the selection. Architectural patterns can also be attached to a tree, based on property contributions known by experience. In this case, an architecture guidance is a mechanisms and patterns proposal in conformance with requirements.

After identifying actors and goals for a system, the i^* framework [23] permits to represent dependencies between components (*e.g.* tasks, resources) and then elaborates alternative architectures. By selecting predefined architectural patterns corresponding to quality attributes, refined solutions of an instance system are then evaluated through metrics (actor-based and dependency-based). This framework, used for system reengineering (*i.e.* SAR*i*M and PR*i*M methods) under quality issues, fruitfully distinguishes functional and extrafunctional aspects. However, its pattern-based approach is flexible but less rigorous than a formal transformation, and stylistic choices remain handmade at a pattern-level rather than at a high structural one.

7 Conclusion and Perspective

Designing software architecture of good quality, satisfying requirements, is recognised as a complex task. To facilitate construction in the lifecycle, well-known architectural abstractions could be employed at the early design stages. However, a style choice could considerably impact extrafunctional properties in the rest of the design cycle with sometimes large influences at the implementation stages. Requirement specifications do not always impose or promote style models.

Depending on the functional and extrafunctional properties, an architect faced with alternatives could rely on his/her know-how. Thus, alternatives scope and criteria for style selection often remain implicit [2] and tend to be put forward in the design process.

It is critical to better manage extrafunctional properties in the architecture design as an engineering discipline. Architecture analysis at the early stages is crucial to satisfy requirements in the final application or system and limits tacit choices for the architect. By specifying the behaviour of an application only in terms of functional concerns, independently of a style, our conjecture prepares for the separation of distribution concerns. Indeed, some extrafunctional properties [15] are intrinsic to specific styles [16] (*e.g.* reliability in client-server or scalability in peer-to-peer).

Therefore, we have proposed in this paper a model-driven framework to shed light on the appropriateness of separating functional system concerns from distributed architectural style. Relying on a process calculus, our formal design process enables different architectural solutions to be systematically generated by a transformation model. We tackled style independent and style specific models at an abstract specification level, and have shown through a classical distributed application that functional models could be expressed by a designer independently of interaction mechanisms. Based on an expandable repository of styles, a functional model of an application could be systematically composed with alternative styles for further comparative analysis before development. The applicability of our approach has been justified, through three distributed styles, with a classical distributed version control system case study. For the architect faced with design alternatives, our framework thus provides early formal support and allows to address the software architecture quality at higher design stages.

Following the NFR framework [22] proposal for quality attributes, future work will take into consideration other extrafunctional properties for styles and analyse their impact on the composition process. Is a property simply intrinsic or not to a style? How to match the extrafunctional requirements with the results of analysis in our approach? Noting that resulting models are analysable with respect to some extrafunctional properties, they could be extended with patterns of mechanisms to meet requirements. Finally, a process calculus is not ideally suited to all of the extrafunctional concerns. Further, it might restrict the expandableness of the style repository (*e.g.* in order to address more dynamic notions). Other formalisms could also be considered in our framework to meet requirement specifications and to increase the expressiveness of style description.

References

1. Bhattacharya, S., Perry, D.E.: Predicting architectural styles from component specifications. In: Proceedings of the 5th Working IEEE/IFIP Conf. on Software Architecture, pp. 231–232. IEEE Computer Society Press, Los Alamitos (2005)
2. Kruchten, P., Lago, P., van Vliet, H., Wolf, T.: Building up and exploiting architectural knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214. pp. 43–58. Springer, Heidelberg (2006)

3. Shaw, M.: Comparing architectural design styles. IEEE Software 12(6), 27–41 (1995)
4. Orfali, R., Harkey, D., Edwards, J.: The essential client/server survival guide. John Wiley and Sons, Chichester (1996)
5. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
6. Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Computing Surveys 36(4), 335–371 (2004)
7. OMG: Object Management Group (Lillerand, J., Mukerji, J. (eds.)) Model Driven Architecture Guide, version 1.0.1 (June 2003), http://www.omg.org/docs/omg/03-06-01.pdf
8. Garlan, D., Shaw, M.: An introduction to software architecture. Advances in Software Engineering and Knowledge Engineering 2, 1–39 (1993)
9. Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A system of patterns. John Wiley and Sons, Chichester (1996)
10. Shaw, M., Clements, P.: Toward boxology: preliminary classification of architectural styles. In: Proceedings of the second international software architecture workshop (ISAW-2) on SIGSOFT 1996 workshops, pp. 50–54. IEEE Computer Society Press, Los Alamitos (1996)
11. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)
12. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons, Chichester (2006)
13. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619. pp. 597–601. Springer, Heidelberg (2003)
14. Schneider, S.: Security properties and CSP. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 174–187. IEEE Computer Society Press, Los Alamitos (1996)
15. Chung, L., Nixon, B.A., Yu, E.: Using non-functional requirements to systematically select among alternatives in architectural design. In: First International Workshop on Architectures for Software Systems (IWASS), pp. 31–43 (1995)
16. Mehta, N., Medvidovic, N.: Composing architectural styles from architectural primitives. In: Proceedings of the 9th European Software Engineering Conference (ESEC), pp. 347–350. ACM press, New York (2003)
17. Magee, J., Kramer, J.: Modelling distributed software architectures. In: First International Workshop on Architectures for Software Systems (IWASS) (1995)
18. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
19. Zhang, S., Goddard, S.: xsadl: An architecture description language to specify component-based systems. In: Proceedings of the IEEE Int. Conference on Information Technology: Coding and Computing, pp. 443–448. IEEE Computer Society, Los Alamitos (2005)
20. Morisawa, Y., Torii, K.: An architectural style of product lines for distributed processing systems, and practical selection method. In: ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 11–20. ACM, New York (2001)

21. ISO: International Organization for Standardization: Systems and Software Engineering – Recommended practice for architectural description of software-intensive systems. ISO/IEC DIS 42010, 90.92 review stage (December 2007)
22. Chung, L., Gross, D., Yu, E.: Architectural design to meet stakeholder requirements. In: Donohue, P. (ed.) Software Architecture, First Working IFIP Conference on Software Architecture (WICSA1), Vienna, Austria, pp. 545–564. Kluwer Academic Publishers, Dordrecht (1999)
23. Grau, G., Franch, X.: A goal-oriented approach for the generation and evaluation of alternative architectures. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758. pp. 139–155. Springer, Heidelberg (2007)

Carmen: Software Component Model Checker

Aleš Plšek[1] and Jiří Adámek[2,3]

[1] INRIA-Lille, Nord Europe, Project ADAM
USTL-LIFL CNRS UMR 8022, France
ales.plsek@inria.fr
[2] Distributed Systems Research Group
Charles University in Prague
Czech Republic
adamek@dsrg.mff.cuni.cz
[3] Institute of Computer Science,
Academy of Sciences of the Czech Republic

Abstract. The challenge of model checking of isolated software components becomes more and more relevant with the boom of component-oriented technologies [20]. An important issue here is how to verify an open model representing an isolated software component (also referred as *the missing environment problem* in [17]).

In this paper, we propose on-the-fly simulation of the component environment to address the issue. We employ behavior protocols [18] and a system coordinating two model checkers: Java PathFinder [4] and BPChecker [15]. This approach allows us to enclose the model representing the behavior of a given component and consequently to exhaustively verify the model. Our solution was implemented as the Carmen tool [1].

We demonstrate scalability of our approach on real-life examples and show that, in comparison with the COMBAT model checker [17], we bring better performance, and also exhaustive and correct verification.

1 Introduction

Model checking [9], as one of the most popular approaches to formal verification of software systems, has already proven to be useful. However, the need for extracting a finite model from a target system (the "classical" model checking) forces researchers to seek approaches on model checking at the source-code level. Despite the complexities of these approaches, particularly *the state explosion problem*, there exist such model checkers (e.g. Java PathFinder [4] or Bandera [10]). One of the methods of coping with the issue of state explosion is decomposition of a system into small parts which can be verified separately.

Independently on this branch of research, widely popular Component-Oriented Programming [20] introduces software components – small compact units providing a certain functionality through strictly defined points. It is therefore natural to tackle the problem of software component verification, since components themselves bring the most straightforward way of decomposition – a property so intensively sought when fighting the state explosion problem.

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 71–85, 2008.
© Springer-Verlag Berlin Heidelberg 2008

In the scope of formal verification, we distinguish between closed and open systems. A closed system is autonomous, i.e. it does not communicate with another system. In the context of component programming, it is e.g. whole component application — there are no interfaces for the communication of the whole application with another component. On the other hand, an open system communicates with other entities; again, in the context of component programming, it is e.g. a single component, that communicates with other components (its environment) via interfaces. A behavior model of a closed system is called a closed model, while a behavior model of an open system is called an open model.

From the verification point of view, a behavior model specified by the code of a single component (an open model) is incomplete, as the behavior of the component depends not only on the decisions made by the component itself, but also on its environment. In the context of different environments, the behavior of the component can differ. However, the source-code level model checkers typically need a closed model as the input. Therefore, an important question arises here: How to enclose the model of a component and thus to allow formal verification? The challenge is also referred as *the missing environment problem* [17].

The goal of our research is to propose an answer to the question above. In this paper we design a method of on-the-fly simulation of software component's environment to achieve a closed model. In our solution, component's implementation and its behavior specification, given in a form of a behavior protocol [18], are processed by two cooperating model checkers - Java PathFinder [4] and BPChecker [15]. These cooperating tools then formally verify component's implementation against a behavior protocol and specified properties.

Our solution was implemented as the Carmen tool [1]. We compared Carmen with COMBAT — the tool presented in [17], addressing the same issue. We concluded that Carmen performs exhaustive verification, while COMBAT does not. Also, the state space traversed by Carmen is smaller, which is important for performance.

To reflect the goal, the structure of the paper is as follows. Section 2 introduces basic insights into Java PathFinder, Component-Oriented Programming, and Behavior Protocols. At the end of the section, we elaborate on the goal of our research. While Section 3 presents possible approaches to *the missing environment problem*, Section 4 describes in detail the concept we have chosen to implement. Section 5 demonstrates our contributions and scalability of the solution on real-life examples. In Section 6 we discuss related work. Section 7 concludes the paper.

2 Background

2.1 Java PathFinder

Java PathFinder (JPF) [4,21] is an explicit state software model checker. It verifies given program by traversing its state space and searching for implementation errors (e.g. deadlocks, unhandled exceptions,...) and property violations. Moreover, user's own properties can be defined. JPF operates at the program

byte-code level which means that a real-life application written in Java is used as a model of a system. A custom Java virtual machine (JPF VM) is used to execute a given program in every possible execution path. The state space of a target program is a directed acyclic graph in principle with branches determined by Java bytecode instructions, thread interleavings, and possible values of input data. JPF fights the state-space explosion problem by implementing POR algorithm [9] and state matching heuristics [12].

2.2 Component-Oriented Programming

We employ the basic idea of Component-Oriented Programming [20] that is further extended in the hierarchical component models, e.g. [3,5]. Here, components are either *composite* (created as a composition of lower-level components) or *primitive* (implemented directly in a common programming language, e.g. Java). Components are viewed as black-box entities. Interfaces of components can be either *required* or *provided*. Through provided interfaces, services of the component are accessible, the required interfaces are connected to other components to intermediate delegation of tasks. By the term *environment* we denote all the components connected to the interfaces of a given component.

We implemented the Carmen tool, introduced in this paper, for the Fractal component model [3]. As a future work, we also plan to adapt Carmen for the SOFA component model [5]. We chose Fractal and SOFA because we had been experienced with the formal verification of the applications written in those component models and because checkers of behavior protocols had been already implemented for both of them [6,15].

2.3 Behavior Protocols

Behavior protocols [18] are a language for component behavior specification. They have been successfully applied to the SOFA [18] and Fractal [6,15] component models. To analyze behavior of components specified via behavior protocols, Behavior Protocol Checker (BPChecker) [15] was developed.

A behavior protocol describes communication of a component with its environment. [1] On the semantic level, such a communication is defined as the set of all admissible sequences of *events* on the component's interfaces. There are two kinds of events: *requests* for method calls and *responses* to those requests.

Syntactically, behavior protocols are similar to process algebra [7]. The basic building blocks of a behavior protocol are *event tokens*, denoting the events. An event token has the following syntax: `<prefix><interface>.<method><suffix>`. The prefix `?` denotes acceptance of an event, the prefix `!` denotes emission of an

[1] To be precise, a behavior protocol can describe not only the communication of a component with its environment, but also the interplay of events inside a composite component. However, this alternative usage of behavior protocols is out of scope of this paper, as our goal is to check consistency of a primitive component code with the protocol of the component; specification of a composite component behavior via behavior protocols is not needed here. For more details, see [18].

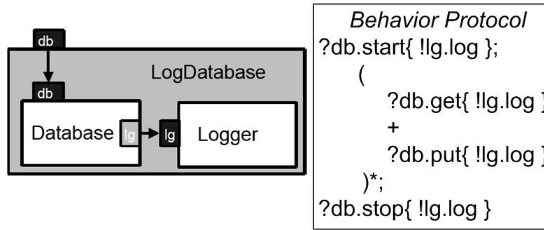


Fig. 1. Motivation Example: the *LogDatabase* composite component consisting of the *Database* and *Logger* subcomponents. The small black and gray boxes denote provided and required interfaces. E.g., `db` is a provided interface of *Database*, while `lg` is a required interface of *Database*. The behavior protocol of *Database* is shown.

event. The suffix ↑ denotes a request (i.e. a method call), and the suffix ↓ denotes a response (i.e. return from a method). Therefore, for `i` being an interface name and `m` being a method name on `i`, `?i.m`↑ stands for accepting the request for a call of `i.m`, while `!i.m`↓ denotes the emission of the response for a call of `i.m`.

Behavior protocols are syntactically constructed from the event tokens using *operators*. There are operators for sequencing (*;*), alternative behavior (+), repetition (∗), and arbitrary interleaving (|), that is useful for behavior specification of parallel processes.

Also, abbreviations are defined for behavior protocols; they serve as syntactic sugar, standing for complex but often used constructs. The abbreviation `?i.m` stands for `?i.m`↑ `;` `!i.m`↓, i.e. acceptance of a request followed by the emission of the associated response (i.e. the typical part of the behavior of a component providing `i.m` to the outside world). Similarly, `!i.m` stands for `!i.m`↑ `;` `?i.m`↓. Finally, if `P` is an arbitrary protocol, `?i.m`{`P`} stands for `?i.m`↑ `;` `P` `;` `!i.m`↓, i.e. it describes a part of the behavior of a component providing `i.m`, where the protocol `P` describes what the component does inside of the implementation of `i.m`.

`NULL` stands for an empty protocol (specifying no behavior).

We demonstrate the usage of behavior protocols on a simple example shown in Fig. 1. Here, the functionality of the *Database* component is expressed by its behavior protocol. First, *Database* accepts the initialization call — `db.start`; this leads to calling `lg.log` and then the result of the `db.start` call is returned. After that, *Database* is able to absorb an arbitrary number of `db.get` or `db.put` calls, each resulting in an `lg.log` call. To finish the execution, the component is stopped by calling `db.stop`.

2.4 Goal Revisited

Behavior protocols give to a component application developer the option to check consistency of his or her design from the point of view of component behavior. For example, if behavior protocols of all three components in Fig. 1 are provided by the developer, it is possible to check correctness of communication between

Database and *Logger*, as well as compliance of the *LogDatabase* internals behavior (determined by the protocols of *Database* and *Logger*) with the *LogDatabase* protocol itself [18]. However, once we take also primitive components into consideration (i.e. the components that are not composed of subcomponents, but directly implemented in some programing language instead — and there must be such components in each application), things get more complicated.

Let as assume that *Database* is primitive (and is implemented in Java). Now, we cannot assure the correctness of the *Database* implementation by pure behavior protocol analysis, as there are no behavior protocols describing the behavior of *Database* internals.

Moreover, we can look at the problem from another point of view: we want to use verification tools for Java code. One of the options to specify the properties to verify is to use *assertions* — conditions that must by true when the control reaches given places in the code[2]. However, as the code of the component is an open code (it has no predefined entry point and the behavior of the code depends on how the environment will use it), it is not possible to use the code verification tool to check the properties expressed as assertions. As mentioned in the introduction, this issue is called missing environment problem [17].

Therefore, the goal of our work is the following: to design and implement a tool that (1) checks the compliance of Java implementation of a primitive component with the behavior protocol of the component (i.e. verifies that the code does what the protocol specifies), and (2) at the same time it checks validity of the assertions in the Java code; only those runs that correspond to the behavior specified via the protocol are taken into consideration.

We chose Java as both the SOFA and Fractal component models (where the behavior protocols were already applied) use Java as the implementation language.

3 Cooperation of Model Checkers

The problem we tackle in this paper is a verification problem. To solve it, we decided rather than developing a brand new tool to adapt an existing model checker. From our study, the Java Path-Finder tool (JPF) emerged as the best option. It provides wide functionality and can be easily modified and extended.

However, JPF itself does not allow to cope with all the issues of a single component verification. Since JPF allows to verify only closed models, we introduce behavior protocols to substitute the environment of the component and thus to enclose the model. During the verification it is then necessary to observe the communication of the component with the environment represented by behavior protocols. To do this, we employ an additional model checking tool – BPChecker. The specific details of such a checker cooperation form our main contribution.

The task of the cooperation is to synchronize the verification of the component implementation, performed by JPF, and the verification of component

[2] Contrary to the classic assertions used for software testing, assertions in formal verification are much more powerful tool, as the verification tool checks the validity of assertions for *all* possible runs.

external behavior, performed by BPChecker, whenever a communication between the component and its environment occurs. Such a synchronization can be achieved using two different concepts discussed further: Virtual Environment or Environment Simulation.

3.1 Virtual Environment Concept

The key idea of this concept, presented in Fig. 2 A), is to automatically generate virtual environment of a component (i.e. a Java code), creating a closed system that can be verified by JPF. Such a code has to provide an entry point (the `main` method). Moreover, the virtual environment has to be able to perform every sequence of events described in the component's behavior protocol. This guarantees that JPF will be able to analyze all the behavior alternatives that are relevant.

While verification of such an enclosed model (code of the component + virtual environment) is simple (this can be done with just a minor modification of JPF [17]), generating a virtual environment from the protocol has many issues. The reason is that some forms of behavior protocols (e.g. those specified using the alternative and repetition operators) can not be equivalently expressed by a Java code. Therefore, no virtual environment can correspond to such protocols, and consequently the verification process cannot be correct. Despite its disadvantages, this concept was implemented in the COMBAT model checking tool [17].

3.2 Environment Simulation Concept

The idea of the Environment Simulation concept is to use JPF to analyze only the code of the verified component itself and to handle the events on the external interfaces of the component via a modification of JPF — see Fig. 2 B). Every time the *Manager* detects communication initiated by the verified component, it interrupts the verification process and let the *Response Generator* simulate appropriate environment responses according to the behavior protocol of the verified component. The information about the appropriate environment responses

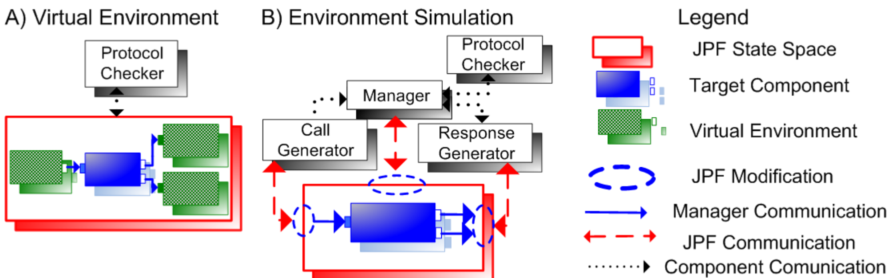


Fig. 2. JPF and BPChecker Cooperation, Proposed Concepts

is taken from the *Protocol Checker*, that is run in a special mode. At the same time, *Call Generator* is used to simulate the calls initiated by the environment. Finally, the *Protocol Checker* is used not only to obtain the information about the environment responses, but also to check that the events emitted by the verified component respect the protocol.

The Environment Simulation concept allows to simulate any form of behavior protocols, including the alternative and repetition operators, providing correct and exhaustive form of verification. Moreover, as *Manager* can interrupt the verification process at any time and force JPF to explore another execution path, it is possible to control verification and to smoothly integrate additional heuristics.

In the light of the outlined options, the Environment Simulation concept was chosen to implement. Based on this decision, the Carmen project [1] was founded. More extensive description of the project can be found also in [19].

4 Environment Simulation

Based on the discussion above, we propose to develop a Software Component Model Checker, which implements the Environment Simulation concept. To facilitate the verification, we have to simulate a component environment by generating events that will be absorbed by the component. The component is then forced by JPF to respond to these artificially created events, its behavior is evaluated and thus the component is being verified.

4.1 Cooperation

Since each tool operates at a different level of abstraction - JPF with byte-code instructions and BPChecker with events, we need to define a proper mapping between their state spaces to achieve cooperation. This would be possible if states that represent absorbed events could be identified. Whereas this is inherently satisfied inside the BPChecker state space, the JPF state space represents only the component itself. To tackle this problem, we have extended the JPF state space with states that represent communication between the component and its

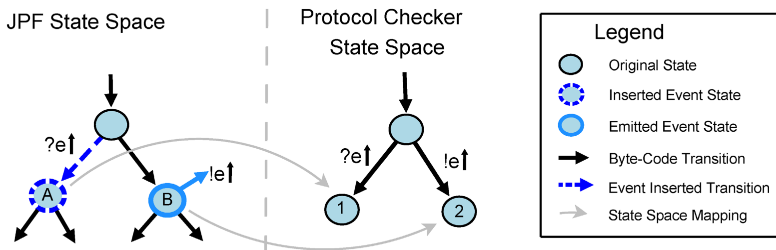


Fig. 3. State Space Mapping

environment. Therefore we are able to find a mapping between state spaces of the checkers. See Fig. 3 for an illustration example.

4.2 Environment Simulation

The environment simulation process generates events that occur on interfaces of the component. These events have to be then inserted into the JPF, afterwards the verification can continue. We are able to determine which event has to be inserted in cooperation with BPChecker. Consequently, state space extensions allow to simulate an absorbed event by creating a new state and by inserting it into the JPF. Verification then continues from a newly inserted state and thus, the component is forced to react to the new event.

Moreover, by employing the backtracking strategy we are able to simulate every possible sequence of events. The component is therefore verified against every behavior of its environment that is in conformance with a behavior protocol of this component.

An absorbed event is however representing also a data which are being transformed from an environment to the component and these data have to be generated as well. We refer to this in Section 5.2.

4.3 Verification

The central unit of the verification process is *Manager*. It communicates with both the checkers, arbitrates the cooperation between JPF and BPChecker and determines future steps of the verification. Manager evaluates states of the checkers and decides which events will be simulated on interfaces of the component. Figuratively speaking, JPF represents the component, BPChecker represents its environment and Manager provides a connecting layer between them.

To better illustrate the role of Manager, we introduce code snippets of methods which are used by Manager to control the progress of the verification. The method `stateAdvanced()` listed in Fig. 4 handles a situation when JPF advanced a new state. First, Manager verifies if there was any emitted event and whether it was in compliance with a given behavior protocol (line 2-3). Consequently, Manager tries to simulate a next event, if BPChecker proposes any event, it is simulated, both tools are notified and we proceed to a new state (lines 5-8). If there is no event to simulate, Manager only verifies that both tools are in accepting states in case the end of an execution path was reached.

The method `stateBacktracked()`, listed in Fig. 4 (line 15), handles situations when JPF backtracked from an already explored state. The task of Manager is to backtrack also a simulated event and then to simulate a new one (lines 20-22). If there is no event to simulate, nothing is to be done since all paths starting by events were already explored.

Thanks to these notification methods Manager is able to coordinate cooperation of both checkers and thus to achieve an exhaustive verification of all execution paths of the component implementation.

```
void stateAdvanced() {
2  if eventEmitted()
     BPChecker.verifyEvent(emittedEvent);
4  newEvent = BPChecker.getEvent();
   if newEvent != null
6    JPF.simulateEvent(newEvent);
     BPChecker.eventSimulated(newEvent);
8    stateAdvanced();
   else
10   if JPF.isEndofExecutionPath() && BPChecker.isNotAccepting()
           reportErrorBehavior();
12 }

14 void stateBacktracked() {
   if isEventToBacktrack()
16   BPChecker.backtrackEvent(event);
     newEvent = BPChecker.getEvent();
18   if newEvent != null
         JPF.simulateEvent(newEvent);
20       BPChecker.eventSimulated(newEvent);
         stateAdvanced();
22 }
```

Fig. 4. Manager Arbitrating an Advanced/Backtracked State

4.4 Motivation Example Revisited

In this section we revisit the motivation example from Section 2.3 to demonstrate the verification process. The Fig. 5 shows the implementation of the `Database` component together with its behavior protocol. The arrows are showing the correspondences between events of the behavior protocol and method calls inside the component implementation code.

From JPF point of view, every event absorbed by a component is represented inside the JPF VM as a thread which invokes a given method on a particular interface. On the other hand, an emitted event is represented as a thread invoking a method on an interface of another component.

Considering our example, both checkers are in their initial states at the beginning, there are no threads in JPF VM. Manager therefore asks BPChecker for a list of events which can be simulated. According to the protocol, an event `?db.start` is proposed. This event is then simulated, a new thread which invokes the method `start` on the interface `db` is created inside JPF. From now JPF starts with the verification of the component's code. This process is monitored by Manager and interrupted whenever the component tries to communicate with its environment. Here, such situation occurs when the component invokes `lg.log`. Manager immediately stops the verification and verifies that the emitted

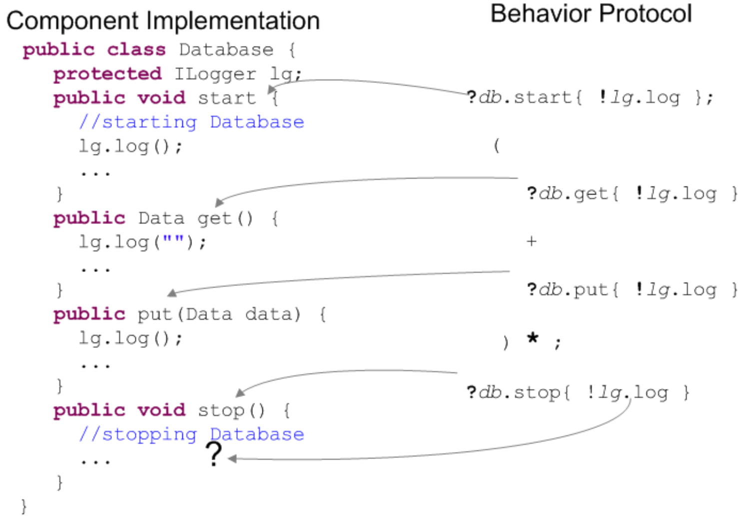


Fig. 5. Example of the Verification

event conforms to a given behavior protocol. Then the thread is interrupted until the moment when BPChecker proposes a simulation of an event which represents a response to the invoked call. In between, the verification of parallel threads inside the JPF state space can continue.

Looking at the behavior protocol in Fig. 5, we can see that an event `lg.log` does not have any corresponding method call in the component implementation, in Fig. 5 indicated by a question mark. During the verification, JPF executes the method `stop`, reaches its end and notifies BPChecker. However, the the protocol specifies that during the `stop` method execution, an even `!lg.log` will be emitted. Since no such event occurred, it is an obvious behavior protocol violation and an extensive report (including stack traces of both checkers) will be send to the developer.

Except the component's behavior, which is verified whenever the component emits an event, the JPF checker itself verifies additional properties, e.g. the presence of deadlocks, unhandled exceptions or any other user defined properties. This finally leads to an exhaustive verification of every property along all the execution paths of the component's implementation.

5 Evaluation

As the biggest contribution of our work we consider the Environment Simulation concept that straightforwardly solves *the missing environment problem*. Contrary to the COMBAT checker [17] described in Section 6, we do not require any reductions of behavior protocols; therefore, our approach provides an exhaustive simulation of the environment and correct verification of components.

To show the quality of our method, we developed a prototype implementation – Carmen Project [1]. The tool verifies Fractal software components [3] implemented in Java against their behavior protocols and the sets of user-defined properties. In this section we present the performance evaluation and discuss the limitations of our tool.

5.1 Case Studies and Performance Evaluation

For performance evaluation, we used real-life case studies from the Component Reliability Extensions for Fractal component model (CRE) [6] and CoCoMe [2] projects.

CRE is an application that manages the airport services for wireless internet connection. It consists of more then twenty components. We have selected three non-trivial components for verification[3]: the *FlyTicketClassfier* component classifies air tickets and provides connections to the appropriate database, the *ValidityChecker* component verifies the airtickets, and the *Arbitrator* component controls the whole system.

For the second part of the performance evaluation, we used the Fractal implementation of the *Store* and *CashDeskExample* components from the CoCoMe case study [8], addressing the simulation of cash desk system in a supermarket.

For the performance evaluation, we did several comparison tests between Carmen and COMBAT, using the code of the components mentioned above. The following parameters have been monitored: *Unique States* (number of unique states that were reached), *Visited States* (total number of reached states), *Time* (total time of the verification), and *States/Second* (the number of states visited per second).

The results of the performance evaluation[4] are presented in Table 1.

While COMBAT verifies a closed system (including the generated environment), Carmen simulates the environment during the verification and therefore the progress of the verification is slower. This can be observed when verifying the components from the CRE case study — *FlyTicketClassifier*, *ValidityChecker*, and *Arbitrator*. However, the state space of COMBAT is larger, which is caused by the necessity to include the generated environment. Thus, the total verification time is better for Carmen in all the three cases and the difference between the total verification times (Carmen vs. COMBAT) is the bigger the larger the state space is.

As to the verification results for the CoCoMe case study (*CashDeskApp*, *Store*), Carmen again generates considerably smaller state spaces and the verification times are reasonable. However, the COMBAT tool achieves better total verification time. We believe that this is caused by the recent progress of the COMBAT tool which was ported to a newer version of JPF (version 4), whereas

[3] More detail information regarding these components, the whole case study, and the Carmen documentation can be found at the project web page [1].

[4] All the tests were run on Pentium 4 3.0 GHz with 2.0 GB RAM, Windows Server 2003 OS.

Table 1. Performance Comparisons

Case Study	Component Name	Checker	# States Unique	# States Visited	Time	States/ Second
CRE	FlyTicketClassifier	Carmen	922	1 920	3s	640
		COMBAT	6 519	10 254	4s	2563
CRE	ValidityChecker	Carmen	435	592	2s	296
		COMBAT	4 033	9 324	4s	2331
CRE	Arbitrator	Carmen	6 074	14 898	34s	438
		COMBAT	166 977	378 437	9m:30s	663
CoCoMe	CashDeskApp	Carmen	3 480 851	6 644 606	1h:32m:17s	1200
		COMBAT	4 839 108	10 541 046	33m:26s	5 254
CoCoMe	Store	Carmen	574 538	1 717 282	2h:29m:09s	192
		COMBAT	11 669 994	28 728 733	1h:49m:08s	4 387

Carmen uses an old one (version 3.3.1). We reflect this finding in our future work (Sect. 7).

The bottom line is that Carmen is able to verify complex components in a reasonable time without any reductions of behavior protocols. The confrontation with COMBAT, which requires additional reductions of behavior protocols, has revealed that Carmen reaches fully correct verification and comparable performance.

5.2 Tool Limitations

Even though our approach potentially achieves exhaustive verification, the real-life application brings several limitations. Specification of parameters that are passed to the methods when generating events is the most important burden to deal with. The range of possible values has to be manually specified and its extensiveness directly affects the state space size. Therefore, the values should be chosen with respect to the component implementation, to allow the checker to explore maximum of execution paths. The details are out of scope of this paper, we briefly discuss some of them in Sect. 6.

More detailed evaluation of Carmen and a discussion of its limitations can be found in [19].

6 Related Work

COMBAT [17] uses, similarly to the approach applied in our work, JPF in co-operation with BPChecker. It generates a virtual environment that is verified together with the component (see Section 3.1). For more information about the environment generation see [16]. However, the significant disadvantage of this checker lies in the absence of any solution to repetition and alternative operator problems addressed by Carmen. Instead, behavior protocols are simplified in

order to avoid unsupported forms of protocols. These constraints consequently lead to a non-exhaustive verification of components. Nevertheless, we demonstrate the performance comparisons between both the approaches in Section 5.

Also, our approach is related to the assume-guarantee principle in model checking [13]. The tools based on this principle report the description of all the environments in which a given model satisfies a given property. We also use the idea of environment, but in the opposite manner: the description of the environment behavior (the calls from the environment to the component described in the behavior protocol) is given by the developer and the tool checks whether the property is satisfied in the environment. Note that the property itself is also specified by the behavior protocol (the reaction of the component to the calls made by the environment).

When searching for an equivalent alternative to Java PathFinder [4], we have been considering an alternative — Bandera [10]. It is a set of tools and modules which are designed to verify Java programs. Bandera accepts a complete Java program as an input and translates it into a language that can be verified by a specified model checker. Although Bandera is not intended to verify software components, it decomposes a target program into a part which is verified and the rest that is represented by specially generated environment. This approach is very similar to the Environment Generation concept presented in Section 3.1. Bandera also allows to use value domains for specifications of method parameters of given classes. However, the recent release of Bandera is an alpha version which is not fully stable yet.

Finally, we chosen Java PathFinder [4] as the basis of our implementation since it allows modification of its core implementation and is designed to support extendability by additional plugins.

In our work, we mainly focus on the verification of the order in which the methods of the component are called; another big issue is to cope with the values of the parameters that are passed to the methods. We use very simple heuristic approach to solve this problem, more sophisticated methods can be found e.g. in [11] or [14]: under-constrained execution [11] is a special kind of symbolic execution, where some of the symbolic values (e.g. those that origin from the parameter values) are marked as under-constrained. If an error involves an under-constrained operand, an error message is produced only if the error occurs for all possible values of the operand (according to its type). This approach reduces the number of spurious errors. In [14], symbolic execution with lazy initialization is used to adapt Java PathFinder for verification of open systems: the method parameters are initialized during the execution in a lazy way; the exact value domains are not required from the developer.

7 Conclusion

In this paper, we present our approach to model checking of software components. Our solution verifies software components implemented in the Java

language against their behavior specifications (behavior protocols [18]) and sets of user-defined properties. To achieve the goal we designed a system that coordinates two model checking tools: Java PathFinder [4] and BPChecker [15]. Our solution was implemented as the Carmen tool [1].

Carmen employs on-the-fly simulation of software component environment to enclose the model representing implementation of an isolated software component. We consider this feature as the biggest contribution of our work.

Scalability of our approach was tested on real-life examples and the results show that our solution provides reasonable performance and brings fully correct verification.

As a future work we plan to improve performance of our tool by porting it to the most recent version of Java PathFinder (and thus to fully use its state-of-the art verification heuristics).

Acknowledgments

Special thanks go to the Distributed Systems Research Group, in particular to Jan Kofroň and Pavel Parizek, for helping with BPChecker integration and for assistance during performance testing.

This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266, by the ANR/RNTL project Flex-eWare and by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

References

1. Carmen Project (2008), http://www.lifl.fr/~plsek/projects/carmen/
2. CoCoMe Project (2008), http://agrausch.informatik.uni-kl.de/CoCoME
3. Fractal Project (2008), http://fractal.ow2.org/
4. Java PathFinder Model Checker (2008), http://javapathfinder.sourceforge.net/
5. SOFA Project (2008), http://sofa.objectweb.org/
6. Adamek, J., Bures, T., Jezek, P., Kofron, J., Mencl, V., Parizek, P., Plasil, F.: Component Reliability Extensions for Fractal Component Model (2008), http://kraken.cs.cas.cz/ft/public/public_index.phtml
7. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra. Elsevier, Amsterdam (2001)
8. Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Rivierre, N., Sery, O., Tuma, P.: CoCoME in Fractal. In: Proceedings of the CoCoME project (June 2007)
9. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
10. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Laubach, R.S., Zheng, H.: Bandera: Extracting Finite-state Models from Java Source Code. In: Proc. of the 22nd International Conference on Software Engineering (June 2000)
11. Engler, D., Dunbar, D.: Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In: International Symposium on Software Testing and Analysis (ISSTA) (2007)

12. Groce, A., Visser, W.: Heuristics for Model Checking Java Programs. Int. Journal on Software Tools for Technology Transfer (STTT) 6(4)
13. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component Verification with Automatically Generated Assumptions. Journal of Automated Software Engineering 12(3) (July 2005)
14. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619. Springer, Heidelberg (2003)
15. Mach, M., Plasil, F., Kofron, J.: Behavior Protocol Verification: Fighting State Explosion. Published in the Int. Journal of Computer and Inf. Science 6(1), 22–30 (2005)
16. Parizek, P., Plasil, F.: Specification and Generation of Environment for Model Checking of Software components. In: Proc. of Formal Foundations of Embedded Software and Component-Based Software Architectures, vol. 176(2) (May 2007)
17. Parizek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In: Proceedings of 30th IEEE/ NASA Software Engineering Workshop (SEW-30) (January 2007)
18. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering 28(11) (November 2002)
19. Plsek, A.: Extending Java PathFinder with Behavior Protocols. Master Thesis (2006), http://www.lifl.fr/ plsek/projects/carmen/download/documents/masterThesis.pdf
20. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Professional, Boston (2002)
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal 10(2) (2003)

MOSES: MOdeling Software and platform architEcture in UML 2 for Simulation-based performance analysis

Vittorio Cortellessa[1], Pierluigi Pierini[2], Romina Spalazzese[1], and Alessio Vianale[3]

[1] Dipartimento di Informatica
Università dell'Aquila
Via Vetoio, 67010 Coppito (AQ), Italy
{cortelle,romina.spalazzese}@di.univaq.it
[2] TechnoLabs S.p.A.
S.S. 17 Località Boschetto
67100 L'Aquila (AQ), Italy
pierluigi.pierini@technolabs.it
[3] Accenture S.p.A.
Largo Donegani 2,
20121 Milano, Italy
alessio.vianale@accenture.com

Abstract. Performance analysis at the architectural level has been a widely studied topic in the last few years. Automated solutions to this problem, such as the ones based on model transformations, would allow early detection of performance critical aspects in the software lifecycle. In this paper, building on top of our existing methodology [11] that aims at integrating software architectural models and platform models in the same notation (UML-RT), we present a new implementation based on the UML 2 metamodel that we call MOSES (MOdeling Software and platform architEcture in UML 2 for Simulation-based performance analysis). The goal of this paper is to provide a proof of concept that the UML 2 metamodel is rich enough to implement our approach that aims at modeling software and platform architecture within the same environment for sake of performance analysis. Finally we compare the results that we obtain with MOSES to the ones that we have obtained with the UML-RT implementation.

Keywords: Software Performance, Resource Modeling, UML, Simulation.

1 Introduction

The performance analysis of software architectures is a crucial issue in the wider domain of architecture verification and validation. In fact, as an emerging property, performance issues easily enter into the software lifecycle too late to be fixed, thus incurring, at the best, in expensive rework on the whole software project and, at worst, in a complete project failure.

In the last few years, several approaches (mostly based on model transformations) have been introduced to tackle this challenge. We have recently worked on a different approach, which basically allows the designers to not change notation only for sake of architectural validation. We intend to provide means to integrate a software architectural

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 86–102, 2008.
© Springer-Verlag Berlin Heidelberg 2008

model with a platform architectural model using the same notation, and to add in the same model all the necessary annotations for performance validation. Where successful, this approach allows to estimate the performance of a software architecture on multiple platform architectures without underlying (possibly incorrect) model transformations.

In this paper we build on top of a general methodology that we have implemented in UML-RT [9,10,11] with the aim to migrate the implementation in UML 2. We call this new implementation MOSES (MOdeling Software and platform architEcture in UML 2 for Simluation-based performance analysis). Our aim is to provide in UML 2 a performance validation methodology at the architectural level that is ”transparent” and easy to use for the software designers and also easy to integrate in the software development life cycle. The general methodology is independent of the modeling notation and the development environment adopted by the designers to model the software architecture. However, we provide here the proof that the UML 2 metamodel is rich enough to represent software and platform architectural models as a new implementation of our performance analysis methodology. By ”rich enough” we mean that UML 2 embeds all the notation elements that are needed to implement our approach. This property, however, it does not necessarily hold for other types of approaches.

From software developers point of view the idea of integrating software and platform architectural models for performance validation is only acceptable if the integration does not bring changes to the software architecture and their development practices. In other words, the ”transparency” is a key factor for such an approach and it is one of the major achievements of our methodology in that it only adds new elements to the software architecture to represent: (i) the platform architecture, (ii) the requests of resources that software components make to the platform.

A significant effort has been spent to select the appropriate supporting UML 2 CASE tool to implement the methodology. Being our approach based on simulation, we needed to consider CASE tools that, in addition to the UML 2 standard notation, support the model simulation and allow the designers to work in a single, stable environment.

The first implementation of our approach was supported by the IBM Rational Rose Real Time (RRT) tool, which is based on the UML Real Time (UML-RT) notation [23]. RRT has been chosen because of: (i) its market predominance; (ii) the availability of an integrated simulation environment; (iii) the presence of a powerful scripting language that allowed us to complete our framework with a set of supporting procedures.With the aim to achieve the maximum level of ”transparency”, we have created a set of scripts that allows the designers to insert and/or remove: (i) service provisioning points in the software model; (ii) the related connection to the service access point in the platform model; (iii) the code required to formulate the resource requests from software to platform. Furthermore, a number of industrial case studies were specified and simulated with the RRT support with satisfactory results [11].

In this paper we present MOSES, the new implementation of our methodology based on the UML 2 metamodel [4] which allows the modeling of software and platform architecture for simulation-based performance analysis within the same modeling environment. This new implementation is based on a mapping between UML-RT stereotypes and UML 2 metaclasses and can work within any UML 2 tool with simulation capabilities. This is due to the MOSES capability of working within the UML 2 standard,

without needing, like UML-RT, any UML profile definition. In order to validate this porting step, we have compared the results obtained with MOSES on a case study with the ones obtained with the original implementation in UML-RT on the same case study.

The remainder of the paper is organized as follows: in Section 2 we review relevant related work, Section 3 briefly resumes the basic concepts of our methodology, in Section 4 we describe the available tool support for modeling and simulation, pointing out the relevant differences between different tools, in Section 5 we describe the main characteristics of MOSES and we present the results obtained on a case study, and finally in Section 6 conclusions are provided.

2 Related Work

A large number of approaches concerning modeling and simulation of software and platform architectures for sake of performance analysis have appeared in literature.

Some of these approaches deal with models that are not described in UML and we provide two relevant examples. The work in [25] which is based on resource data coming from Resource Function Management Utility (RFMU) [26] and on the ObjecTime environment that supports the execution of a design model to produce software execution traces allowing the automated generation of a performance model based on Layered Queuing Networks [20]. A new component metamodel has instead been introduced in [19] to represent component-based software systems and to study different properties such as their performance [21].

In other approaches, although models are represented in UML they are not translated into simulation models, but they are transformed in different performance notations, like Queuing Networks, Petri Nets and Process Algebras [7]. Special cases of these approaches are the works in [13,18]. Both translate a software model in an intermediate model that is in turn transformed in a performance model.

A direct comparison with all these approaches is not appropriate here because they all are based on model transformation whereas we integrate platform models and annotations into software models. In other words we integrate in one notation software and platform models plus annotations, thus building a performance model in the same notation [11].

With respect to the validation of UML models based on simulation, we do not consider for comparison approaches like [15,6,17] in which simulations provide only a mean to observe the system dynamics, without collecting performance indices.

Instead the approach in [8] exploits Use Case and Deployment Diagrams performance annotations and translate them into simulation code whose execution produces performance indices that can be annotated back on the original diagrams. In [27] the capability of introducing additional code on states and transitions of UML-RT state diagrams has been exploited to build a framework for verification of timing constraints in real-time systems. In [22] the MASCOT design language is integrated with time and resource representations to build simulation models for system performance analysis.

After the standardization of UML 2 several new approaches and engines have been proposed. In [14] UML 2 diagrams are used for developing and evaluating network protocol simulation models in an event-driven simulation framework called OMNeT++.

The SYNTONY framework translates the XMI format of the UML 2 annotated models into executable network simulation models in C++ (the input format of OMNeT++).

In [12] the proSPEX methodology and tool are described for the design and performance analysis of communication protocols specified with UML 2 extended with ITU Z.109 profile. proSPEX exploits some features of Tau G2 tool (i.e., modeling, design checking, XMI exporting) and allows the translation of some UML 2 models into SDL specifications. Moreover the methodology allows the analysis of models with process-oriented simulation.

With regard to simulation engines, IBM has proposed a generic model execution engine, as reported in [16]. The engine has been developed for the construction, execution, control and observation of model behavior, and it is used to implement a UML Model Simulator as extension to Rational Software Architect (RSA) with execution and debugging capabilities. In this direction we still expect a new IBM Rational tool offering design and simulation facilities as it is in Rose RealTime [2], Telelogic TAU G2 tool [3] and ARTISAN Studio tool [1].

Our approach is conceptually independent of the specific simulation engine adopted. It has been conceived to be general enough to model architectures in different application and environmental domains. As an example, although the work in [22] may seem similar to ours, our approach is more general because our implementations have been built in the standard and widely adopted UML-RT profile and in UML 2 notations. Moreover our approach, basing on simulation, has the advantage to allow to the system designers to handle problems at different levels of abstraction to get results that, in some cases, are not experimentally measurable with the current level of technology. At the same time the approach presents the typical disadvantage of simulation that is an intensive computation that can imply long execution times to get reliable results.

3 The General Methodology

The general methodology basically proceeds through the following steps [9,10,11][1]:

1. Separately build a software architectural model and a platform architectural model.
2. Merge software and platform model to obtain an integrated architectural model.
3. Annotate the integrated model with data related to performance.
4. Simulate the annotated model to obtain the indices of interest.

The methodology was previously implemented using the UML-RT notation [2] that is an UML 1.x profile for real-time systems. The whole methodology implementation, therefore, was constrained to be run on simulation tools able to accept UML-RT models as inputs. In this paper we present the porting step that we have accomplished to represent basic elements of UML-RT in UML 2. This step allows to run the MOSES over any UML 2 simulation tool, thus widening by far its scope.

We focus here on the first step of the methodology, because it represents the step where basic elements of the modeling notation are manipulated. All the following steps

[1] For sake of space, we do not provide many details about our UML-RT implementation, however interested readers can refer to the cited papers.

remain the same, once initial models have been built, because they are not specifically bounded to the adopted modeling notation.

In particular, we have worked to find a suitable mapping between UML-RT basic elements and UML 2 metamodel elements. The challenge that we faced was to find a reasonable mapping that can avoid modelers to adopt any specific UML profile and, instead, allows them to work within the standard UML 2.

The resource model (platform architectural model) built in the first step is based on the general definition of platform proposed in [24] that describes it as a processing system partitioned into processing nodes. Each processing node consists of physical resources and a system environment (e.g., a PC or a workstation) offering a set of services to the hosted software applications whose design depend on the type of application. Thus a platform can be modeled by processing nodes each one embedding a set of private (local) resource instances plus additional supporting components. The platform can be represented only by the resources that play a critical role in the performance analysis and not necessarily by the complete processing environment.

Depending on the platform characteristics, one or more processing nodes have to be modeled in the platform model of the integrated architecture.

In figure 1 we show an integrated model whose platform is made of only one processing node. Each processing node has a three-layers structure. The bottommost layer contains the instances of the local resources (the Round-Robin CPU of 1. The uppermost layer contains the *Main Dispatcher* component that provides a single access point to send resource requests from software side to a node (in this case to the only one that is present). All the software components hosted on the same site send to the same *Main Dispatcher* their resource requests. We assume that each request is originated from a software action that may be performance critical, and it is encoded as a vector made of elementary demands. Each elementary demand represents the amount of a resource category that the action needs to be executed.

The *Main Dispatcher*, following its own strategy, dispatches each elementary demand to the *Internal Dispatcher* that manages the corresponding category of resources in the intermediate layer. Every *Internal Dispatcher*, in turn, following its own strategy, forwards each elementary demand to one of the resource instances that it manages (in the example in figure 1 it manages only one RoundRobinCpu). Thereafter a “demand satisfied” message is replied back from the resource, through the *Internal Dispatcher*, to the *Main Dispatcher*. Once all the elementary demands making up a resource request have been satisfied, the *Main Dispatcher* updates counters and data required for performance evaluation on a per request basis.

The statistical data that can be collected per request are usually related to the total execution time of the request itself. Thus, assuming that a request is made of several elementary requests, the total execution time is the time for completing the processing of all the elementary requests.

The previous implementation of this methodology, was based on a resource model based on a library of prototypes that we have built in UML-RT and that we called *PAlib*. It is a repository of Capsules and Statecharts, where each pair <Capsule, Statechart> represents a (static and dynamic model of a) specific type of platform element, such as a Round-Robin CPU. These building blocks can be easily used to assemble a platform

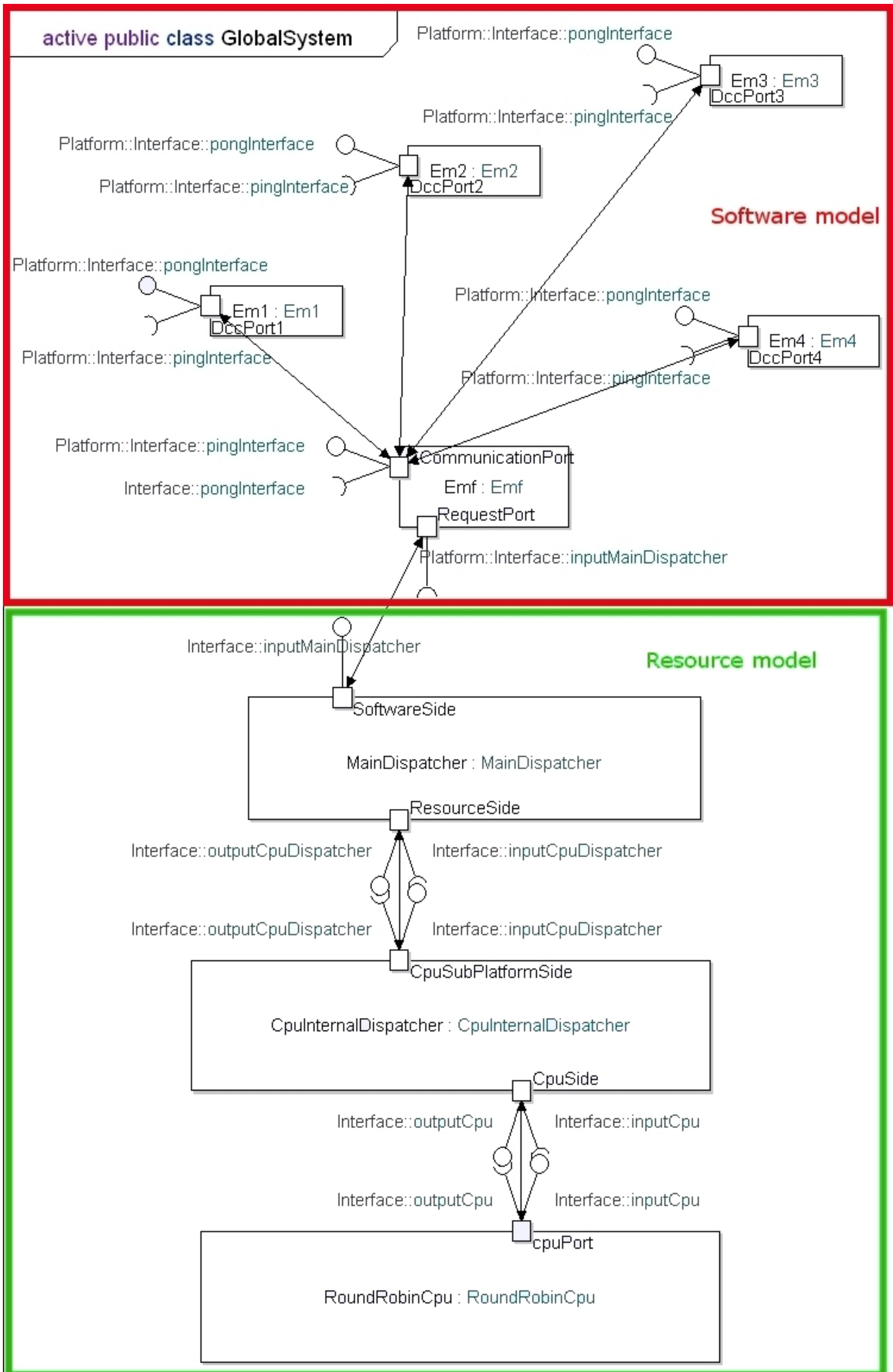


Fig. 1. An integrated architectural model

architectural model by instantiating, configuring, and connecting them together. The obtained platform model will be then integrated with a software architectural model.

More specifically, the *PAlib* can be partitioned in the following types of elements:

1. **Resource prototypes**, that model static and dynamic characteristics of physical resources such as CPUs, mass memories, network connections;
2. **Dispatching components**, that models middleware components that offer uniform platform interfaces to the software model;
3. **Utilities** required to model probes, service request queues, etc;

Besides, we provided a set of **procedures and rules** to support the construction of a platform architectural model, to integrate it with the software architectural model and to define resource request points, that are the critical points of the software model where a resource request is formulated. As for the latter aspect, we provide the possibility (at the integration time) to annotate resource demands on the software component operations (i.e. Statechart transition) that can be critical with respect to performance. Optional annotations allow to concentrate software developers on critical aspects and subsystems of their architecture.

The simulation of an integrated architectural model allows to simultaneously take into account the software dynamics and the platform mechanisms that generate critical latency time in the software execution, mostly due to the resource contention.

In this paper we migrate *PAlib* to the UML 2 representation.

4 Tool Support to MOSES

When we have implemented our methodology in UML-RT, software designers used the UML 1.x standard version, Rational tools were recognized the more advanced ones for simulation support and they were widely used by software companies. In particular, Rose Real Time (RRT) [2] allowed us, along with the UML SPT profile [5], to easily model the dynamic behaviour of the resource models. In addition, RRT was equipped with an integrated simulation environment and with a powerful scripting interface.

Table 1. UML 2 Modeling Tools (S = Supported; N = Not supported)

Tool name	Simulation	Company	WWW
Magic Draw	N	No Magic	http://www.magicdraw.com
Poseidon for UML	N	Gentleware	http://www.gentleware.com
Apollo for Eclipse	N	Gentleware	http://www.gentleware.com
Enterprise Architect	N	sparxsystems	http://www.sparxsystems.com
TAU G2	S	Telelogic	http://www.telelogic.com
Omondo	N	Eclipse	http://www.eclipsedownload.com
Artisan Studio	S	ARTISAN Software	http://www.artisansw.com
Visual Paradigm for UML	N	Visual Paradigm Int. Ltd.	http://www.visual-paradigm.com
Rational Software Modeler	N	IBM	http://www.ibm.com
Together	N	Borland	http://www.borland.com

After the release of the UML 2 metamodel, a number of related tools have been created and much effort is continuously spent to extend and improve them. All these tools (some of which are listed in Table 1) provide visual modeling facilities, but at the moment only two of them (i.e. Artisan Studio and Telelogic Tau G2) offer an integrated simulation facility. This was one of the two requirements that we had for selecting the tool to run our UML 2 implementation. The other requirement was the possibility to insert code into the dynamic models, in particular into states and transitions between states. This capability is necessary to model resource requests originated from the software side.

Finally we have chosen Telelogic Tau G2 due to the peculiarities of its simulation environment briefly presented and compared with RRT in this section. The simulation environment provided by both RRT and TAU G2 is conceived to support systems engineers and software architects in verifying the correctness of the designed solution. It is possible to detect and fix errors and problems, since the early stages of the development life-cycle, by directly working on models. Both simulation engines can execute the models and provide graphical execution traces through Sequence Diagrams or allow step-by-step animation on Statecharts. In addition, a log file may be produced with details on the sequence of execution events. It is also possible to define and monitor some instance properties like attribute values, activity queues, current state in the Statecharts, etc.

From a performance analysis viewpoint, we have used both simulation engines as integrated model executors. Our simulations follow the evolution of the internal logic of the software and platform architectural components, thus computing the relevant performance parameters by means of the enabled probes.

One of the main differences of these simulation environments is the way they manage the time. RRT strategy is based on the machine real time clock. Thus, the time evolution during model execution can be significantly affected by the host environment (i.e. concurrent processes, CPU speed, etc.). While this problem may not be a main issue during model debugging and refining, it can represent a serious problem for performance indices computing because it is strictly coupled with time. To solve this problem we needed to implement a ”simulation time abstraction layer” that manipulated the ”target library” provided by RRT itself. In this way we have been able to run simulations based on a virtual clock that can be tuned with respect to the machine clock to guarantee the correct computation of performance indices. On the other hand, time required to complete the simulation for performance analysis might become too high. The worst case we experienced was 20 minutes for 1 second simulation.

The TAU G2 tool adopts a completely different strategy for time management. The time evolution during the model execution is not related at all to the host real time clock, rather it is based on the model event timing. All the scheduled events are enqueued basing on their trigger time. As soon as an event has been completely processed, the next event is scheduled and the simulation time is updated to the value of the trigger time of such event. The TAU G2 strategy is very efficiently implemented, it allows to speed up the whole simulation time and to guarantee an accurate performance indices estimation.

5 MOSES: The UML 2 Implementation

In order to tackle MOSES, that is the implementation in UML 2 of the general methodology (with the support of the TAU G2 environment) we have first studied the syntax and semantics of the UML 2 metamodel to find the relationships between UML-RT entities and UML 2 classifiers. Once completed this task, the implementation of the methodology proceeded as a straightforward activity.

The most important UML-RT entities to model resources and dispatching prototypes were capsules, ports and protocols. The corresponding classifiers we have identified in UML 2 are active classes, ports, and interfaces. Moreover, the internal behavior of classes and capsules in both UML-RT and UML2 can be modeled with Statecharts.

We had exploited the RRT capability to add source code into states and/or transitions in order to implement modeling details like: message sending between capsules, counters and probes computation, job queuing, and resources scheduling strategies. Writing code implies for the designers to move down their activity from the high level abstraction of the model to a lower implementation level.

With TAU G2 the designer is supported by a graphical language called SDL (Standard and Description Language), that is ITU-T Z.100 Recommendation, to describe Statecharts and to model actions. The SDL blocks represent the placeholders to include source code when the design progresses towards the implementation. Thus, designers can exploit SDL capabilities to delay the platform architecture specification of the required computational details.

5.1 Mapping UML-RT Stereotypes into UML 2 Metaclasses

We provide here a more detailed description of the metaclasses used in our software and platform architectural models and also the rationale that has led us to define such correspondences. Clearly the mapping presented here is one of the sound mappings that could be found. We have summarized the mapping of UML-RT stereotypes and UML 2 metaclasses in Table 2, and in the following we comment each row of the table.

Table 2. Model entity mapping between UML-RT and UML2

UML-RT STEREOTYPES	UML 2 METACLASS
Capsule	Active Class
Port	Port
Protocol	Interface
Signal	Signal
Connector	Connector
Class	Class

UML-RT Capsule - UML 2 Active Class. Capsules are the fundamental modeling element of real-time systems in UML-RT. Like Classes in UML 1.x, they can have operations and attributes and may also participate in dependency, generalization, and association relationships, but differently from Classes they represents independent flows

of control in a system. They also have several properties that distinguish them from Classes such as: the possibility to own public ports, the nesting to specify their internal organization and behavior, the message passing instead of method invocation, state machines for defining behavior instead of operations.

Differently from UML 1.x, the UML 2 Class is more expressive and can own ports as interaction points to send and receive signals, admits the nesting and can describe its internal behavior with a state machine. Thus we have naturally mapped an UML-RT Capsule in an UML 2 Active Class, that is a Class with a different thread of control for every class instance. Since an UML 2 Port is public by default, in order to precisely map the Capsule semantics an UML 2 Active Class only needs an elementary detail: to declare class operations and attributes as private thus leaving the only communication means to be public ports and signals.

UML-RT Port - UML 2 Port. The UML-RT Port is a fundamental classifier that describes the communication between Capsules. In our approach it is particularly relevant to highlight the exchange of resource demands between software and platform architectural models. Ports are objects that send/receive messages to/from capsule instances. Each port has its own identity but they are owned, created and destroyed by their capsule instance. To specify the set of messages sent to and from a port, a port is associated with a protocol role. The protocol role essentially defines the port type. In order for two ports to be connected by a connector, the ports must be compatible, namely every signal in the 'Out' set of one protocol role must be in the 'In' set of the other protocol role.

The Port Classifier, which did not exist in UML 1.x, has been introduced in UML 2. The semantics is quite similar to the one of UML-RT, that is the port represents an interaction point between a classifier (Class and/or Component) and its environment.

The main difference is in the definition of the interaction between two ports connected by a connector. In order to specify which messages can be sent to and from a port, in UML 2 the port is associated with interfaces, whereas in UML-RT the port is associated with protocol roles.

UML-RT Protocol - UML 2 Interface. An UML-RT Protocol is a contractual agreement defining the valid types of messages that can be exchanged between the participants in the protocol. Therefore a protocol is composed of the different participants, called protocol roles, that are defined by *sent* and *received* messages. Protocols are primarily used to identify the type of a port, where the latter plays the role of one participant in a communication relationship.

The classifier we adopted in UML 2 to map an UML-RT Protocol is Interface. An interface specifies a contract. It represents a declaration of a set of coherent public features and obligations and it is not possible to instance it. Instead, any instance of a classifier that realizes the interface must fulfill that contract. The set of *provided* interfaces of a classifier, which represent obligations, are the services offered to its clients. Instead, the services that a classifier needs in order to perform its functionalities specify its *required* interfaces that fulfill its own obligations to its clients.

UML-RT Signal - UML 2 Signal. The Signal classifier did not undergo significative changes. It has the same semantics in both modeling languages, UML-RT and UML2. A signal event occurs when a signal message, originally caused by a send action executed

by some object, is received by another object. It may be originated by actions or by the execution of an operation, and it represents the reception of a particular asynchronous message. The receipt of a signal event usually triggers a state transition in the state machine of the target object.

UML-RT Connector - UML 2 Connector. Connector in UML-RT is the element that allows interconnection for communication between cooperating capsules. The interconnection is built on compatible ports of the capsules.

In UML 2 it specifies a link that allows the communication between two or more instance of connectable elements.

Thus their semantics is similar except for the level of abstraction at which they can be used. In UML-RT a connector refers to the implementation level while in UML 2 it can refer to both implementation level or any more abstract level.

UML-RT Class - UML 2 Class. UML-RT Class describes at design time one or more distinct objects that have the same structure, attributes, operations and behaviour. At run time a number of instances of classes cooperate for a common goal.

The Class has received in UML 2 several important changes. It has maintained the properties that had in UML 1.x and has been enriched with additional ones. In fact Classes represent sets of objects that share the same characteristics, constraints and semantics. In addition new properties like nesting of classes and owning of ports have been added.

In this way, the translation has been quite simple thanks to the invariant semantics and also to the new ability of the classifier.

5.2 Two Example Prototypes: Round-Robin CPU and Main Dispatcher

In order to describe the implementation of *PAlib* in UML 2, we illustrate here the representation in UML 2 of two prototypes, i.e. a CPU prototype and a dispatcher prototype.

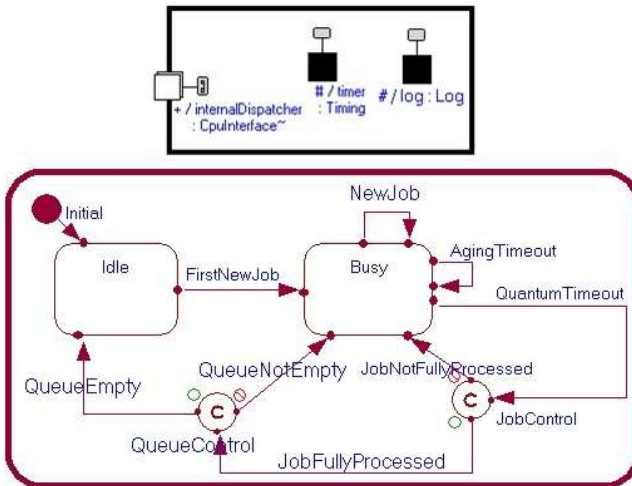


Fig. 2. UML-RT Capsule and Statechart of a Round-Robin CPU in RRT

Figure 2 shows the UML-RT implementation of a Round Robin CPU resource prototype, which includes the Capsule and the associated Statechart. As described in Section 3 the Round Robin CPU is a local resource, thus it is located in the bottommost layer of the resource side, and (behaving exactly like a local resource) satisfies the software requests of resources as described in the following. It leaves the idle state when the first job enters the resource. At this point two events may occur while being in the busy state. If a new job request enters the resource, then it is simply queued. If the CPU quantum expires for the currently processed job, then two alternatives are possible: (i) if the job has been fully processed, either the next job is extracted from the queue (if any) or the CPU goes back to an idle state, (ii) if the job still needs processing time, it is queued again and the next job is extracted from the queue and processed.

Figure 3 shows the same resource as implemented in UML 2 using an Active Class associated with the related Statechart. It has the same states and transitions as the UML-RT one, the UML-RT choice points correspond to the UML 2 decisional nodes, and guards and triggers names are differently represented in UML 2. We can note a difference in the UML 2 model in that two transitions and one action are not actually added since they are auxiliary for counters and probes computation. In UML-RT they are hidden in the code whereas in UML 2 they are only made explicit.

Figure 4 shows the UML-RT Capsule and the Statechart of a Main Dispatcher. As described in the section 3 the Main Dispatcher is located in the uppermost layer of the resource side. It dispatches the resource requests and provides a single access point to

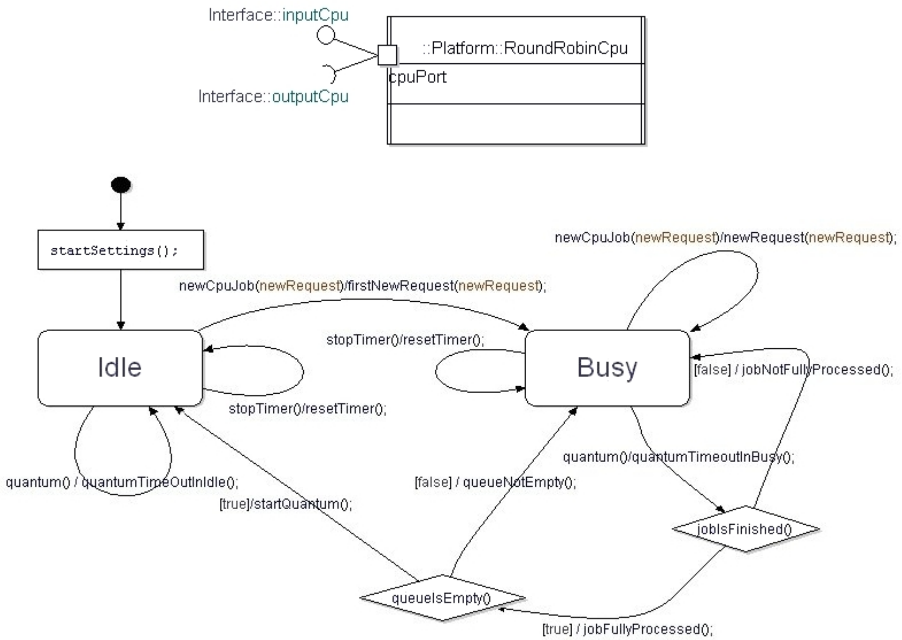


Fig. 3. UML 2 Active Class and Statechart of a Round-Robin CPU in TAU G2

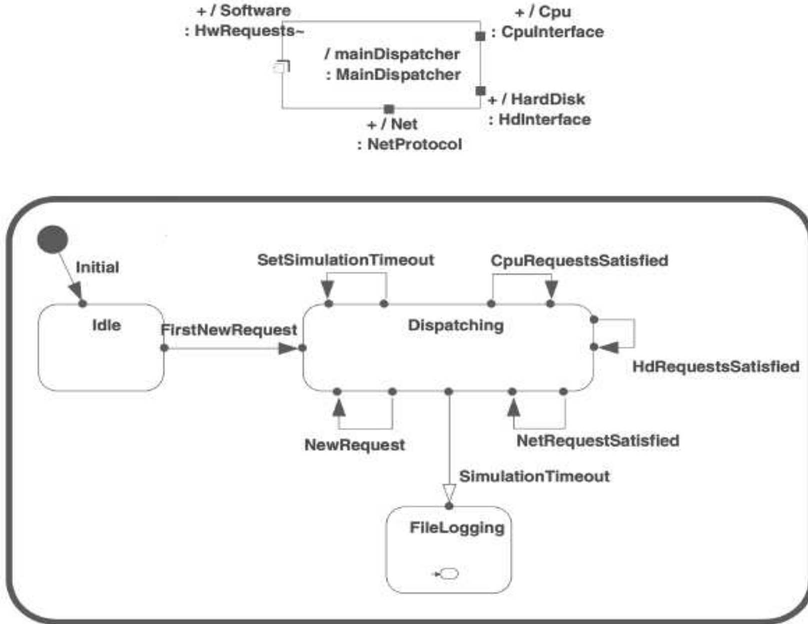


Fig. 4. UML-RT Capsule and Statechart of a Main Dispatcher in RRT

send resource request messages from software side to a specific processing node. It satisfies software requests of resources as described in the following. From an initial Idle state it migrates to the Dispatching state upon the arrival of the first resource request. It remains in this state until the Simulation Timeout expires (i.e., a message from the simulation control is received). This event triggers the transition to the File Logging state, where all the simulation statistics are collected and sent out as simulation results.

Figure 5 shows the UML 2 Active Class and Statechart of the Main Dispatcher. It has the same states and transitions as the UML-RT one, and guards and triggers names are differently represented in UML 2. We can note differences in the *endTimer* labeled transitions in the UML 2 model. One corresponds to the transition labeled *Simulation-Timeout* in UML-RT, whereas the other one has only been made explicit in UML 2.

5.3 Validation of the UML 2 Implementation

Once MOSES (UML 2 implementation of the *PAlib*) has been completed we have performed an experimental validation. The strategy we have adopted has been to replay one of the case study already implemented in UML-RT.This one allows us to clearly highlight incidental discrepancies in performance indices computation with respect to the real system test results is the one presented in [10]. The example focused on one of the system tests executed to verify the equipment compliance to a non-functional requirement related to the amount of the CPU load induced by the routing activity related to the traffic over a Telecommunication Management Network (TMN).

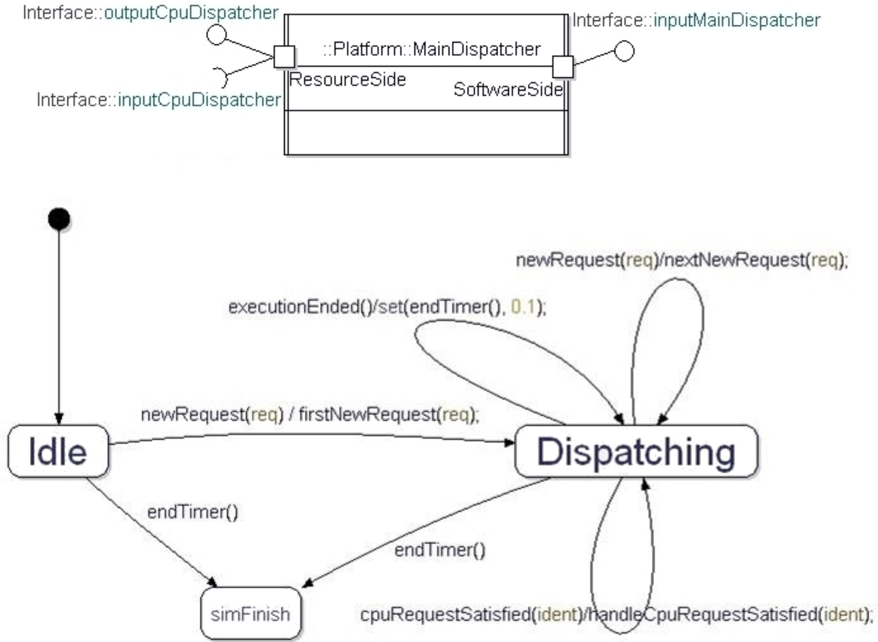


Fig. 5. UML 2 Active Class and Statechart of a Main Dispatcher in TAU G2

In more detail the case study is in the domain of an SDH telecommunication system in which there was a set of transmission nodes called Network Elements (NE), supervised by Element and Network Managers (EM, NM) for OAM&P (i.e. Operation, Administration, Maintenance and Provisioning) activity. Connections between NM, EM and NEs defined the Telecommunication Management Network (TMN) partially overlaid by the SDH network; i.e. NM, EM and closest NEs can be directly connected by Ethernet links, whereas farthest NEs can be connected by free SDH overhead bandwidth using the Data Communication Channel (DCC) bytes. In the example in figure 1 we have four EMs - Em1, Em2, Em3, Em4 that are instances of the Em prototype - which supervise four NEs (not represented in figure).

The NEs applicative software is split in two main components: (i) the Synchronous Equipment Management Function (SEMF) and (ii) the Message Communication Function (MCF) that acts as a router between the messages flowing through TNM and SEMF. In figure 1 we can see the instance of the Emf prototype which represents a single MCF that routes all the resource requests coming from the EMs residing in the software side. As explained in section 3, in the resource side there are three layers. In the example the uppermost contains the instance of the MainDispatcher prototype, the middle layer contains the instance of the CpuInternalDispatcher prototype, and the bottommost contains the instance of the local resource prototype that is the RoundRobinCpu. All these components behave as explained in section 3.

In both cases - the simulation and the real test - the Ems solicit the DCC channels that connect them with the Emf component by means of pings. This is done, during

the experimentations, dimensioning both the ping packets with different sizes, and the bandwidth with different occupancies.

The test focuses on the CPU performance related to the ISO/OSI protocol stack processing and ignores the application processing. The results of the simulation have been compared with the values obtained by the ”MC Test”, that is a test case to validate the equipment with respect to a specific system requirement asking for a CPU load less then 50% due to DCC traffic. Table 3 shows the results obtained by the system verification tests and the two sets of simulation results obtained with the UML-RT and UML 2 *PAlib* implementation. The table represents the percentage of the CPU load obtained stimulating the system with a ping generator able to generate different traffic rates. The results highlight that the TAU G2 simulation environment is more accurate and provides results that better reflect the model linearity. However, the comparison of these results has brought a proof of concept in support to the correctness of the UML 2 implementation.

Table 3. Simulation and test results

PING (pps)	*25*	*50*	*75*	*100*	*105*
MC TEST (%CPU)	11.56	22.68	33.72	43.86	46.18
Estimated UML-RT (%CPU)	10.9	21.9	32.8	43.8	46
Estimated UML 2 (%CPU)	11	22	33	44	46.2
Δ %CPU MC TEST vs UML-RT	0.9	0.78	0.92	0.06	0.18
Δ %CPU MC TEST vs UML 2	0.56	0.68	0.72	- 0.14	- 0.02

6 Conclusions

The work presented in this paper concerns our experience in developing a concrete framework in UML 2 on the basis of an existing architectural performance validation methodology.

From a notational viewpoint, we have compared the UML 2 framework implementation with the previous one based on UML-RT. In addition, with the support of the Tau G2 tool, we have also compared the results obtained by simulating a case study with the two UML implementations and the values obtained by a real test. This first experience not only has demonstrated at a certain extent the correctness of the implementations, but it has also shown that the available tools that support UML 2 modeling have larger benefits than the ones adopted to support UML 1.x.

Performance analysis can be carried out at different levels of abstraction and accuracy. In the presented methodology - and thus in the UML-RT and UML 2 implementations - the level of abstraction is not pre-determinated and is left to the designer. It depends on the modeled system and on what one intends to represent and observe. Besides, specific elements of our implementations is somehow related to the level of abstraction. With respect to the level of accuracy, in the presented methodology this is ruled by the accuracy of the simulation engine. After all, for sake of our methodology implementation, the only two major requirements for tools to be adopted have been: (i) the power of the simulation engine, (ii) the capability of annotating dynamic models with resource requests.

The separate modeling of software architectures and platform architectures is an interesting topic for modern software systems, where many attributes depend on the running platform, but where (at the same time) it is necessary to keep Platform Independent Models as a separate representation with respect to the Platform Specific Model. We intend to consolidate our approach in this direction, for example by extending the library of resources prototypes, in order to make easier the modeling of software and platform architectures in new application domains.

From a technical viewpoint, we plan to migrate the necessary annotations of MOSES from the UML SPT profile to the new MARTE profile which has been designed for the UML 2 metamodel. Besides, due to the wide diffusion of SysML-based approaches to design integrated software/platform models, we plan to study pros and cons of our approach with respect to SysML.

Finally, we plan to study the portability of MOSES over other simulation-based UML 2 tools, as well as experimenting the approach on large real world case studies to observe its scalability.

Acknowledgments

Authors would like to thank the reviewers for their excellent comments that have helped to improve the paper quality.

This work was partially supported by the European Community via the 6th FP IST PLASTIC project.

References

1. ARTISAN Sofware - ARTISAN Studio, http://www.artisansw.com
2. IBM Rational Rose Real Time, http://www-306.ibm.com/software/rational/
3. Telelogic TAU G2, http://www.telelogic.com
4. UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), http://www.omg.org/cgi-bin/doc?formal/05-07-04
5. UML profile for schedulability, performance, and time specification. formal/03-09-01, omg adopted specification, http://www.omg.org/technology/documents/formal/schedulability.htm
6. Arief, L.B., Speirs, N.A.: A UML tool for an automatic generation of simulation programs. In: Proceedings of the Second International Workshop on Software and Performance (WOSP2000), Ottawa, Canada, pp. 71–76. ACM, New York (2000)
7. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (2004)
8. Balsamo, S., Marzolla, M.: A simulation-based approach to software performance modeling. In: Proc. of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (2003)
9. Cortellessa, V., Gentile, M.: Performance modeling and validation of a software system in a RT-UML-based simulative environment. In: International Symposium on Object-oriented Real-time distributed Computing, pp. 52–59 (2004)

10. Cortellessa, V., Pierini, P., Rossi, D.: On the adequacy of UML-RTfor performance validation of an sdh telecommunication system. In: ISORC, pp. 121–124. IEEE Computer Society, Los Alamitos (2005)
11. Cortellessa, V., Pierini, P., Rossi, D.: Integrating software models and platform models for performance analysis. IEEE Trans. Softw. Eng. 33(6), 385–401 (2007)
12. de Wet, N., Kritzinger, P.: Using UML models for the performance analysis of network systems. Comput. Networks 49(5), 627–642 (2005)
13. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. J. Syst. Softw. 80(4), 528–558 (2007)
14. Isabel, D., Volker, S., Falko, D., Reinhard, G.: SYNTONY: Network Protocol Simulation based on Standardconform UML 2 Models. In: 1st ACM International Workshop on Network Simulation Tools (NSTools 2007), Nantes, France. ACM, New York (October 2007)
15. Kabajunga, C., Pooley, R.: Simulating UML sequence diagrams. In: Pooley, R., Thomas, N. (eds.) UK Performance Engineering Workshop (UK PEW), pp. 198–207 (July 1998)
16. Kirshin, A., Dotan, D., Hartman, A.: A UML simulator based on a generic model execution engine. In: Lecture Notes in Computer Science - Models in Software Engineering, pp. 324–326 (2007)
17. Ober, I., Graf, S., Ober, I.: Validating timed UML models by simulation and verification. STTT, Int. Journal on Software Tools for Technology Transfer 2005 (under press, 2004)
18. Petriu, D., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. Software and Systems Modeling (SoSyM) 6(22), 163–184 (2007)
19. Koziolek, H., Happe, J., Kuperberg, M., Reussner, R.H., Becker, S., Krogmann, K.: The palladio component model. In: Universität Karlsruhe (TH), Interner Bericht 2007-21 (2007)
20. Rolia, J.A., Sevcik, K.C.: The method of layers. IEEE Transactions on Software Engineering 21(8), 689–700 (1995)
21. Koziolek, H., Becker, S., Reussner, R.H.: Model-based performance prediction with the palladio component model. In: Proc. Workshop on Software and Performance (WOSP2007) (February 2007)
22. Sancho, P.P., Juiz, C., Pugjaner, R.: Integrating system performance engineering into MASCOT methodology through discrete-event simulation. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236. pp. 278–292. Springer, Heidelberg (2004)
23. Selic, B.: Using UML for modeling complex real-time systems. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 250–260. Springer, Heidelberg (1998)
24. Selic, B.: On software platforms, their modeling with UML 2, and platform-independent design. In: ISORC 2005: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), Washington, DC, USA, pp. 15–21. IEEE Computer Society, Los Alamitos (2005)
25. Woodside, M., Hrischuk, C., Selic, B., Bayarov, S.: A wideband approach to integrating performance prediction into a software design environment. In: WOSP 1998: Proceedings of the 1st international workshop on Software and performance, pp. 31–41. ACM, New York (1998)
26. Woodside, M., Hrishchuk, C., Selic, B., Bayarov, S.: Automated performance modeling of softwaree genrated by a design environment. Perform. Eval. 45(2-3), 107–123 (2001)
27. Yacoub, S., Ibrahim, A., Ammar, H.H., Lateef, K.: Verification of UML dynamic specification using simulation-based timing analysis. In: Proc. of 6th International Conference on Reliability and Quality in Design (ISSAT 2000), pp. 65–69 (August 2000)

Designing the Enterprise Architecture Function

Bas van der Raadt[1] and Hans van Vliet[2]

[1] Capgemini, Global Financial Services / Architecture & Governance Improvement, Papendorpseweg 100, 3528 BJ Utrecht, The Netherlands
bas.vander.raadt@capgemini.com
[2] VU University, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
hans@cs.vu.nl

Abstract. Enterprise Architecture (EA) is becoming an increasingly mature field of work, but many large organizations still struggle with implementing an integral and truly effective EA function. The literature provides a fragmented picture of the EA function, describing the various separate elements that make up the total package of activities, resources, skills, and competences of the EA delivery function. In our view, the EA function reaches beyond EA delivery and also includes the stakeholders, structures and processes involved with EA decision making and EA conformance. A holistic and integral view on the EA function is essential in order to properly assess an EA function on its performance, and to allow identifying the key points of improvement. In this article, we give such a description of the EA function, which provides the reference model in EA function performance assessments as part of our Normalized Architecture Organization Maturity Index (NAOMI) approach.

Keywords: Enterprise Architecture, Management, Organizational, Function, Reference Model, Governance, Conformance.

1 Introduction

The dream of every CEO is to have one standardized, integrated, flexible and manageable landscape of aligned business and IT processes, systems and procedures. Having complete control over all projects implementing changes in that landscape so that they deliver solutions that perfectly fit the corporate and IT change strategies, makes this dream complete. The reality for many large organizations is quite the opposite. Many large organizations struggle to keep their operational and change costs in control. Key reasons are the inflexibility and enormous complexity of their business and IT structures, processes, systems, and procedures, often distributed across lines of business (LoB) and business divisions (BD) spread out over various regions, countries or even continents [1], [2]. Over the last decade, Enterprise Architecture (EA) has been one of many instruments used by organizations in their attempt to get grip on the current operational environment and the implementation of changes. EA provides standardization, and sets a clear direction for the future to guide changes. Compared with architecture in the physical world, EA provides the

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 103–118, 2008.
© Springer-Verlag Berlin Heidelberg 2008

mechanism for city planning where software architecture is the architecture of one building. EA thus gives boundaries within which a software architect has to operate. Effectively applying EA leads to a reduction in operational maintenance costs due to increased discipline and control, as well as increased responsiveness because EA leads to reduced project duration. Additionally, EA improves risk management as it leads to reduced complexity, and it increases management satisfaction, because it provides an enterprise-wide view on organizational changes [3]. Finally, EA enhances strategic business outcomes because it helps increasing the effectiveness of business processes, applications, data and infrastructure through standardization [4].

Although seen as a vital management instrument by many large organizations [2], EA has generally not reached the desired result. EA has been practiced for at least ten years now, but it still suffers from relative immaturity, as we have experienced in various assessments of EA functions at client organizations (e.g., see [5] and Section 3). They have difficulties establishing an EA function that is fully integrated into the existing corporate or IT governance, as well as stimulating effective collaboration between architects and other stakeholders. Such a fragmented and badly integrated EA function typically fails to fulfill the expectations of all EA stakeholders which leads to the goals set with EA – if explicitly set at all – not being achieved.

In EA research, much effort has been put into various separate elements that make up the total package of activities, resources, skills, and competences that a mature EA delivery function should have in place – e.g., proper tools [6], frameworks [7]; [8] and architects [9]. In order to measure an organization's EA capability maturity, various EA function performance assessments (e.g., [5]; [10]; [11]) have been developed. These primarily focus on assessing whether the elements that determine the maturity of the EA delivery function (e.g. an architecture department or team) are in place. However, based on our experience, this is a limited view. The activities of the EA function should reach beyond merely delivering EA products and should also include other organizational roles, bodies, and activities responsible for EA decision making (e.g. an architecture council) and EA conformance (e.g. project managers and designers). An organization will only be effective with Enterprise Architecture when there is effective formal and informal interplay between the members of the EA delivery function and the stakeholders responsible for EA decision making and EA conformance. Moreover, the entire EA function must be properly integrated into the overall organizational and governance structures in order to be effective.

Currently, the literature lacks a complete reference model of an EA function. Existing EA capability maturity assessment approaches (e.g., [10]; [11]) have incorporated a reference model into their maturity model, but this model is often limited to the EA delivery function. Other practitioner's literature (e.g., [7]; [11]) provides a fragmented view of elements of the EA function. In this paper we provide a clear definition and integral description of the EA function, established into our EA function reference model. This model describes the norm we compare client organizations to, while assessing their EA function's performance. Both the EA function reference model and assessment model are part of our Normalized Architecture Organization Maturity Index (NAOMI) approach [5].

Section 2 of this paper contains our reference model of the EA function. Section 3 contains a case study that shows how one company has implemented its EA function.

In Section 4 we discuss the lessons learned regarding our EA function reference model based on this case study. Finally, in Section 5 we draw our conclusions and discuss future research we will conduct on the topic of the Enterprise Architecture function.

2 Reference Model

Based on scientific and practitioner's literature, and various case studies at a Global Financial Services Companies (e.g. [5]), we have created an integral description of the EA function. We define the EA function as: *The organizational functions, roles and bodies involved with creating, maintaining, ratifying, enforcing, and observing Enterprise Architecture decision-making – established in the enterprise architecture and EA policy – interacting through formal (governance) and informal (collaboration) processes at enterprise, domain, project, and operational levels.*

Based on this definition, we describe our EA function reference model with Section 2.1 describing its *structure*, Section 2.2 its *products*, Section 2.4 its process model, and Section 2.5 the *bodies and roles* involved. Section 2.3 provides a detailed description of *EA delivery* as part of the entire EA function.

2.1 Structure of the EA Function

Figure 1 shows the three main responsibilities of the EA function: (1) EA decision making, (2) EA delivery, and (3) EA conformance (see Fig. 1). *EA decision making* at strategic and tactical level is responsible for approving new EA products or changes in existing EA products, and for handling escalations regarding EA conformance. This is typically performed by one or more governance bodies (e.g. an EA council). Having such governance bodies in place – with proper representation from various stakeholder groups (see Section 2.5.1) – results in better perceived importance, involvement and support of both management and other stakeholders, and it improves effectiveness of the EA function [1]. EA governance bodies vary in the degree to which they have an advisory or formal decision making authority [12].

EA delivery is responsible for providing advice to guide EA decision making at strategic and tactical level. Additionally, EA delivery creates and maintains EA products, validates change results to see whether they conform to the EA, as well as provides support in applying EA products (see Section 2.3).

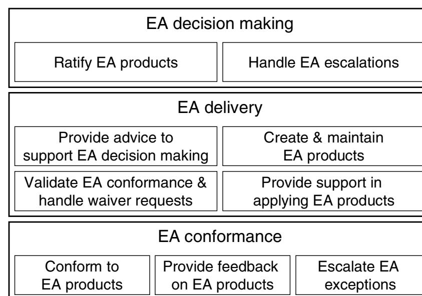


Fig. 1. Responsibilities of the three main Enterprise Architecture functions

Finally, *EA conformance* is responsible for implementing organizational changes through solutions as described in the target architectures, complying with the EA policy, and providing feedback on the applicability of the EA products to the EA delivery function. EA conformance is typically the responsibility of members of the organization who are affected by the EA products [13] while running change projects (e.g. project managers) or implementing operational changes (e.g. operational maintenance) at tactical and operational level.

2.2 Products of the EA Function

There are generally two types of EA products: (1) architectures and (2) EA policies [14]. An *architecture* document provides an abstraction of *what* a complex environment looks like, and acts as a means of communication and decision making regarding that environment [2]. Three types of architecture documents exist: (1) target state (to-be, soll) architecture that provides an abstraction of the desired situation, (2) current state (as-is, ist) architecture that describes the current operational environment, and (3) roadmap that describes a realization path from the current state to the target situation. These types of architecture documents aim at one or more of four aspect areas: (1) business architecture, (2) information architecture, (3) information systems, and (4) technical infrastructure [8]. The first two dimensions represent the business aspects of an organization; the latter two represent the IT aspects. In our view, Enterprise Architecture comprises both the business and IT aspects of an organization, and the alignment between them [2].

EA policy prescribes *how* projects should implement organizational changes across various LoBs and BDs through unified principles and practices. EA policies may be specified in three possible forms: (1) standards, (2) rules, or (3) guidelines. Both a standard and a rule must be adhered to; a guideline may be deviated from, provided a waiver has been granted. Enforcing EA policy enables organizations to influence the change activities of subunits without dictating exactly how they handle all of their operational activities [1]. Keeping up-to-date with industry standards allows organizations to change in a predictable way as a response to external developments [4], such as market changes, technological innovations and regulatory changes.

2.3 EA Delivery Function

The EA delivery function is often organized as a separate department [15] or team [1], typically as an organizational staff function. Depending on the size of the organization, the EA function may also consist of one or more individually operating architects. The EA department or team is sometimes led by a chief architect [1]. The origin of the EA function may differ, resulting in a difference in focus on either business aspects or IT aspects [2]. Regardless of the focus, there are generally four types of responsibilities of EA delivery:

1) Provide advice to support EA decision making regarding the target architecture by:

- Helping in building a vision and strategy for the future, based on its relation with its external environment regarding social, environmental and market developments, technological innovations, regulatory changes, etc.

- Describing decision alternatives regarding the target situation [16], and performing an impact analysis on predefined evaluation criteria and indicators (e.g. financial, regulatory) to determine the consequences of those alternatives in order for management to select the most desirable one [17]

2) Create and Maintain EA products that describe the:

- Current state architecture, which provides insight in the as-is situation of the operational environment, together with its bottlenecks and accompanying risks
- Concrete target state architecture, based on the vision and strategy, describing the chosen decision alternative in detail, which is assessed on its ability to cope with possible internal and external changes using various future scenarios
- Roadmap from the current state to the target situation, in which the mutual relation and impact of the elements in the architecture is described, and the sequence of implementation steps is given
- EA policies based on up-to-date knowledge of industry standards and developments within the organization, and determine their potential impact [4]

3) Validate EA conformance by:

- Reviewing programs or projects on their compliance with the applicable:
  - Target architectures at enterprise and domain levels, to ensure that individual program and project results contribute to achieving the general business goals and the target situation described in those target architectures
  - EA policies, to ensure that change activities of programs or projects contribute to achieving the standardization and integration goals set with EA
  - Current state architectures, ensuring the operational readiness of the program and project results before deployment, thus safeguarding the continuity of the operational processes and systems
- Handling waiver requests, assessing the implications of allowing programs and projects that file the requests to deviate from a specific guideline

4) Provide support in applying EA products towards programs and projects (e.g. through training and coaching) in:

- Creating program and project target architectures based on the EA products at domain and enterprise levels
- Conforming to the EA products in running programs and projects

2.4 EA Process Model

Pulkkinen [18] describes an EA process model for the management of architectural decisions in enterprise architecture planning that has three abstraction layers: (1) enterprise level, (2) domain level, and (3) systems level. Decisions made at higher management levels are made explicit in EA products that flow downwards to lower levels, introducing more detail. The architectures and EA policies at a higher level set the boundaries for decision making and implementation at lower levels. From our practical experience with implementing EA functions, this has proven to be an appropriate model. However, based on our practical experience and an exploratory study on the stakeholder's perception of EA performance [19], we altered and extended the EA process model.

The EA process model makes a distinction between *permanent* (e.g., business process chains, BDs, or LoBs) and *non-permanent* (e.g., large programs) domains. However, it is also vital to make a distinction between specific and generic business domains because of their conflicting operating models as a result of different optimization principles. A *specific* domain typically entails a customer facing LoB, which provides a specific product or service, servicing a specific market or client segment, or operating within a defined geographical region. It therefore optimizes its operating model in order to fine tune its services to the needs of its customers [20]. On the other hand, a *generic* business domain (e.g., a shared service center) typically offers generic or infrastructural services to various LoBs and BDs within an organization – thus acting as a cost center – optimizing its structures, processes, systems and procedures so it can minimize its operational costs [21]. In order to best deal with the horizontal integration of specific and generic domains, EA decision making may be centralized, decentralized or implemented in a federal model, depending on the organizational characteristics [12].

We changed the name of 'systems level' into '*project level*'. This leaves open what type of solutions projects deliver. The term 'systems level' suggests that EA decision making and implementation always results into an IT solution [18]. However, within the business and information architecture aspect areas [8], projects may deliver case handling processes that require human involvement and physical information flows (through paper forms); it is not always possible to fully automate business processes into Straight Trough Processing (STP) [22].

Also, we added an *operational level* to the process model, because of the conflict in decision making regarding organizational changes at project level, and organizational stability and continuity at operational level [23]. Decision making about exploiting a continuous and repeating operational environment aims at refinement, through predictable small impact changes, to maximize its continuity and stability. Decision making at project level often is different in nature, because it concerns realizing less predictable high impact changes in the operational situation, potentially compromising the continuity and stability at operational level.

Enterprise-wide decision making – as is the case with EA – should encompass feedback from group and individual levels to ensure continuous improvement [24]. However, in practice such a feedback process is hardly performed. EA decision makers (e.g. senior management) feel that a one yearly decision making cycle is adequate in managing changes [25]. The EA process model incorporates a learning cycle with a downstream flow of decisions (*feed-forward*), and an upstream flow to feed the successes and constraints of implementing those decisions at lower levels back to higher levels (*feedback*) [18]. We elaborate on these concepts using the organizational learning framework of Crossan et al. [24]. We translated the four underpinning key premises of their framework to the situation of EA to enhance the EA process model (see Table 1).

In parallel with the organizational learning theory, the Enterprise architecture practice experiences a tension between *exploration* of new possibilities and *exploitation* of old certainties [26]. EA exploration takes place during decision making at enterprise and domain levels, and results in new architectures and EA

Table 1. The Organizational Learning premises taken from Crossan et al. [24] specified to the Enterprise Architecture construct space.

Premise	Organizational Learning (OL)	Enterprise Architecture (EA)
1	OL involves a tension between assimilating new learning (exploration) and using what has been learned (exploitation).	EA involves a tension between creating new EA products through exploration and exploiting the existing EA products that describe the operational structures, systems and processes, and prescribe the current standards and procedures.
2	OL is multi-level: individual, group, and organization.	EA is multi-level: enterprise, domain, project, and operational.
3	The three levels of OL are linked by social and psychological processes: intuiting, interpreting, integrating, and institutionalizing.	The four levels of EA are linked by formal (governance) and informal (collaboration) processes.
4	Cognition affects action, and vice versa.	Theory (architectures and standards) affects practice (change projects and operational structures, processes and systems), and vice versa.

policies being created and approved. Following, these EA products, describing how changes should be implemented, are fed forward to project and operational level, where these are to be interpreted and followed. The EA delivery function plays a vital role, as mediator between EA decision making and EA conformance, in getting this shared understanding and common behavior. It requires a more pro-active attitude than merely writing down the central decisions and publishing them so that they are available to lower levels. Having an integrated and effective roll out and acceptance plan is vital for the EA delivery function to realize this organizational change [27].

Feedback is vital in respecting the constraints and problems that arise at project or operational level with applying the EA products prescribed. These may not have been anticipated during EA decision making at domain or enterprise level. *Informal feedback* during the collaboration between architects and EA stakeholders at lower levels allows continuous improvement of EA products through refinement of EA decisions at higher levels. This ensures their practical applicability and prevents them from being exclusively used by architects [2]. *Formal feedback* through escalation of EA conformance exceptions and waiver requests also provides vital information for improving the EA products; the number of escalations and waiver requests regarding a specific EA products acts as a quality indicator of that product.

Also, feedback allows incorporating what has been learned at lower levels, through exploration, experimentation and innovation, into EA products prescribed at higher levels. *Best practices* (e.g., a proof-of-concept of a new, innovative technology) at project or operational level are identified and evaluated on their generic applicability [18]. This leads to a *proposal* for changes in existing, or the creation of new, EA products. When ratified, an EA product receives the formal status ‘approved’, and will go through the validity statuses: future, actual, confined and obsolete, before receiving the formal status ‘retired’, introducing an *EA product life cycle*.

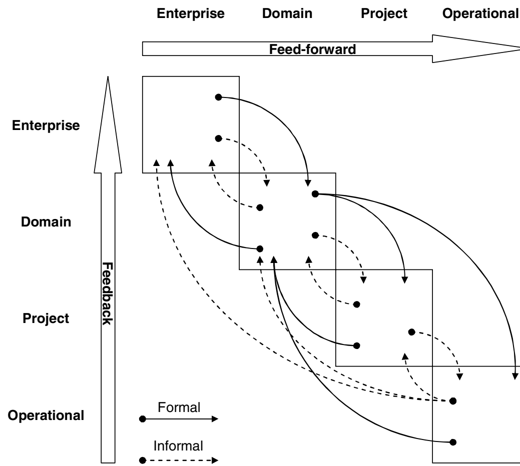


Fig. 2. EA Process Model including a learning cycle of feed-forward and feedback across enterprise, domain, project and operational levels

Figure 2 shows the EA learning cycle constructed of formal and informal EA processes at various organization levels. Table 2 describes the in and output regarding the feed-forward and feedback of these formal and informal processes.

2.5 Bodies and Roles within the EA Function

Within the process model of the EA function described in Section 2.4, various bodies and roles interact while pursuing different objectives and goals.

2.5.1 Bodies and Roles within EA Decision Making

The EA governance bodies within the EA function are responsible for decision making about EA products, giving them a formal status. Also, it handles escalations of non-conformity. An effective EA governance body at any organizational level should: (1) be composed of the various roles that represent the potentially conflicting interests that occur at that organizational level, (2) perform transparent decision making based on objective criteria, and (3) have the proper mandate to enforce the decisions at that organizational level.

The *EA council* at enterprise level acts as a steering committee [1] in order to achieve horizontal integration for coordinating of EA decision making [12]. It is comprised of representatives of the domains within the organization, the chief architect, and a chairman. Both the chairman – a key EA sponsor – and the chief architect – responsible for the quality and effectiveness of the overall Enterprise Architecture – should act in the interest of the enterprise-wide structures, processes, systems and procedures to achieve the corporate strategy. The domain owners are concerned with optimizing their specific domains to achieve their domain specific strategies. When issues cannot be resolved within the EA council, they are escalated towards senior management sponsoring the EA council for final decision making.

Table 2. EA functions and activities performed at enterprise, domain, project and operational levels; at each level input and output is fed forward or back, creating the EA learning cycle

Organizational level	Functions & activities	Formal processes	Informal processes
Enterprise	EA decision making EA delivery	Feed-forward: • (Out) Validate domain level EA conformance Feedback: • (In) Handle domain level EA escalations and waiver requests	Feed-forward: • (Out) Provide support in applying EA products at domain level Feedback: • (In) Use feedback to maintain enterprise level EA products • (In) Use operational expert knowledge and data in EA decision making
Domain	EA decision making EA delivery EA conformance	Feed-forward: • (In) Conform to enterprise level EA products • (Out) Validate project and operational level EA conformance Feedback: • (In) Handle project and operational level EA escalations and waiver requests • (Out) Escalate domain level EA exceptions, and file waiver requests towards enterprise level	Feed-forward: • (In) Utilize support in applying EA products • (Out) Provide support in applying EA products at project and operational level Feedback: • (In) Use feedback to maintain domain level EA products • (In) Use operational expert knowledge and data in EA decision making • (Out) Provide feedback on existing or potentially new EA products
Project	EA conformance	Feed-forward: • (In) Conform to domain level EA products Feedback: • (Out) Escalate project level EA exceptions, and file waiver requests	Feed-forward: • (In) Utilize support in applying EA products • (Out) Provide support in deploying the project result Feedback: • (In) Use operational expert knowledge to run project • (Out) Provide feedback on existing or potentially new EA products
Operational	EA conformance	Feed-forward: • (In) Conform to domain level EA products Feedback: • (Out) Escalate operational level EA exceptions, and file waiver requests	Feed-forward: • (In) Utilize support in deploying the project result Feedback: • (Out) Provide operational expert knowledge and data

At domain level, there may be a formal authority or informal advisory EA governance body (e.g., *domain architecture council*), which is responsible for EA decision making within that domain. Membership is similar to the EA council, only the roles stay within the domain. The domain architecture council handles the ratification of domain specific EA products and handles the escalations regarding non-conformity with those EA products. Only when disputes regarding non-conformity cannot be resolved within the domain architecture council, or the impact of decisions made by the domain architecture council reaches beyond that domain, is that issue escalated towards the EA council at enterprise level. This reduces the workload for the EA council to only the hard-to-resolve, domain overarching issues.

At project level there is typically no formal EA governance body. The *project steering committee* may act as an informal EA governance body. Issues of non-conformity that cannot be resolved and may lead to a project deviating from the enforced EA products will be escalated towards the domain architecture council.

2.5.2 Roles within EA Delivery

At enterprise level, the EA delivery function usually consists of a central EA team [1] or staff department [11], comprised of an EA manager, the chief enterprise architect, and various enterprise architect roles. Each *enterprise architect* is responsible for a specific EA aspect area (i.e. business, information, information system, or technical infrastructure [8]), performing the primary activities of the EA delivery function (see Section 2.1) at enterprise level. The *chief enterprise architect* [1] typically acts as the functional lead of the EA delivery function, overseeing all aspect areas of the enterprise architecture. He or she acts as trusted advisor to the CxO, and is responsible for the quality and effectiveness of the overall Enterprise Architecture. The *EA manager* runs the EA delivery function, performing budget and resource management, planning and coordination, and other operational management tasks.

Organizational domains (e.g., LoBs) typically employ their own specific architects at domain level, who are experts in a specific business or IT area. The *domain architect* acts as trusted advisors to the domain owner (e.g. head of the LoB). Depending on the size and structure of the domain level EA delivery function, autonomously operating architects, a team of architect-like roles, or a formal architecture department may be present. This domain level EA delivery function acts as a *sub-team* [1] of the central EA delivery function at enterprise level.

2.5.3 Roles within the EA Conformance

The members of a project team are responsible for managing and running change projects. These projects should deliver solutions that transform specific parts of the organization's operational environment into the desired situation described in the target architecture(s) at enterprise and domain levels. Additionally, they should comply with EA policy while running the project. A special role is *project architect*, who acts as an advisor guarding the quality of the project. He or she provides advice in the start-up phase of a project to discuss the important implementation decisions, and is responsible for the delivery of a project design which complies with the enforced EA products. Also, the project architect should provide feedback on the practical applicability of EA products towards the domain level EA delivery function.

A project architect is not member of the EA delivery function. This role has project result responsibility, and can therefore not perform project validations independently.

At operational level, the EA delivery function performs a *gatekeeper* role, performing post implementation reviews of: (1) solutions projects deliver and (2), changes made in the operational environment. In performing these reviews, changes are assessed on operational readiness and EA conformance before being deployed.

3 EA Function at a Large International Company

We conducted a case study at a large international company, henceforth called company A, assessing its EA function against our EA function reference model. We held fully structured interviews with various roles – i.e., domain owners, EA council members, program and project managers, operational managers, architecture managers, architects, designers, subject matter experts – addressing the assessment topics which are part of our NAOMI approach [5]. Also, the assessors studied an extensive set of strategic, project, operational, and communication documents, in order to check the findings from the interviews. With these findings we created an image of the EA function, and compared these to the reference model described in Section 2.

We have done a second case study of the EA function within a comparable organization (company B) using the same approach. This case study had comparable results. For confidentiality reasons, no details hereof can be given. In the case description in this section, we indicate which findings we confirmed with the case study conducted at company B, and which findings were different.

The case study we conducted involved the assessment of an EA function within the operations and IT division of a large international company, with technical infrastructure as the primary focus area. This back-office division consists of various verticals providing operational and IS services to the various LoBs within the front-office, as well as a technology department providing infrastructural services to the verticals. The EA function, as part of the technology department, is responsible for creating enterprise wide infrastructure policies and validating solution designs on their conformance. The EA delivery function consists of a team of architects, each an expert regarding a specific infrastructural domain (e.g., storage, mainframe, internet, etc.), responsible for creating EA policy and performing conformance validations related to that domain. When a solution touches several infrastructural domains, it had to be validated by each domain architect responsible for those domains. The chief infrastructure architect and the infrastructure domain architects held a monthly meeting to approve new infrastructure policies. This was not a formal EA council with representatives from the verticals responsible for EA decision making regarding the infrastructural policies. The infrastructure policies did have impact for those verticals. This monthly meeting resulted in few policies getting a formal status. There was no standard procedure, for policies that received a formal status, to store and publish them in one central repository.

Company B did have a formal EA council with proper representation from the Business Divisions (BD) within the company. The EA council, however, was also

unable to assess and approve EA policy proposals created by the EA delivery function, and provide them with a formal status.

Our assessment of the EA function in company A showed that there was no enterprise infrastructure architecture written down that describes the relations and coherence between the infrastructure domains. This resulted in inconsistent and incoherent EA policies across the infrastructure domains. The domain architects provided conflicting advice to the project managers and designers, because they collaborated insufficient with each other, and did not have a enterprise infrastructure architecture to guide them. This made creating a coherent solution design that complied with the EA policies complex for the designers, which frustrated them. Many designers also had little experience with creating solution designs according to the template provided by the EA function. Many solution designs sent to the EA function for validation were therefore of low quality, and were either found inadmissible or were rejected.

Company B did have an enterprise architecture that described the relations and coherence between domains. However, this enterprise architecture wasn't detailed enough to provide a concrete reference for the domain architects. This resulted in similar problems regarding conflicting architectures, policies, and advice by the EA delivery function we found at company A.

The conflicts of opinions and insufficient collaboration between domain architects at company A caused the validation outcome of solution design to be unpredictable; the result depended on which architect performed the validation. All involved domain architects had to accept the solution design in order for the project to receive a building permit. Projects sometimes had to wait months in order for their design to be accepted, because the domain architects could not agree on the outcome. The feedback projects got on the rationale why a solution was rejected, and the explanation on what to improve in their design in order to pass the validation successfully was often insufficient.

In order to deviate from a policy, or request permission to continue implementing the solution when the design was rejected by the EA function, project managers at company A could request a waiver. Decision making about granting projects a waiver was not transparent; they were granted based on undefined criteria, and inadequately communicated to the stakeholders. Domain architects were not always informed about a granted waiver. During the next solution validation they rejected the solution of a project that were granted a waiver. This resulted in projects being stopped even though a waiver was granted, to the frustration of various EA stakeholder groups.

Company B had a similar procedure for projects to request permission to deviate from a EA policy. The EA council that handled these requests was not fully effective.

There were too many EA policies at company A. They were unstructured, and the formal status of many of them was often unknown; there was no life cycle and change management for the policies. The EA policies the domain architects created were often not tested before they were implemented. Because there was no feedback loop from project level upwards, the domain architects were not aware of the practical applicability of the EA policies. There was no central administration of escalations and waiver requests to allow identifying malfunctioning EA policies to be changed. This all resulted in many projects deviating from the EA policies because they were impossible to work with.

The EA policies at company B were also not tested before they were implemented, and there was no feedback loop from projects upwards. Company B did have a central administration of escalation and waiver requests, but these were not used to identify malfunctioning EA policies for improvement.

4 Lessons Learned

Section 3 describes only a fraction of the findings we collected during the EA function assessment we conducted at the large international company A. However, this case shows that the EA maturity level in this company requires quite some improvement. It also shows it is insufficient to only take EA delivery into account to be truly effective with EA; both EA decision making and EA conformance have to be considered as well. In this section we elaborate on the key lessons we have learned.

1) Governance and collaboration must go hand in hand
The case study at company A shows that, if there are no formal and informal structures and processes, it is hard for EA stakeholders to trust each other and to work together. For example, an informal process of EA delivery performing an intake to pro-actively explain projects that are starting up how to create a solution design that satisfies the desired quality criteria, and conforms to the policies may help considerably. This will result in project managers and designers to better understand the purpose and working of solution validations, and deliver high quality designs. However, formal processes are also required. For example, having a transparent policy approval procedure, and a standard procedure for publishing the policies in a central, well-structured repository. This would make it more clear for the EA stakeholders, who are to conform with the policies, which EA policies apply to them. Therefore, it is vital to have both formal and informal structures and processes in place [28]. Formal processes ensure proper connection and coordination of EA decision making and conformity. Informal processes stimulate collaboration. Only combining both allows an effective implementation of EA governance in complex and dynamic environments [12].

2) Don't omit steps in the process model; keep the learning cycle in tact
A feedback loop is essential in getting EA products to be accepted and adhered to at project level. For example, the case study at company A shows that EA policies were not tested, and were not always applicable in practice. By ensuring a feedback loop from projects to the domain architects will solve this issue. This feedback loop may be implemented in the formal processes (i.e., make changes to policies based on escalations and waivers), or informal processes (e.g., by having regular meetings between architects who create the policies and designers who use them).

Having architectures at enterprise and domain level which are connected is vital in getting horizontal integration across domains. For example, the case study at company A shows that there was no enterprise infrastructure architecture available for the domain architects in order to integrate the various infrastructural domains. This illustrates that if one or more steps in the EA process model is omitted, the EA learning cycle becomes incomplete, with negative results.

3) Keep decision making and conformance reviews transparent and consistent
In order for the EA stakeholders to accept EA decision making and EA conformance validation results, it is vital to be transparent and consistent [12]. For example, the case study at company A shows that an unpredictable and unexplained validation result leads to frustration with the project manager and designer. This frustration will decrease with a transparent and consistent validation process, providing that proper feedback is given to guide the validation outcome. Regarding EA decision making, again transparency and consistency is essential. For example, the case study at company A shows that EA stakeholders will become frustrated with impractical and conflicting EA policies, and opaque EA decision making. In this case, transparent decision making regarding policies by representatives of the verticals impacted by those policies will increase the acceptance with the EA stakeholders within those verticals.

4) Governance bodies must represent all EA stakeholder groups with conflicting interests
An organization typically consists of various stakeholder groups at different organizational levels that have conflicts of interest, resulting in power struggles and political disputes [29]. For example, in the large international company we performed our case study there was a conflict of interest between solution delivery centers within the verticals delivering IT solutions, and the data center that deploys those IT solutions. The solution delivery center is concerned with providing a solution that best fits the business requirements; the data center wants to ensure the stability and continuity of the data center. The composition of an EA governance body is vital in properly addressing these conflicts of interest in decision making in order for EA governance to be effective [12].

5 Conclusions

Up till now, the literature provided a fragmented description of the EA function. In this article, we provide an integral description of the EA function in order to set the norm in performing EA function performance assessments. The case study we discuss in detail in this article shows that the maturity of EA functions is typically quite low, resulting in low performance of those EA functions. A second case study – due to reasons of confidentiality we do not describe this case study in detail in this article – confirmed this. In order to properly identify the essential points of improvement and compose an effective improvement plan, one needs a holistic perspective on the EA function. Comparing a specific EA practice with our integral EA function reference model, using an assessment model describing the standard topics of investigation, allows for EA practices to be compared with each other. Our NAOMI approach provides both an EA function reference, and assessment model. In this article we describe our EA function reference model we use to design and implement EA functions within organizations based on the assessment outcome.

In order to better understand what determines the performance of the EA function, we are conducting an empirical study to validate our EA performance framework, which addresses the three main topics of EA efficiency, EA effectiveness, and EA stakeholder satisfaction. We conducted an exploratory study on EA stakeholder

perception of EA function performance [19], and we are currently constructing a stakeholder satisfaction assessment approach based on that exploratory study in order to extend our NAOMI approach.

References

1. Boh, W., Yellin, D.: Using Enterprise Architecture Standards in Managing Information Technology. Journal of Management Information Systems, 23(3), 163–207 (2007)
2. Van der Raadt, B., Soetendal, J., Perdeck, M., Van Vliet, H.: Polyphony in Architecture. In: Proceedings 26th International Conference on Software Engineering (ICSE 2004), pp. 533–542. IEEE Computer Society, Los Alamitos (2004)
3. Ross, J.W., Weill, P., Robertson, D.C.: Enterprise Architecture as Strategy – Creating a Foundation for Business Execution. Harvard Business School Press, Boston (2006)
4. Bird, G.B.: The business benefit of standards. StandardView 6(2), 76–80 (1998)
5. Van der Raadt, B., Slot, R., Van Vliet, H.: Experience Report: Assessing a Global Financial Services Company on its Enterprise Architecture Effectiveness Using NAOMI. In: Proceedings of the 40th Annual Hawaii international Conference on System Sciences (HICSS 2007), p. 218b. IEEE Computer Society, Washington (2007)
6. Peyret, H., Leganza, G., Hoekendijk, C., King, O., McCormack, M., Carini, A.: Enterprise Architecture Tools. In: The Forrester Wave™, Q2, April 25 (2007)
7. Open Group.: TOGAF (The Open Group Architecture Framework) version 8.1.1 ('The Book') (2007), `http://www.opengroup.org/bookstore/catalog/g063v.htm`
8. Mulholland, A., Macaulay, A.L.: Architecture and the Integrated Architecture Framework. Capgemini (2006), `http://www.capgemini.com/services/soa/ent_architecture/iaf/`
9. Clerc, V., Lago, P., Van Vliet, H.: The Architect's Mindset. In: Overhage, S., Szyperski, C.A., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880. pp. 231–249. Springer, Heidelberg (2008)
10. META Group, Inc.: Architecture Capability Assessment. META Practice, Vol. 4(7), META Group, Inc. (2000)
11. Van den Berg, M., Van Steenbergen, M.: Building an Enterprise Architecture Practice: Tools, Tips, Best Practices, Ready-to-use Insights. Springer, Heidelberg (2006)
12. Peterson, R.: Crafting Information Technology Governance. Information Systems Management 21(4), 7–22 (2004)
13. Smolander, K., Päivärinta, T.: Describing and Communicating Software Architecture in Practice: Observations on Stakeholders and Rationale. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 117–133. Springer, Heidelberg (2002)
14. Ross, J.W., Beath, C., Goodhue, D.L.: Develop long-term competitiveness Through IT assets. Sloan Management Review 38(1), 31–45 (Fall, 1996)
15. Van der Raadt, B., Hoorn, J.F., Van Vliet, H.: Alignment and Maturity Are Siblings in Architecture Assessment. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS. vol. 3520, pp. 357–371. Springer, Heidelberg (2005)
16. Simonsson, M., Lindström, Å., Johnson, P., Nordström, L., Grundbäck, J., Wijnbladh, O.: Scenario-Based Evaluation of Enterprise Architecture - A Top-Down Approach for CIO Decision-Making. In: Proceedings of the International Conference on Enterprise Information Systems (ICEIS 2005) (May 2005)

17. Kazman, R., Klein, M., Clements, P.: ATAM: Method for Architecture Evaluation. Technical Report, CMU/SEI-2000-TR-004 (2000), http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr004.pdf
18. Pulkkinen, M.: Systemic Management of Architectural Decisions in Enterprise Architecture Planning. Four Dimensions and Three Abstraction Levels. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS 2006), p. 179a (2006)
19. Van der Raadt, B., Schouten, S., Van Vliet, H.: Stakeholder Perception of EA Function Performance, In: Second European Conference On Software Architecture (ECSA 2008), (Submitted to, April 2008)
20. Moore, G.A.: Strategy and your stronger hand. Harvard Business Review 83(12), 62–71 (2005)
21. Janssen, M., Joha, A.: Issues in relationship management for obtaining the benefits of a shared service center. In: Janssen, M., Sol, H.G., Wagenaar, R.W. (eds.)Proceedings of the 6th international Conference on Electronic Commerce, ICEC 2004, Delft, The Netherlands, vol. 60, pp. 219–228. ACM, New York (2004)
22. Van der Aalst, W., Hofstede, A., Weske, M.: Business process management: A survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 1–12. Springer, Heidelberg (2003)
23. Leana, C.R., Barry, B.: Stability and Change as Simultaneous Experiences in Organizational Life. The Academy of Management Review 25(4), 753–759 (2000)
24. Crossan, M., Lane, H., White, R.: An organizational learning framework: From intuition to institution. Academy of Management Review 24, 522–537 (1999)
25. Baker, B.: The role of feedback in assessing information systems planning effectiveness. The Journal of Strategic Information Systems 4(1), 61–80 (1995)
26. March, J.G.: Exploration and Exploitation in Organizational Learning. Organization Science 2(1), 71–87 (1991)
27. Kotter, J.P.: Leading Change. Harvard Business School Press, Boston (1996)
28. Henderson, J.C.: Plugging into strategic partnerships: The critical IS connection. Sloan Management Review 31(3), 7–18 (1990)
29. Eisenhardt, K.M., Bourgeois III, L.J.: Politics of Strategic Decision Making in High-Velocity Environments: Toward a Midrange Theory. The Academy of Management Journal 31(4), 737–770 (1988)

Quality Prediction of Service Compositions through Probabilistic Model Checking

Stefano Gallotti, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli

Politecnico di Milano
DeepSE Group–Dipartimento di Elettronica e Informazione
Piazza Leonardo Da Vinci, 32 – 20133 Milano, Italy
{gallotti,ghezzi,mirandola,tamburrelli}@elet.polimi.it

Abstract. The problem of composing services to deliver integrated business solutions has been widely studied in the last years. Besides addressing functional requirements, services compositions should also provide agreed service levels. Our goal is to support model-based analysis of service compositions, with a focus on the assessment of non-functional quality attributes, namely performance and reliability. We propose a model-driven approach, which automatically transforms a design model of service composition into an analysis model, which then feeds a probabilistic model checker for quality prediction. To bring this approach to fruition, we developed a prototype tool called `ATOP`, and we demonstrate its use on a simple case study.

1 Introduction

Service-Oriented Architectures (SOAs) provide a new paradigm for the creation of business applications. This paradigm enforces decentralized developments and distributed systems compositions: new added-value services may be created by composing independently developed services. Web services are an increasingly important and practical instance of SOAs, supported by standards and by specific technology. Typically, services can be composed in an *orchestrated* manner by using a workflow language, like the Business Process Execution Language (BPEL) [4].

We argue that SOAs can benefit from the Model Driven Development (MDD) [6] paradigm. In essence, this means that models are built to support software engineers in reasoning at the software architecture level. As a satisfactory solution is built at the model level, transformation steps (possibly automated) derive the final, platform-specific implementation. In the case of SOAs, model-level reasoning should support the early QoS assessment of a service composition. The composition may be assessed at design time, before a concrete binding from the workflow to the externally invoked services is established. The assessment is thus performed on the abstract workflow. It is requested, however, that a specification of the external services in terms of their functional and non-functional attributes is available. The actual binding from the abstract workflow to concrete services may then be established dynamically at run time, provided that the selected

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 119–134, 2008.
© Springer-Verlag Berlin Heidelberg 2008

concrete services fulfill their specification. This may be enforced by a suitable QoS-driven binding mechanism.

The use of models extends beyond the initial development of an application. Models may be used to support evolution of the software architecture. They can also be useful to devise suitable reconfiguration strategies for the dynamic contexts where the application will be deployed. Once the application is running, model-based reasoning may be used to predict the impact of different reconfigurations in a changing context, driving in this way the reconfiguration process.

In this perspective, hereafter we tackle the following two issues: i) which kind of model is suitable for quality analysis of service-based applications; ii) how we can support the construction of such a model.

Concerning the first issue, we build on past work on architectural reasoning and analysis of quality aspects through model checking [10,11], and in particular on the probabilistic model checker PRISM [34], which was used for a preliminary assessment in [23]. This choice is motivated both by the encouraging results we achieved, which demonstrated the applicability of these techniques to a wide set of systems, and by the existence of tools implementing these techniques.

Concerning the second issue, we leverage on the aforementioned MDD paradigm, to transform a high-level description to executable code. To perform analysis of non-functional quality attributes at the model level, we propose a model transformation step that takes as input a "design-oriented" model of the software system (plus some additional information related to the non-functional attribute of interest) and generates an "analysis-oriented" model, that lends itself to the application of an analysis methodology [26]. Specifically, we provide an integrated framework that, starting from an high level description of the service composition, given in terms of activity diagrams, automatically derives stochastic models that can be solved using the different features of the PRISM model checker. The provided methodology and tool are called ATOP, which stands for *from Activity diagrams TO Prism models*. Besides, we try to overcome a weakness of PRISM. The ATOP tool, in fact, includes the possibility to perform some kind of parametric analysis that at present is not fully supported in PRISM.

This paper is organized as follows. Section 2 presents the basic concepts of probabilistic model checking and PRISM. Section 3 illustrates the proposed MDD approach, while Section 4 provides the details of the approach for early quality assessment of service compositions. Section 5 describes the tool implementing our methodology and Section 6 describes how the proposed approach can be applied to a case study. Section 7 briefly surveys related work and Section 8 presents the conclusions.

2 Background

In this section we shortly review the basic concepts of the probabilistic model checking approach and the PRISM tool.

Probabilistic Model Checking is an automatic procedure for establishing if a desired property holds in a probabilistic system model. Conventional model

checkers start from a description of a model and a specification (using a state-transition system and a formula in some temporal logic, respectively) and return a boolean value, indicating whether or not the model satisfies the specification. In the case of probabilistic model checking, the models are probabilistic (typically, variants of Markov chains) and they add a probability to the transitions between states. In this way it is possible to calculate the likelihood of the occurrence of certain events during the execution of the system. This, in turn, allows quantitative analysis about the system, in addition to the qualitative statements made by conventional model checking. Probabilities are modeled via probabilistic operators that extend conventional (timed or untimed) temporal logic.

Probabilistic modeling is widely used in the field of performance evaluation; for example several algorithmic techniques and tools exist for Markovian models [12]. However, the key point of probabilistic model checking is the ability to combine probabilistic analysis and conventional model checking in a single tool. The first extension of model checking algorithms to probabilistic systems was proposed in the 1980s. However, work on implementation and tools did not begin until recently, when the field of model checking matured [24,25]. Probabilistic model checking draws on conventional model checking, since it relies on reachability analysis of the underlying transition system, but must also entail the calculation of the actual likelihoods through appropriate numerical methods, such as those employed in performance analysis tools [24,25].

PRISM is the model checker we selected to verify our models. PRISM [34] is a probabilistic model checker developed at the University of Birmingham. PRISM is a tool for the design and analysis of systems that exhibit probabilistic behaviors. It supports three types of probabilistic models: Discrete-Time Markov Chains, Markov Decision Processes, and Continuous-Time Markov Chains ([12]). Models are specified in a simple, high-level modeling language, which is a variant of the Reactive Modules formalism of Alur and Henzinger [3]. Properties are described by the PRISM property specification language, which is based on the two probabilistic temporal logics, called Probabilistic Computation Tree Logic (PCTL) [22] and Continuous Stochastic Logic (CSL) [7].

3 The ATOP Methodology

As introduced in Section 1, our approach derives quality predictions for service compositions. Each simple service of the composition is considered as a black-box entity. The process involved in this quality prediction analyzes abstract representations of service compositions to derive models suitable for applying probabilistic model checking techniques. Software architects may exploit this prediction to evaluate and compare different alternatives at design-time.

In Figure 1 we show a UML Activity Diagram (AD) outlining the main steps involved in the application of our methodology, by highlighting also who is in charge of them. Our approach starts from the application workflow specifications

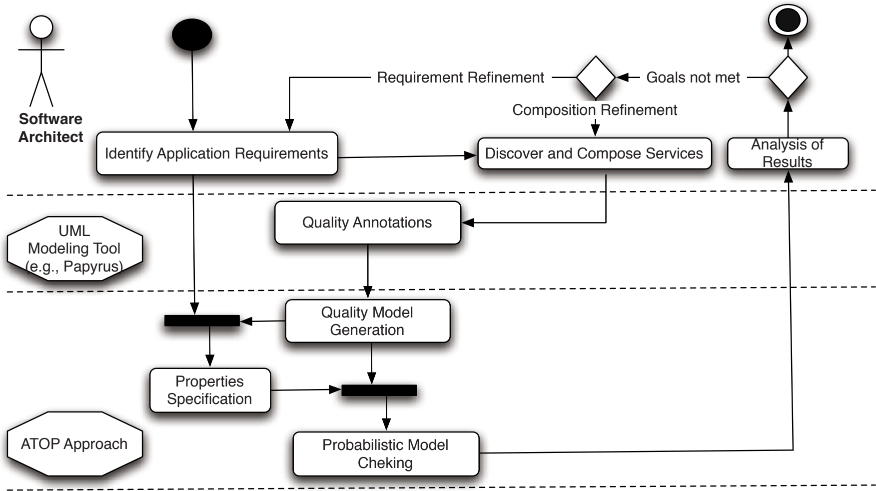


Fig. 1. The ATOP methodology

and derives quality predictions, such as application *Success probability* (an example of this quality prediction is illustrated in Section 6), through the following steps:

Identify application requirements. The software architect describes the application (s)he intends to realize and details its functional and non-functional requirements.

Discover and compose services. The software architect considers the services to compose only through their functional and non-functional annotations and builds the application through a service composition language. More precisely, our approach addresses the design phase of service compositions, which are represented through a UML AD [30]. At the implementation level, software architects exploit techniques like [14] for service discovery and workflow languages like BPEL [4] for service composition representation.

We assume that ADs representing service compositions are generated by using an ad-hoc tool, such as the UML *Papyrus* framework [32]. Our approach considers a subset of UML Activity Diagrams to represent sound service compositions. Supported diagrams are composed by only one *InitialNode* and only one *FinalNode*. Between these two elements there can be a sequence of the following elements: (i) activity, (ii) conditional block, and (iii) concurrent blocks, connected by means of arrows specifying the flow of execution.

Activity models invocation of a service. A conditional block is defined with a *DecisionNode* and a *MergeNode*. The conditional block appears in two different configurations, *If* or *Loop*, depending on the composition topology. Each branch of the decision block can contain a sequence of activities, decision

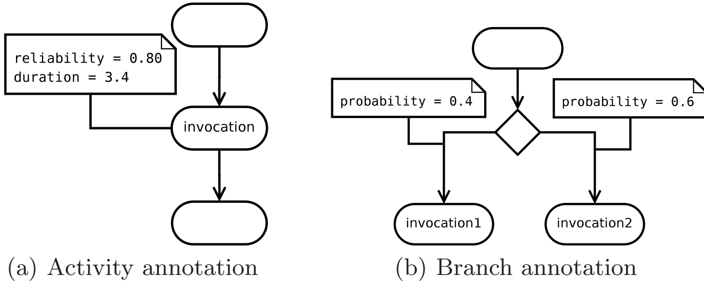
blocks and concurrent blocks. Concurrent blocks are defined by means of *ForkNodes* and *JoinNodes*. Each outgoing branch can contain the same aforementioned sequence of elements.

Quality annotations. Activity diagrams describing the composition are enriched by exploiting UML extensions. Through a UML profile, every ActivityNode is annotated with quality attributes of the selected service. Output arrows from a DecisionNode are annotated with the probability to follow each branch. Our approach exploits a subset of MARTE [31], a UML profile designed for specification of non-functional requirements of software systems and available in the Papyrus framework. We use the support of MARTE to represent a restricted subset of information; the main considered annotations are:

- *Service Reliability*: associated with an ActivityNode. It is a real number between 0 and 1 that represents the reliability of a single service invocation;
- *Service Execution Time*: associated with an ActivityNode. It represents the expected execution time of a service invocation.
- *Service Invocations Attempts*: associated with an ActivityNode. It represents the number of failed invocations necessary to declare a service to be faulty.
- *DecisionNode Output probabilities*: associated with output branches of a DecisionNode: they represent the probability to follow a given branch.
- *Service Degradation Function*: associated with an ActivityNode It is a domain-related law specifying dependency of service values from the execution context, e.g. size of input parameters. Due to its nature, this law can be inferred from observations or can be obtained from domain experts.

The Service Degradation Function is a peculiarity of the ATOP methodology. In particular, it supports parametric analysis. Indeed, this annotation describes the relation between a service composition input parameter and the quality properties of the basic services involved in the composition. For example, if a degradation function related to a parameter (e.g., input size) is specified for a service, during the model generation step all the values expressed by the other annotations in the service composition are updated by evaluating this degradation function. This mechanism is necessary to express the fact that reliability, execution time, invocation attempts, and branch probabilities are often dependent on specific service composition input parameters. Figure 2 illustrates two examples of non-functional annotations on ADs.

Quality models generation. A service composition, where model and annotations are described as presented before, is automatically translated into a quality model by the ATOP tool. The target quality model must be chosen according to the characteristics of the model and to the properties to be verified via the probabilistic model checker. ATOP considers the following Markovian models: (1) Discrete Time Markov Chains (DTMC),



(a) Activity annotation (b) Branch annotation

Fig. 2. Annotated Activity Diagrams

(2) Markov Decision Processes (MDP), and (3) Continuous Time Markov Chains (CTMC). A detailed description of this step is given in Section 4.

Properties specification. The set of properties to be verified on the model should be specified according to overall quality requirements. These properties are formulated through logic formulas expressed as CTL logic extensions.

Probabilistic Model Checking. The automatic modeling step generates a model, which is given as input to the probabilistic model checker. The model checker analyzes the received model with respect to the properties specified by the user.

Analysis of Results. The software architect analyzes the output produced by the probabilistic model checker to verify if the service composition matches the quality goals required by the application domain. If these goals are met, the development process continues to produce an implementation; otherwise, alternative compositions are evaluated in order to reach the required goals. A detailed example of the possible analysis is illustrated in Section 6.

4 Quality Modeling of Service Compositions

In this section we present the details of the quality model generation step of the ATOP methodology (illustrated in Figure 3). To this end, we provide a short description of (i) the target transformation models, (ii) the properties specifications and (iii) the translation process.

4.1 Target Models

The translation process is based on the exploration of the AD, starting from the *InitialNode* until the *FinalNode*. Depending on the nature of the model and on the type of analysis to be performed, different Markovian models can be chosen as output of the translation process.

In our framework, the DTMC model is used to model simple service compositions without concurrent branches and without timing information associated

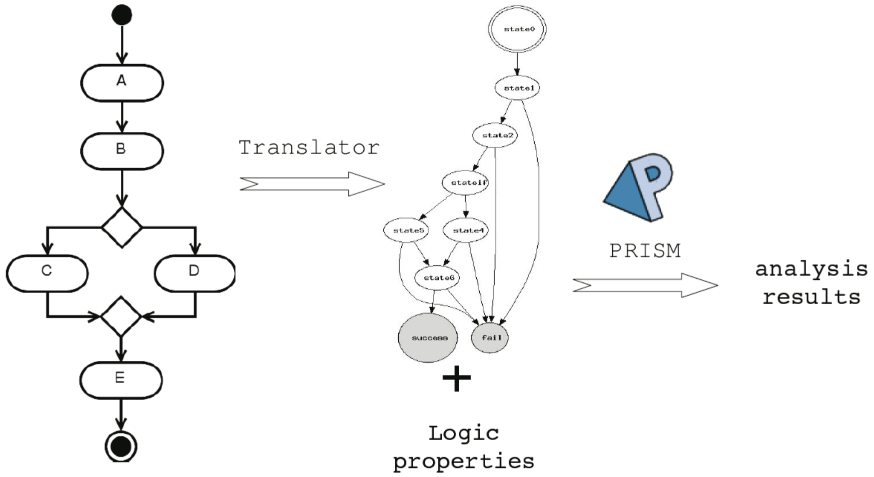


Fig. 3. Service composition analysis

with services. Should service compositions include concurrent sections (where service invocations are executed in parallel), it is necessary to model all the possible interleaved invocations. To this end, it is necessary to use a MDP model, which exploits non-determinism modelling all the paths the system could follow.

If the analysis focuses on the time necessary for the system to perform its functionalities, a CTMC model is instead required. By modeling the transition probability as an exponential distribution, each service invocation can be represented as a state whose transition parameter is related to the expected duration of the service execution. Using a parameter λ representing the rate of the exponential distribution and defining it as $1/expected_duration$ the model approximates the real temporal behavior of the system, giving a time-depending probabilistic result. The system is characterized by an initial transient phase and finally probability values asymptotically stabilize.

4.2 Properties Specifications

We analyze the model by verifying properties specified in temporal logic and evaluated through model checking. Basic properties on a service composition can be the reliability value of the whole complex system (e.g., the probability that starting from the initial state the system eventually reaches the success state), specified in PCTL as

$$P[F(system_state = success)]$$

where Ff (eventually operator) represents the short form of $true$ U f (U is the "until" temporal operator).

Similar properties can be evaluated starting from each state of the system

$$system_state = \text{"a certain service invocation"} \Rightarrow P[F(system_state = success)]$$

The evaluation of these properties support the discovery of configurations that can be critical for the system. Properties can also be specified to obtain a boolean result. Indeed, we can also express properties like

$$P_{\geq threshold}[F(system_state = success)]$$

whose evaluation yields a boolean value (true if the probability result complies with the threshold bound). Depending on the desired analysis, different logic properties can be formulated over the model and then submitted to the model checker.

4.3 Translation Elements

The translation process from ADs to Markov models is based on the exploration of the original model, starting from the *InitialNode* along the execution path defined by the control flow. In the following we describe how the translation of the main AD elements is performed. The tool implementing the translation is described in Section 5. The *Initial* and *Final* nodes of the AD correspond to the initial and final states of the Markov model.

The *Activity* is the core element of an AD that describes a service invocation and can contain additional information through annotations. An Activity is translated into a node with two outgoing transitions: the success transition and the failure transition. The probabilities associated with the two transitions depend on the annotations of the original diagram; the destination states can be the next state, in case of success, and a retry or a fail state, in case of failure.

Decision and *Merge* nodes in ADs decision blocks can assume the *If* and *Loop* configurations, depending on the topology of the components. In the former case, the translation process creates in the Markovian model a new node representing the *If* node and as many transitions as there are outgoing branches. The exploration continues for each branch over the path, until the MergeNode is reached. Then the exploration resumes from the node following the MergeNode. In the latter case (*Loop*), the exploration is performed over the loopback arc until the MergeNode is reached. Then the exploration continues over the remaining branch exiting the *Loop* node.

Fork and *Join* nodes define a concurrent block. The translation requires the exploration of all the branches exiting the ForkNode, until a JoinNode is reached. Each branch is modeled by an independent sub-diagram. The exploration resumes starting from the node following the JoinNode.

Additional information on the model define non-functional properties. They are used to drive the creation of the probabilistic model, as described hereafter:

- Service Reliability: the value α of reliability is used as the probability of the success transition. The related fail transition has the probability $1 - \alpha$.
- Service Execution Time: in a CTMC model, the rate of the exponential distribution λ is defined as $1/expected_duration$
- Service Invocations Attempts. The service can be invoked a given numbers of time before being declared as failed. Therefore, the fail state is reached only when the last attempt is performed and fails.

The translation process is realized in the ATOP tool described in the next section.

5 The ATOP Tool

As described in Section 3, the ATOP approach takes as input a formal representation of the service composition drawn as a UML AD extended with quality annotations. This representation is exported in the XMI format, elaborated by the ATOP tool and translated into a PRISM model. Note that, although the XMI format is near to be a standard, each tool representation differs for small details, which implies that a single interpreter is not valid in each case. The translator tool builds a graph-based representation obtained by means of a tool-specific interpreter. In this way the translator can be easily extended to support new design tool just adding an extension tailored on the new XMI format. The translation process is based on the recursive exploration of the graph, performing the operations described in Section 4 to generate an output model on the basis of the information provided by the AD. This tool can be executed directly form the command line or through a graphical user interface. The latter offers an aid to select the input and output format options and shows translation results.

In several contexts, service quality needs to be evaluated depending on some execution parameters. At present the PRISM model checker does not offer full support for parametric analysis of the model; the ATOP tool overcomes such limitation generating models whose values are set depending on given parameters. Using ATOP it is possible to specify the execution context parameters (e.g., input size) based on which, if a degradation law is defined for services components, the non-functional values are adjusted. In this way the system behaviour can be automatically analyzed in different contexts. The model is then evaluated by means of automatic analysis techniques, via PRISM.

6 Case Study

In this section, we illustrate our approach through a case study of a service composition. The system resulting from this composition offers a travel management functionality. Starting from travel location, it offers booking services and notifications. The high level composition specification is shown in Figure 4 through an extended AD. The workflow initially performs three service invocations, to identify the requirements of the travel. Then two concurrent task are executed: a parking booking and a process of notification to the user. After execution of both, the service performs a meeting arrangement and subsequently notifies the commitment for the travel. The activities in the diagram represent service invocations, annotated with a reliability value and an execution time expressed in seconds. This information is obtained from the service providers. The diagram offers a view in which single elements are organized to offer a more complex system. The concrete implementation is a BPEL xml file (although any other equivalent orchestration language could be adopted).

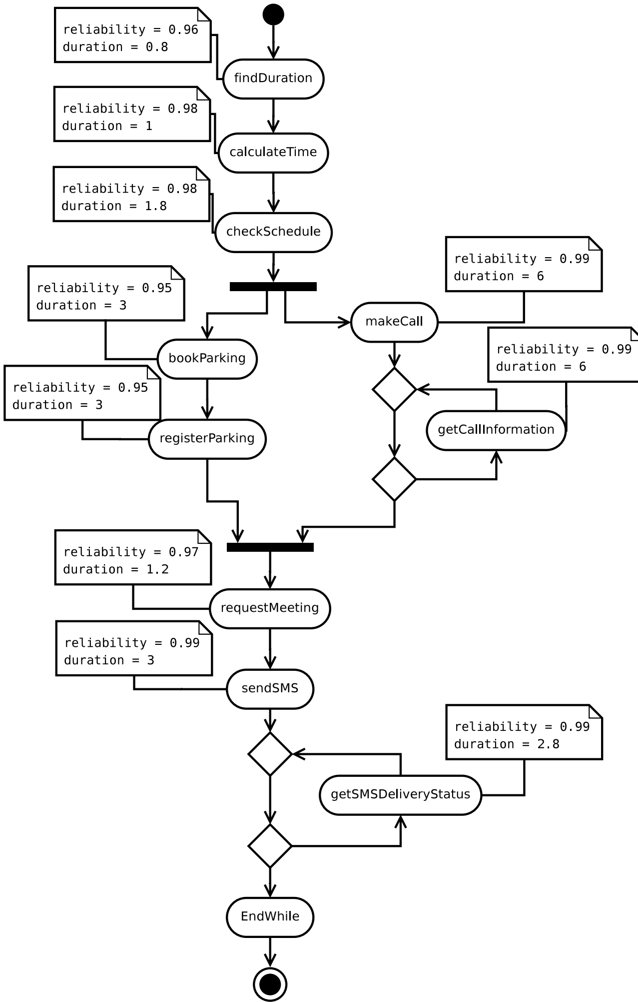


Fig. 4. Travel Management Service Activity

The AD represented by an XMI file can be translated in three different Markovian models, as discussed in Section 4. The first model (DTMC) does not take into account time and cannot model concurrency. The concurrent branches in the AD of Figure 4 are automatically condensed into a single activity[1]. DTMC supports the evaluation of the expected reliability value concerning the whole service composition or the expected reliability value concerning part of it (i.e., starting from an inner state, different from the initial state). The property associated with the global reliability of the system is expressed in PCTL as:

[1] This step is carried out by PRISM generating, with equal probabilities, all the possible paths composed by interleaved activities. The paths are then evaluated and the results of the analysis are aggregated in a single node.

$$P[F(system_state = success)].$$

The probabilistic model checker returns a probability of `0.775`, which represents the reliability of the service composition. The same property can be locally evaluated starting from any specific inner state. For example, the table below shows the reliability values obtained starting from two inner states.

state	PCTL formula	result
`checkSchedule`	$state = checkSchedule \Rightarrow P[F(system_state = success)]$	`0.824`
`requestMeeting`	$state = requestMeeting \Rightarrow P[F(system_state = success)]$	`0.950`

The second kind of model (MDP), supports the analysis of concurrent executions. More precisely, it is possible to compute the reliability values associated with internal states of concurrent areas of the diagram. In presence of concurrent branches the order in which elements of different branches are interleaved cannot be predicted. By using MDP, it is possible to compute reliability values associated with internal states of concurrent sections of the diagram, which would otherwise be impossible using DTMC. Considering all the possible non deterministic evolutions of the system, the model checker can return the upper and lower bounds of reliability. The table below reports the maximum and minimum reliability values obtained from a concurrent state of the composition.

state	PCTL formula	result
`makeCall`	$state = makeCall \Rightarrow Pmax[F(system_state = success)]$	`0.932`
`makeCall`	$state = makeCall \Rightarrow Pmin[F(system_state = success)]$	`0.841`

The third type of model generated is CTMC, which focuses on the time associated with every service invocation. This analysis shows how the reliability of the system changes with respect to the time. This result approximately indicates a time bound for global service execution. The composition illustrated in this case study contains a concurrent block and, consequently, we cannot directly adopt a CTMC model. The analysis is performed by substituting the concurrent block with a synthesis node containing results of probabilistic model checking related to each concurrent branch. The properties checked are similar to the properties obtained with a DTMC model, but with an indication of the time dependency.

Figure 5 shows the probability value of reaching the success state within time t, computed for t ranging in an interval $[0, 70]$ (seconds). This represents how the probability of success evolves over time after the invocation of the composed service. This value tends in the long run to the value obtained with the DTMC model (`0.775`), the reliability value of the service.

The CSL specification of this property is

$$P[true\ \mathrm{U}^{[0,t]} system_state = success]$$

and its evaluation is obtained by varying parameter t that represents the time. This example does not model any recovery behavior, each service is invoked just once. The reliability of the whole system could be improved, by increasing the expected execution time, modeling for each service the number of retries for failure invocations.

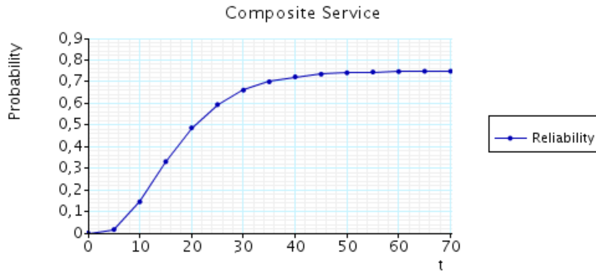


Fig. 5. Success probability evolution

7 Related Work

In the last years, Quality of Service (QoS) has been extensively studied in the context of *traditional software systems*. In particular, there has been a great interest in model transformation methodologies for the generation of analysis-oriented target models (including performance and reliability models) starting from design-oriented source models, possibly augmented with suitable annotations. In particular, several proposals have been presented concerning the direct generation of performance analysis models. Each of these proposals focuses on a particular type of source design-oriented model and a particular type of target analysis-oriented model, with the former spanning UML, Message Sequence Chart, Use Case Maps, formal language as AEmilia, ADL languages as Acme, and the latter spanning Petri nets, queueing networks, layered queueing network, stochastic process algebras, Markov processes (see [8] for a thorough overview of these proposals and [1] for recent proposals on this topic). Some proposals have also been presented for the generation of reliability models. All the proposals we are aware of start from UML models with proper annotations, and generate reliability models such as fault trees, Markov processes and Bayesian models (see [21] for more details). Moreover, some attempts have been made in the literature [2,19,20] to translate UML specifications into models to be solved by probabilistic model checkers. Specifically, in [2] it is proposed a methodology that translates UML statecharts (with annotations for real-time systems) into probabilistic timed automata that are then solved using PRISM. Gilmore et al. in [19,20] propose a method translating UML statecharts into stochastic process algebra (PEPA) models that are then solved using PRISM.

Recently, QoS issues in service selection and composition have obtained great interest in the *service research community*. Different approaches have been followed so far, spanning the use of QoS ontologies [28], the definition of ad-hoc methods in QoS-aware frameworks [33,38], and the application of optimization algorithms [5,13,39].

One of the first works in this area is proposed in [29] where a framework for composed services modeling and QoS evaluation is presented. A composite service is modeled as a directed weighted graph where each node corresponds to a Web Service (WS) and edge weights represent the transition probabilities of

two subsequent tasks. The author shows how to evaluate quality of service of a composed service from basic services characteristics and graph topology.

Some recent proposals face the problem of composition of WSs by implementing genetic algorithms. In Canfora et al. [13] the reduction formulas presented in [16] are adopted, and the problem is also periodically re-optimized in order to take into account WS performance variability. However, only sub-optimal solutions are identified since WSs specified inside execution loops are always assigned to the same Web service implementation. The paper [17] proposes a mechanism that implements an optimizing WS composition combining performance optimization, price optimization, and payload optimization when meeting the requirements of Service-level Agreement (SLA). In [35] the authors propose both an evaluation approach for QoS attributes of WS, which is completely service and provider independent, and a method to analyze WS interactions and extract important QoS information without any knowledge about the service implementation. In [37] the WS composition from a performance viewpoint is studied and measured. This measurements demonstrate that WS composition may reduce the maximal load of a system drastically (i.e., quasi-exponentially with the number of service compositions). In order to mitigate this performance reduction, the author proposes an optimized service composition architecture as a solution.

The works closest to ours concern methods to derive performance related measures of workflow processes [15,27,36]. Cardoso [15] proposes two different metrics to evaluate the control-flow complexity of BPEL web processes before their actual implementation. In [36] a mathematical model based on operations research techniques is proposed to estimate the influence of the execution of orchestrated processes on utilization and throughput of the system. In [27] starting from annotated BPEL and WSDL specifications, performance bounds on response time and throughput are derived. In such a way users are able to assess the efficiency of a BPEL workflow, while service provider(s) can perform sizing studies or estimate performance gains of alternative upgrades to existing systems.

A related approach to verifying service-oriented architectures is described in [9]. This work deals with new added-value services obtained by composing existing services through workflows described in the BPEL language. The workflows are verified at design time via model checking. The properties against which designs are checked are then transformed into run-time monitored assertions to support run-time verification. Properties are expressed in a temporal logic language (ALBERT), which can express both functional and non-functional properties, such as expected response time. Properties are of two kinds: Assumed Assertions (AAs), which state the expected behavior of the external services invoked by the workflow, and Guaranteed Assertions (GAs), which state the properties the workflow is expected to ensure to its users. Model checking is used to prove that GAs are satisfied, assuming that AAs hold. The approach explored in this work, however, does not support any kind of probabilistic reasoning. It is based on the use of the Bogor model checker [18].

With respect to existing work, our approach represents a first attempt at a design methodology (and supporting tools) through which QoS properties may be specified and formally analyzed for service compositions.

8 Conclusions

In this paper we have presented ATOP: a design methodology (and supporting tools) through which QoS properties may be specified and formally analyzed for service compositions. In particular, QoS reasoning is based on probabilistic modelling, which is crucial for performance and reliability prediction. Our approach is supported by a tool that translates a high-level design description into a form that is amenable to probabilistic model checking using PRISM. Our initial assessment of the approach has been quite encouraging, and a further development of case studies will be part of our future activities.

Our future work will also focus on systematizing the feedback loop from run-time observations of the performance attributes of a running composite service back to the design environment. If the running system is found to behave inconsistently with respect to the design model, it would be desirable to re-calibrate the design model with more accurate data drawn from run-time gathered information, and possibly derive an improved implementation. This of course requires that models should be kept alive at run time. It also requires ways to perform analyses at run time in an efficient and light-weight manner, possibly incrementally. To the best of our knowledge, this is still an unexplored research path.

Acknowledgments

This work has been partially supported by Project Q-ImPrESS (FP7-215013) funded under the European Union's Seventh Framework Programme (FP7).

References

1. Wosp : Proceedings of the international workshop on software and performance, (1998-2007)
2. Addouche, N., Antoine, C., Montmain, J.: Combining extended uml models and formal methods to analyze real-time systems. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688. pp. 24–36. Springer, Heidelberg (2005)
3. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design: An International Journal 15(1), 7–48 (1999)
4. Alves, A.: et al. Web service business process execution language version 2.0. Committee Draft, 17 (May 2006)
5. Ardagna, D., Pernici, B.: Global and Local QoS Guarantee in Web Service Selection. In: Proc. of Business Process Management Workshops, pp. 32–46 (2005)
6. Atkinson, C., Kuhne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (2003)

7. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time markov chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
8. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Software Eng. 30(5), 295–310 (2004)
9. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Software 1(6), 219–232 (2007)
10. Baresi, L., Gerosa, G., Ghezzi, C., Mottola, L.: Playing with time in publish-subscribe using a domain-specific model checker. In: SAVCBS 2007: Proceedings of the 2007 conference on Specification and verification of component-based systems, pp. 55–62. ACM, New York (2007)
11. Baresi, L., Ghezzi, C., Mottola, L.: On accurate automatic verification of publish-subscribe architectures. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, pp. 199–208. IEEE Computer Society, Los Alamitos (2007)
12. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.: Queuing Network and Markov Chains. John Wiley, Chichester (1998)
13. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An Approach for QoS-aware Service Composition Based on Genetic Algorithms. In: Proc. of Genetic and Computation Conf. Washington, DC, pp. 1069–1075 (June 2005)
14. Cardellini, V., Casalicchio, E., Grassi, V., Mirandola, R.: A framework for optimal service selection in broker-based architectures with multiple QoS classes. In: Services computing workshops, SCW 2006, pp. 105–112. IEEE Computer Society, Los Alamitos (2006)
15. Cardoso, J.: Complexity analysis of BPEL web processes. Software Process: Improvement and Practice 12(1), 35–49 (2007)
16. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. J. Web Sem. 1(3), 281–308 (2004)
17. Dong, W.L., YU., H.: Optimizing web service composition based on qos negotiation. EDOCW 0, 46 (2006)
18. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby,: Building your own software model checker using the bogor extensible model checking framework. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 148–152. Springer, Heidelberg (2005)
19. Gilmore, S., Haenel, V., Kloul, L., Maidl, M.: Choreographing security and performance analysis for web services. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670. pp. 200–214. Springer, Heidelberg (2005)
20. Gilmore, S., Kloul, L.: A unified tool for performance modelling and prediction. Reliability Engineering and System Safety 89, 17–32 (2005)
21. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. Journal of Systems and Software 80(4), 528–558 (2007)
22. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
23. He, F., Baresi, L., Ghezzi, C., Spoletini, P.: Formal analysis of publish-subscribe systems by probabilistic timed automata. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 247–262. Springer, Heidelberg (2007)
24. Kwiatkowska, M.: Quantitative verification: Models, techniques and tools. In: Proc. 6th joint meeting of the European Software Engineering Conference and the ACM

SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 449–458. ACM Press, New York (2007)

25. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
26. Di Marco, A., Mirandola, R.: Model transformation in software performance engineering. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 95–110. Springer, Heidelberg (2006)
27. Marzolla, M., Mirandola, R.: Performance prediction of web service workflows. In: Overhage, S., Szyperski, C.A., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880. Springer, Heidelberg (2008)
28. Maximilien, E.M., Singh, M.P.: A Framework and Ontology for Dynamic Web Services Selection. IEEE Internet Computing 8(5), 84–93 (2004)
29. Menasce, D.A.: QoS Issues in Web Services. IEEE Internet Computing 6(6), 72–75 (2002)
30. Object Management Group. UML 2.0 superstructure specification (2002)
31. Object Management Group OMG. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems. ptc/07-08-04 (2007)
32. Papyrus UML, http://www.papyrusuml.org/
33. Patel, C., Supekar, K., Lee, Y.: A QoS Oriented Framework for Adaptive Management of Web Service Based Workflows. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 826–835. Springer, Heidelberg (2003)
34. PRISM, Probabilistic Model Checker, http://www.prismmodelchecker.org/
35. Rosenberg, F., Platzer, C., Dustdar, S.: Bootstrapping performance and dependability attributes of web services. ICWS 0, 205–212 (2006)
36. Rud, D., Schmietendorf, A., Dumke, R.: Performance modeling of WS-BPEL-based web service compositions. SCW 0, 140–147 (2006)
37. Schmid, H.A.: Service congestion: The problem, and an optimized service composition architecture as a solution. ICWS 0, 505–514 (2006)
38. Yu, T., Lin, K.J.: A Broker-Based Framework for QoS-Aware Web Service Composition. In: Proc. of 2005 IEEE Int'l Conf. on e-Technology, e-Commerce and e-Service (March 2005)
39. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Trans. Softw. Eng. 30(5), 311–327 (2004)

Model-Driven Performance Analysis

Gabriel A. Moreno and Paulo Merson

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{gmoreno,pfm}@sei.cmu.edu

Abstract. Model-Driven Engineering (MDE) is an approach to develop software systems by creating models and applying automated transformations to them to ultimately generate the implementation for a target platform. Although the main focus of MDE is on the generation of code, it is also necessary to support the analysis of the designs with respect to quality attributes such as performance. To complement the model-to-implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. This paper describes an approach to model-driven analysis based on reasoning frameworks. In particular, it describes a performance reasoning framework that can transform a design into a model suitable for analysis of real-time performance properties with different evaluation procedures including rate monotonic analysis and simulation. The concepts presented in this paper have been implemented in the PACC Starter Kit, a development environment that supports code generation and analysis from the same models.

1 Introduction

Model-Driven Engineering (MDE) is an approach to create software systems that involves creating models and applying automated transformations to them. The models are expressed in modeling languages (e.g., UML) that describe the structure and behavior of the system. MDE tools successively apply pre-defined transformations to the input model created by the developer and ultimately generate as output the source code for the application. MDE tools typically impose domain-specific constraints and generate output that maps onto specific middleware platforms and frameworks [1]. MDE is often indistinctively associated to OMG's Model-Driven Architecture and Model-Driven Development.

The ability to create a software design and apply automated transformations to generate the implementation helps to avoid the complexity of today's implementation platforms, component technologies and frameworks. Many MDE solutions focus on the generation of code that partially or entirely implements the functional requirements. However, these solutions often overlook runtime quality attribute requirements, such as performance or reliability. Fixing quality attribute problems once the implementation is in place has a high cost and often requires structural changes and refactoring. Avoiding these problems is the main

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 135–151, 2008.
© Carnegie Mellon University 2008

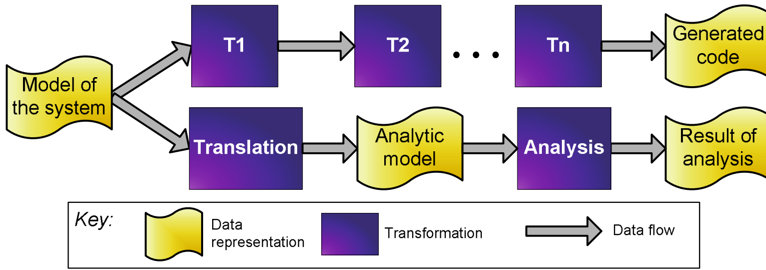


Fig. 1. Model-Driven Engineering and Model-Driven Analysis

motivation to perform analysis early in the design process. To complement the model-to-implementation path of MDE approaches, an MDE tool infrastructure should provide what we call model-driven analysis. The model to code path and the model-driven analysis path are notionally represented in Figure 1. The goal of model-driven analysis is to verify the ability of the input design model to meet quality requirements.

The Software Engineering Institute has developed an MDE tool infrastructure that offers code generation along with various analytic capabilities. This tool infrastructure is called the PACC Starter Kit (PSK) [2]. It uses the concept of reasoning frameworks [3] to implement analytic capabilities. Several reasoning frameworks have been developed and applied to industry problems. This paper focuses on our performance reasoning framework, which analyzes timing properties of component-based real-time systems. The paper describes the model-driven approach used in the implementation of the reasoning framework.

The remainder of the paper is organized as follows. Section 2 introduces the concept of a reasoning framework and then describes the elements of our performance reasoning framework. Section 3 describes the intermediate constructive model, which is the first model created when analyzing an input design. Section 4 explains the performance model and how it is generated from the intermediate constructive model through a transformation called interpretation. Section 5 briefly describes how the performance model is used by different evaluation procedures to generate performance predictions. Section 6 shows an example of the application of the concepts described in the paper. Section 7 discusses related work and Section 8 has our concluding remarks.

2 Performance Reasoning Framework

A reasoning framework provides the ability to reason about a specific quality attribute property of a system's design [3]. Figure 2 shows the basic elements of a reasoning framework. The input is an architecture description of the system, which consists of structural and behavior information expressed in a modeling language or any formally defined design language. Reasoning frameworks cannot analyze any arbitrary design. Furthermore, there is a tradeoff between the analytic power of a reasoning framework and the space of designs it can analyze.

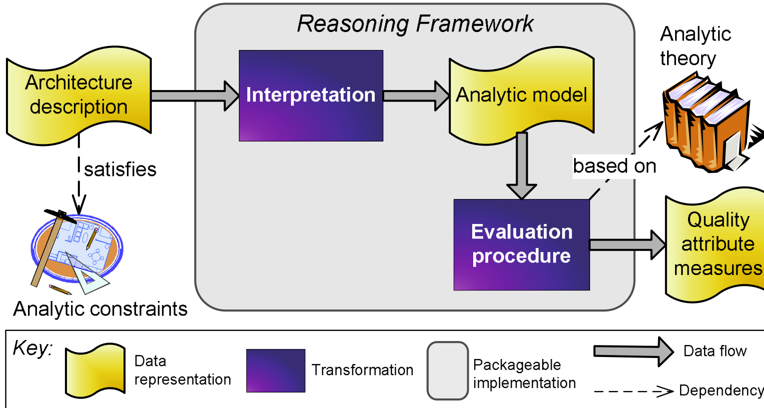


Fig. 2. Basic elements of a generic reasoning framework

Reasoning frameworks restrict that space by imposing analytic constraints on the input architecture description. The analytic constraints restrict the design in different ways (e.g., topological constraints) and also specify what properties of elements and relations are required in the design.

The architecture description is submitted to a transformation called interpretation. If the architecture description is well-formed with respect to the constraints and hence analyzable, the interpretation generates an analytic model representation. This model is an abstraction of the system capturing only the information relevant to the analysis being performed. The types of elements, relations and properties found in an analytic model are specific to each reasoning framework. The analytic model is the input to an evaluation procedure, which is a computable algorithm implemented based on a sound analytic theory. The implementation of the evaluation procedure may be purely analytic, simulation-based or a combination of both.

Figure 3 shows the basic elements of our performance reasoning framework. Comparing to Figure 2, we see an additional step that translates the architecture description to a data representation called ICM. ICM stands for intermediate constructive model and it is a simplified version of the system's original design. The ICM is described in Section 3. The analytic model seen in Figure 2 corresponds to the performance model in Figure 3, which will be described in Section 4. The evaluation procedure box in Figure 2 corresponds to the performance analysis box in Figure 3, which is carried out by analytic and simulation-based predictors created based on rate monotonic analysis (RMA) [4] and queuing theory [5]. As described in Section 5, the predictors can estimate average and worst-case latency of concurrent tasks in a system that runs on a fixed priority scheduling environment.

The performance reasoning framework is packaged and independently deployed as an Eclipse plug-in [6]. Thus any Eclipsed-based tool used to model software systems in a parseable design notation could benefit from the performance reasoning

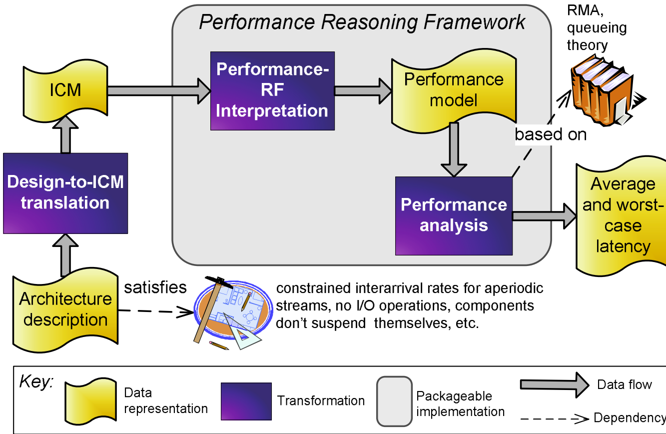


Fig. 3. Performance Reasoning Framework

framework by exporting their designs to ICM. Examples of such tools include: IBM Rational Software Architect, OSATE tool for the AADL language, Eclipse Model Development Tools (MDT) UML2, and AcmeStudio. When adding the performance reasoning framework plug-in to a modeling tool, the only thing one has to do is to create a class that implements the following method:

```
AssemblyInstance translateDesignToIcm(IFile designFile);
```

As expected, the implementation of this method is entirely specific to the design language representation used by the tool. We have implemented this method for the CCL design language [7] that is used in the PSK. An explanation of CCL or how to translate it to ICM is beyond the scope of this paper. But we should note that the design language used as input for the performance reasoning framework shall support the following elements and relations:

– Components with input and output ports[1]. If the design language were UML, for example, UML components with required and provided interfaces could be used.
– Special components called environment services (or simply services) that represent external elements of the runtime environment that the system interacts with. Clocks, keyboards, consoles, and network interfaces are examples of services. Like regular components, services may have sink and source pins. In UML, they could be represented as stereotyped UML components.
– A way to wire the components together, that is, to connect a source pin to a sink pin. In UML, a simple UML assembly connector could be used.
– A way to differentiate between synchronous and asynchronous interactions, as well as threaded and unthreaded interactions. In UML, stereotypes could be used to indicate these characteristics.

[1] In CCL, an input port is called "sink pin" and an output port is called "source pin"—these are the terms used in this paper.

– A way to annotate any element with specific pairs of key and value. These annotations are used for properties required by the reasoning framework (e.g., the performance reasoning framework requires the priority of each component to be specified). In UML, annotations could be done with tagged values.

3 Intermediate Constructive Model

The architecture description that is the input to a reasoning framework (see Figure 2) is itself a constructive model of the system. However, it often contains many details that are not used in the performance analysis. For example, the state machine of a component may be used for code generation but is not needed by the performance reasoning framework. To remove details specific to the input design language and hence simplify the interpretation translation, we created the intermediate constructive model (ICM). The design-to-ICM translation (see Figure 3) abstracts the elements of the architecture description that are relevant to performance analysis to create the ICM.

Figure 4 shows the ICM metamodel. A complete description of the elements, properties and associations in the ICM metamodel is beyond the objectives of this paper. Here, we present the information pieces that are key to the performance analysis discussion in subsequent sections. The entry point to navigate an ICM is *AssemblyInstance*, which is the return type of the `translateDesignToIcm` method mentioned earlier. The system being analyzed is an assembly of components. In the ICM metamodel, a component is generically called *ElementInstance*. The use of the suffix "instance" avoids confusion when the input design language allows the definition of *types* of components. For example, in CCL one can define a component type called *AxisController*. Then a given assembly may contain two components (e.g., *axisX* and *axisY*) that are instances of that component type. The ICM for that assembly would show two *ElementInstance* objects with the same type (*AxisController*) but different names.

An *ElementInstance* object (i.e., a component) has zero or more pins (*PinInstance* objects in the metamodel). Each pin is either a *SinkPinInstance* or a *SourcePinInstance* object. A sink pin is an input port and when it is activated it performs some computation. The performance analysis is oblivious to most details of that computation. However, it is necessary to know what source pins (output ports) are triggered during the computation and in what order—that is given by the *reactSources* association between *SinkPinInstance* and *SourcePinInstance*. It is also necessary to specify the priority that the sink pin computation will have at runtime. Another important property of a sink pin is the execution time for the corresponding computation, which is shown in the ICM metamodel as the *execTimeDistribution* association between *SinkPinInstance* and *Distribution*. A sink pin can be synchronous or asynchronous. If it is synchronous, it may allow concurrent invocations (reentrant code) or enforce mutual exclusion on the sink pin's computation.

Real-time systems, especially ones with aperiodic threads, sometimes exhibit different behavior depending on certain conditions of the environment or different pre-set configurations. For example, a system can be operating with limited

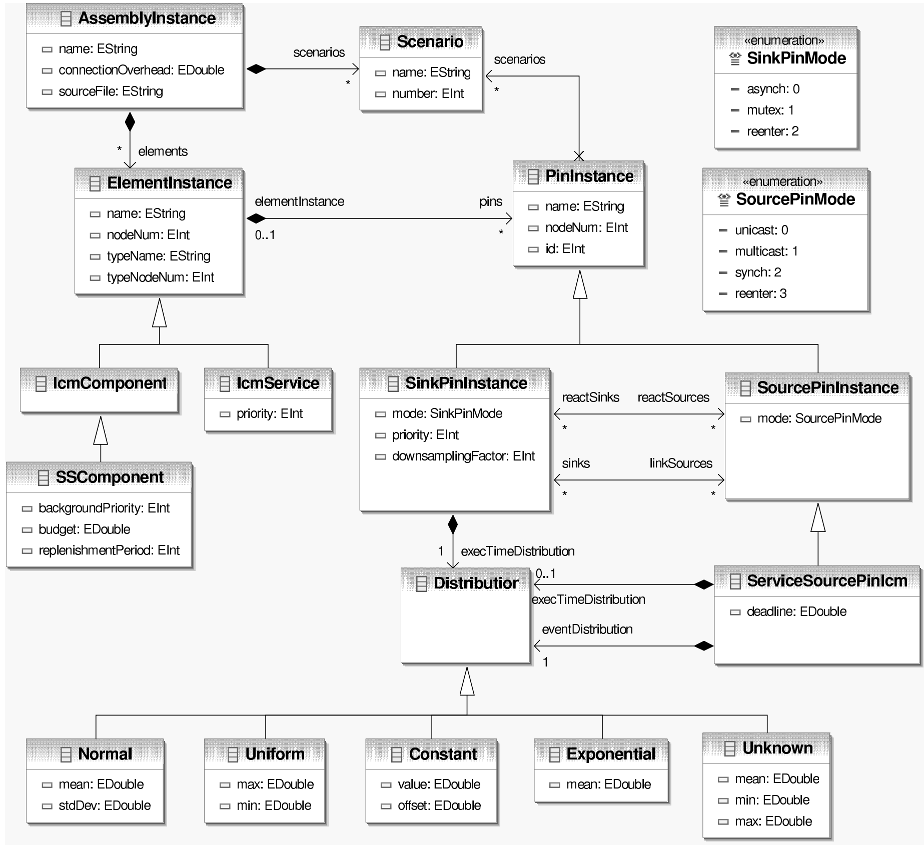


Fig. 4. ICM metamodel (notation: UML)

capacity due to a failure condition, or a system can have optional "pluggable" components. These kinds of variability are represented in the ICM as *Scenario* objects. A scenario can be represented in the architecture description as annotations to the assembly and the sink pins that are active under that scenario.

Once the ICM for a given architecture description is created, the performance reasoning framework can perform the interpretation translation to generate the performance model, which is used as input to the performance analysis. These steps are described in the following sections.

4 Performance Model Generation

An important component of a reasoning framework is the interpretation process that transforms an architecture description into an analytic model. Section 2 showed that the architecture description is translated to an intermediate representation (ICM) in our performance reasoning framework. In this reasoning

framework, interpretation starts with an ICM model and produces a performance model that can then be analyzed by different evaluation procedures.

4.1 Performance Metamodel

The performance metamodel (Figure 5) is based on the method for analyzing the schedulability of tasks with varying priority developed by Gonzalez Harbour et al. [8]. In this method, a task is a unit of concurrency such as a process or a thread. Tasks are decomposed into a sequence of serially executed subtasks, each of which has a specific priority level and execution time.

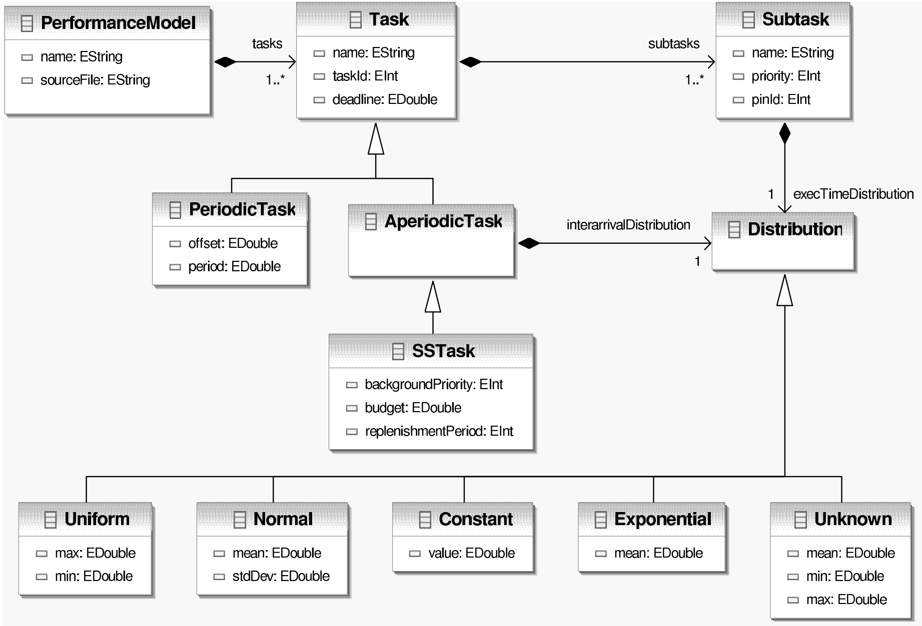


Fig. 5. Performance metamodel (notation: UML)

The root element of a performance model in our reasoning framework is *PerformanceModel*, which contains one or more *Tasks*. Unlike in the method by Gonzalez Habour et al., where all tasks are periodic, in our metamodel a task is either a *PeriodicTask* or an *AperiodicTask*. Periodic tasks are characterized by a period and an offset that is used to model different task phasings at startup. Aperiodic tasks, on the other hand, are tasks that respond to events that do not have a periodic nature. For that reason, they have an *interarrivalDistribution* to describe the event arrival distribution. For example, the events may follow an exponential distribution where the mean interarrival interval is 10ms. The *SSTask* models aperiodic tasks scheduled using the sporadic server algorithm [9], which allows scheduling aperiodic tasks at a given priority while limiting their impact on the schedulability of other tasks.

Tasks do not have an explicit execution time attribute because their computation is carried out by the subtasks they contain. Thus, each *Subtask* has an execution time and a priority level. The metamodel supports both constant and random execution times by providing different kinds of *Distribution*. The task is the unit of concurrency. That is, tasks execute in parallel—subject to a fixed priority scheduling policy—and within a task there is no concurrency.

4.2 From ICM to Performance Model

As depicted in Figure 2, the interpretation that generates the analysis model can only be carried out if the analytic constraints of the reasoning framework are satisfied by the design. The assumptions and analytic constraints of the performance reasoning framework are the following.

1. The application being analyzed executes in a single processing unit (i.e., in one single-core CPU or in one core of a multi-core CPU).
2. The runtime environment uses fixed-priority preemptive scheduling.
3. Components perform their computation first and then interact with other components.
4. Each sink pin in a component reacts with all the source pins in the reaction.
5. No two subtasks (or equivalently, sink pins) within a response can be ready to execute at the same time with the same priority level.
6. Priority of mutex sink pins is assigned according to the highest locker protocol.
7. Components do not suspend themselves during their execution.

Even though some of these constraints may seem too restrictive, they are the result of a process called co-refinement [10], a process that evaluates the tradeoffs between constraints imposed on the developers, the cost of applying the technology, and the accuracy of the resulting predictions. For example, constraint 3 makes the interpretation simpler because it does not require looking into the state diagram of the component, and also makes the use of the technology simpler because it requires fewer annotations to be provided by the developer.

The ICM and the performance model are different in several aspects. For example, the ICM can model a complicated network of computational elements, while the performance model only supports seemingly isolated sequences of them. The rest of this section describes the concepts guiding the transformation from ICM to performance model.

An event is an occurrence the system has to respond to. The tick of an internal clock and the arrival of a data packet are examples of events. A response is the computational work that must be carried out upon the arrival of an event [4]. The main goal of the performance reasoning framework is to predict the latency of the response to an event, taking into account the preemption and blocking effects of other tasks. In an ICM, a source of events is represented as a source pin in a service (i.e., a *ServiceSourcePinIcm*). Therefore, the goal translates into predicting the latency of all the components that are connected directly or indirectly to that service source pin. Since the response to an event is modeled as

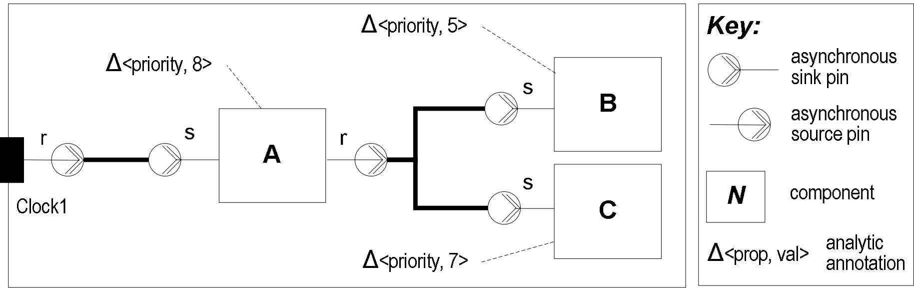


Fig. 6. Response with concurrency

a task in the performance model, it follows that for each *ServiceSourcePinIcm* in the ICM, a *Task* needs to be created. Depending on the event interarrival distribution of the service source pin, the task will be a *PeriodicTask* or an *AperiodicTask*.

The next step, and the most complex one, is transforming a possibly multi-threaded response involving several components into a sequence of serially executed subtasks. This transformation deals with two main issues: the internal concurrency within a response, and the blocking effects between responses.

Concurrency within a Response. Figure 6 shows an example of a response with concurrency. Component *A* asynchronously activates components *B* and *C*. Since *B* and *C* can execute concurrently, it seems they cannot be serialized as a sequence of subtasks. However, because they have different priorities, they will actually execute serially,[2] first the high priority component *C* and then the low priority component *B*, even when they are ready to execute at the same time. Thus, it is possible to determine the sequence of subtasks that represents the actual execution pattern. In order to do that, the design has to satisfy one analytic constraint: *each threaded component has to be assigned a unique priority within the response*. This is a sufficient but not necessary constraint because two components can have the same priority as long as they are never ready to execute at the same time. The interpretation algorithm flags situations where priority level sharing is not allowed.

Blocking between Responses. In Figure 7 there are two responses that use a shared component. Since this component is not reentrant, the responses can potentially block each other. This blocking is addressed in the performance model by using the highest locker protocol [4]. The shared component is assigned a priority higher than the priority of all the components using it. In addition, the component must not suspend itself during its execution. The benefit of these analytic constraints is twofold. First, they make the behavior more predictable because calling components are blocked at most once and priority inversion is bounded. Second, the non-reentrant component can be modeled as a subtask in

[2] The analysis assumes the application runs in one processing unit (constraint 1). Therefore, threads that are logically concurrent are executed serially by the processor.

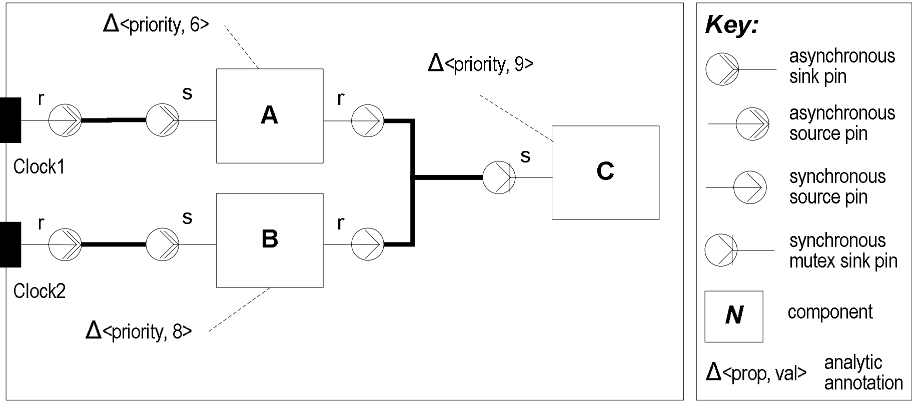


Fig. 7. Blocking between responses

each of the responses. There is no need to have special synchronization elements in the model because the highest locker protocol and the fixed-priority scheduling provide the necessary synchronization.

Interpretation is a model transformation that translates from the ICM metamodel to the performance metamodel. This transformation has been implemented both using a direct-manipulation approach with Java and the Eclipse Modeling Framework (EMF), as well as with the ATL model transformation language [11]. Details of both implementations are beyond the scope of this paper.

5 Performance Analysis

The performance model produced by interpretation can be analyzed with a variety of evaluation procedures ranging from sound performance theories, such as RMA to efficient discrete-event simulators. The reason for this flexibility is twofold. First, the evaluation procedures need neither be able to handle asynchronous calls nor keep track of call stacks because interpretation translates responses involving both into a sequence of serially executed subtasks. Second, evaluation procedures do not need an explicit notion of synchronization between tasks other than that resulting by virtue of the fixed priority scheduling.

Depending on its characteristics, a given performance model can be analyzed by some evaluation procedures and not by others. For example, models with unbounded execution or interarrival time distributions can be handled by a simulation-based evaluation, but not by worst-case analysis like RMA. For this reason, different evaluation procedures may dictate adhering to additional analytic constraints. In most cases, this is not a limiting constraint, but rather an enabler for predictability. For instance, when developing a system with hard real-time requirements where RMA will be used to predict worst-case response time, one should avoid designing responses with unbounded execution time.

The rest of this section describes the different evaluation procedures used in our performance reasoning framework. Some of them are third-party tools and have their own input format. In such cases, the performance model is translated to the particular format before it is fed to the tool for evaluation.

5.1 Worst-Case Analysis

Worst-case analysis predicts the worst-case response time to an event in the system by considering the maximum execution time of the components and the worst preemption and blocking effects. Worst-case analysis in the performance reasoning framework is carried out with MAST, a modeling and analysis suite for real-time applications [12]. Among other techniques, MAST implements RMA with varying priorities [8].

5.2 Average-Case Analysis

Average-case analysis computes the average response time to an event. Although this is achieved by discrete-event simulation in most cases, the performance reasoning framework also includes an analytic average-case analysis.

Simulation-based average-case analyses simulate the execution of a system taking into account the statistical distribution of interarrival and execution times. By collecting the simulated response time to thousands of arrivals, they can compute the average response time. In addition, they keep track of the best and worst cases observed during the simulation.

The performance reasoning framework supports three different simulation-based average case latency predictions. One of them is with Extend, a commercial general-purpose simulation tool that supports discrete-event simulation [13]. The simulation model is built out of custom blocks for Extend that represent the concepts in the performance model (i.e., tasks, subtasks, etc.). The Extend-based simulation can model several interarrival distributions and is able to simulate sporadic server tasks. Another simulation analysis is based on a discrete-event simulator called *qsim*. Being a special-purpose simulator, *qsim* is very efficient and is able to run much longer simulations in less time. The last of the simulation-based analyses uses Sim_MAST, a simulator that is part of the MAST suite. Sim_MAST can simulate a performance model providing statistical information about the response time to different events.

The analytic average case analysis in the performance reasoning framework uses queuing and renewal theory to compute the average latency of a task scheduled by a sporadic server [5]. Although this method requires specific constraints such as exponential interarrival distribution, it provides an envelope for the average latency without the need to run long simulations.

6 Example

Figure 8 shows a screenshot of the PSK showing the CCL design diagram for a simple robot controller. The controller has two main components that execute

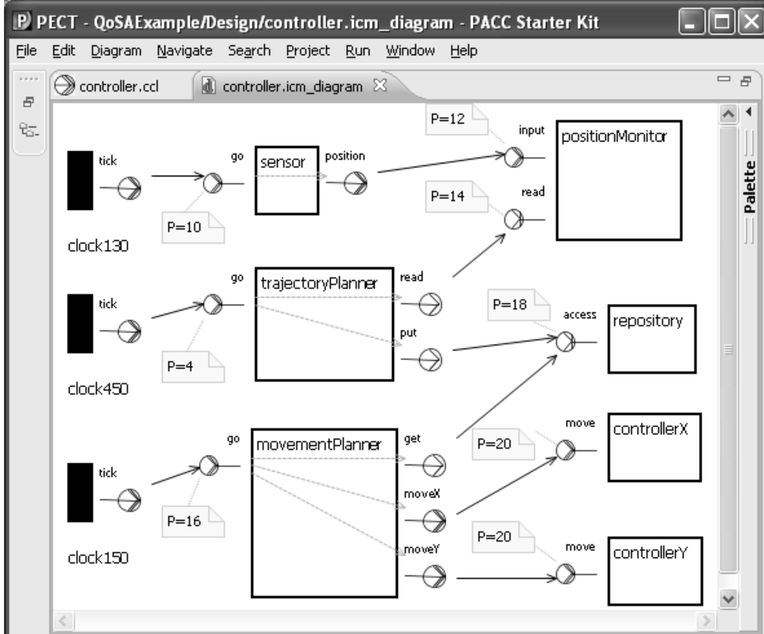


Fig. 8. Component and connector diagram for controller

periodically. The trajectory planner executes every 450ms and takes work orders for the robot. Using information from the position monitor, it plans a trajectory, translating the orders into subwork orders that are put into the repository. The movement planner, on a 150ms period, takes orders from the repository and converts them into movement commands for the two axes. The responses to these two periodic events have hard deadlines at the end of their period.

When the performance reasoning framework is called, the user selects the desired evaluation procedure and enters some analysis parameters. After that, the design model is transformed to a performance model, which is then evaluated. Figure 9 shows the performance model created from the design of the robot controller. As previously described, the performance model does not have concurrency within the responses to the different events, and it does not involve explicit synchronization between the responses. Figure 10 shows the results of a worst-case latency analysis using MAST. In this particular case, the result viewer indicates that the response to the 450ms clock is not schedulable because it has a worst-case execution time that overruns its period.

7 Related Work

There has been recent work in integrating performance analysis into model-driven development approaches [14,15,16]. Here we describe the similarities and differences with some of them. Woodside et al. [17] and Grassi et al. [14] have

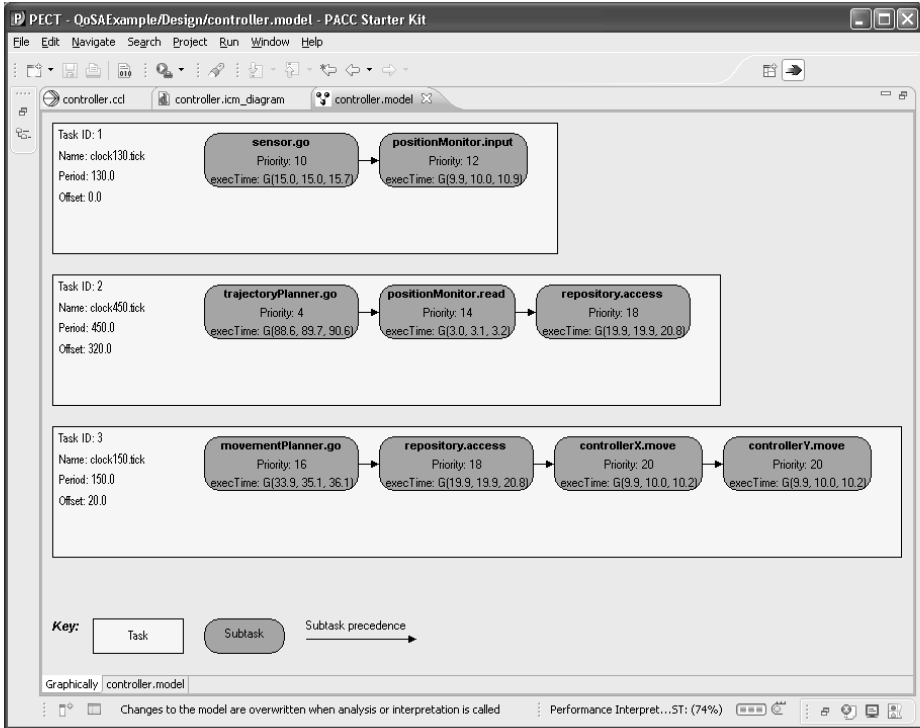


Fig. 9. Generated performance model for controller

proposed intermediate models—CSM and KLAPER respectively—to reduce the semantic gap between the design models and the analysis models and enable the use of analysis tools with different design languages. In that regard, ICM has the same intent. However, ICM is only one element in our reasoning framework, serving as an input meta-model for one of the key elements in the reasoning framework, namely, the interpretation that transforms the design into a performance model. Since our approach uses two meta-models, they are respectively closer to the start and end of the model-driven analysis process. For instance, ICM is closer to the component and connector view of the architecture than CSM and KLAPER. And the performance meta-model we use is close to the input needed by evaluation procedures based on RMA. An important contribution of our reasoning framework is the interpretation, which transforms the intermediate model into a performance model with simple semantics that can be analyzed by different procedures, including those that do not directly support rich semantics such as forking, joining, and locking.

D'Ambrogio [15] describes a framework to automate the building of performance models from UML design models. The approach uses meta-models to represent the abstract syntax of source and target models and then describe the transformation from one to the other using a model transformation language.

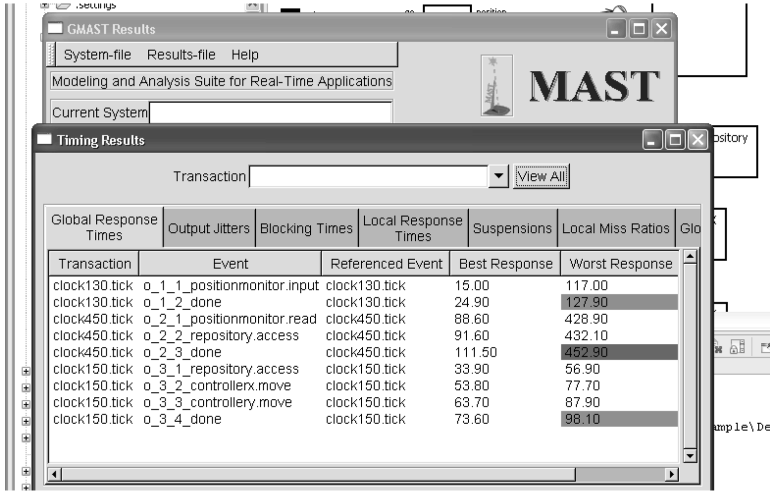


Fig. 10. Analysis results for the controller

This approach does not use intermediate models to reduce the semantic distance between source and target models.

Gilmore and Kloul [18] do performance modeling and prediction from UML models that include performance information in the transition labels of the state diagrams. They use performance evaluation process algebra (PEPA) [19] as an intermediate representation of the model. A key difference with our work is that our reasoning framework focuses on fixed-priority preemptive scheduling, making it suitable to analyze hard real-time systems. PEPA, on the other hand, assumes activities with exponentially distributed duration, whose memoryless property allows to treat preemption-resume scenarios as preemption-restart with resampling [20]. This approach is not suitable for real-time systems where more determinism is required.

Becker et al. [21] use the Palladio Component Model (PCM) to model component-based architectures including the information necessary for performance prediction. PCM is much more detailed than the ICM. For instance, component interfaces are first-class elements of the metamodel because they are used to check whether the connections between components through required and provided interfaces are valid. The elements closest to interfaces in ICM are sink and source pins. They are not associated with a type or service signature because it is assumed that the validity of the connection has already been established at the architecture description level–the CCL specification in our case. PCM also allows modeling the behavior of the component as far as necessary–an approach called gray-box–to determine the way required services and resources are used. This includes modeling parameter dependencies, loops, and branching probabilities. In ICM all the required services are assumed to be used exactly once for every invocation of the component. Certainly, compared to ICM, PCM allows modeling more details, that in turn means making fewer assumptions. However, to the best

of our knowledge, the complete details in PCM models cannot be automatically generated and PCM models can only be evaluated by simulation. In contrast ICM models are automatically generated from architecture descriptions and can be analyzed not only by simulation but also by a sound performance theory for worst-case response time and schedulability. Also, the simulation framework used with PCM does not support priority-based preemptive scheduling.

Analytic constraints play an important role in our approach because they define the space of designs that are analyzable by the reasoning framework. This characteristic is also present in the work of Gherbi and Khendek [16] where OCL is used to specify the constraints and assumptions of the schedulability analysis.

8 Conclusions

We began to work in the performance reasoning framework circa 2001. The initial versions had limited analysis capability, but successful validation of the predictions revealed great potential. More recently, we have expanded the space of analyzable systems by incorporating and adapting performance theories and diversifying the set of tools used in the evaluation procedure. In this process we found that creating metamodels and model transformations greatly reduced complexity in the reasoning framework implementation.

The performance reasoning framework has been applied successfully in several industry scenarios [22,23,24] and has proven to be very useful for early adopters of this technology. In maintenance scenarios, model-driven analysis is also useful. The performance annotations of the components in the architecture description can be changed to reflect the intended modifications and a new run of the analysis can verify whether the modifications will yield the required performance.

The performance reasoning framework is packaged as an Eclipse plug-in and can be used with different design languages thanks to the ICM metamodel and design-to-ICM adapters. A simple architecture description with structural information (wiring of components and connectors through synchronous and asynchronous ports) and performance annotations (e.g., priority, execution time) is the input for performance analysis. ArchE [25], an architecture expert tool, is an example of an Eclipse-based tool that has been adapted to use the reasoning framework. The PSK is a fully automated solution that includes the performance reasoning framework and uses CCL as the design language, providing a comprehensive MDE solution: the same architecture description enhanced with behavior information can also be used as input for code generation. An important consequence is that conformance between the code, the architecture and the analysis results is maintained.

Architecture description languages that have explicitly considered the characteristics of an application domain or the business needs of adopting organizations have been more successful [26]. The CCL language was designed to support the development of component-based safety-critical real-time systems, and its semantics are close to the target runtime environment. As a result, annotations that express the properties of components and connectors are simpler—by

contrast, a UML generic component or assembly connector would require extensive stereotyping and far more annotations to express the same things.

The performance reasoning framework continues to evolve. Working with academic and industry collaborators, we plan to extend the space of analyzable systems by relaxing some of the current analytic constraints. Integration with other performance analysis tools is also a goal. A limitation of our performance reasoning framework is that execution time variations caused by branching are only represented by the resulting execution time distributions. Since CCL and other design languages can express the behavior inside components, future work intends to overcome that limitation by having the interpretation look inside the state machine of the components, perhaps using a gray-box approach as in PCM, but trying not to hinder the automation or ability to analyze the model by means other than simulation.

References

1. Schmidt, D.: Model-driven engineering. IEEE Computer Magazine 39(2) (2006)
2. Ivers, J., Moreno, G.A.: Model-driven development with predictable quality. In: Companion to the OOPSLA 2007 Conference (2007)
3. Bass, L., Ivers, J., Klein, M., Merson, P.: Reasoning frameworks. Technical Report CMU/SEI-2005-TR-007, Software Engineering Institute (2005)
4. Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Gonzalez Harbour, M.: A practitioner's handbook for real-time analysis. Kluwer Academic Publishers, Dordrecht (1993)
5. Hissam, S., Klein, M., Lehoczky, J., Merson, P., Moreno, G., Wallnau, K.: Performance property theories for predictable assembly from certifiable components (PACC). Technical Report CMU/SEI-2004-TR-017, Software Engineering Institute (2004)
6. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, Reading (2003)
7. Wallnau, K., Ivers, J.: Snapshot of CCL: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute (2003)
8. Gonzalez Harbour, M., Klein, M., Lehoczky, J.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. Softw. Eng. 20(1) (1994)
9. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic task scheduling for hard real-time systems. Real-Time Systems 1(1) (1989)
10. Hissam, S., Moreno, G., Stafford, J., Wallnau, K.: Enabling predictable assembly. Journal of Systems and Software 65(3), 185–198 (2003)
11. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713. Springer, Heidelberg (2005)
12. Gonzalez Harbour, M., Gutierrez Garcia, J.J., Palencia Gutierrez, J.C., Drake Moyano, J.M.: MAST: Modeling and analysis suite for real time applications. In: The 13th Euromicro Conference on Real-Time Systems (2001)
13. Krahl, D.: Extend: the Extend simulation environment. In: WSC 2002: Proceedings of the 34th Winter Simulation Conference (2002)
14. Grassi, V., Mirandola, R., Sabetta, A.: From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In: 5th International Workshop on Software and Performance (2005)

15. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from UML models. In: 5th International Workshop on Software and Performance (2005)
16. Gherbi, A., Khendek, F.: From UML/SPT models to schedulability analysis: a metamodel-based transformation. In: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (2006)
17. Woodside, M., Petriu, D., Petriu, D., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: 5th International Workshop on Software and Performance (2005)
18. Gilmore, S., Leila, K.: A unified tool for performance modelling and prediction. Reliability Engineering and System Safety 89(1), 17–32 (2005)
19. Hillston, J.: Tuning systems: From composition to performance. The Computer Journal 48(4), 385–400 (2005) (The Needham Lecture paper)
20. Gilmore, S., Hillston, J.: The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In: Haring, G., Kotsis, G. (eds.) TOOLS 1994. LNCS, vol. 794, pp. 353–368. Springer, Heidelberg (1994)
21. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP 2007: Proceedings of the 6th International Workshop on Software and Performance, pp. 54–65. ACM, New York (2007)
22. Hissam, S., Hudak, J., Ivers, J., Klein, M., Larsson, M., Moreno, G., Northrop, L., Plakosh, D., Stafford, J., Wallnau, K., Wood, W.: Predictable assembly of substation automation systems: An experiment report, second edition. Technical Report CMU/SEI-2002-TR-031, Software Engineering Institute (2003)
23. Larsson, M., Wall, A., Wallnau, K.: Predictable assembly: The crystal ball to software. ABB Review (2), 49–54 (2005)
24. Hissam, S., Moreno, G.A., Plakosh, D., Savo, I., Stelmarczyk, M.: Predicting the behavior of a highly configurable component based real- time system. In: ECRTS 2008: Proceedings of the 20th Euromicro Conference on Real-Time Systems. IEEE Computer Society, Los Alamitos (2008)
25. Bachmann, F., Bass, L.J., Klein, M., Shelton, C.P.: Experience using an expert system to assist an architect in designing for modifiability. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA) (2004)
26. Medvidovic, N.: Moving architectural description from under the technology lamppost. In: 32nd Euromicro Conference on Software Engineering and Advanced Applications (2006)

Architectural Specification and Static Analyses of Contractual Application Properties*

Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien

Université Lille 1 - LIFL CNRS UMR 8022 - INRIA
40, avenue Halley - Bât. A, Park Plaza
59650 Villeneuve d'Ascq, France
{Guillaume.Waignier,Anne-Francoise.Le-Meur,Laurence.Duchien}@lifl.fr

Abstract. Being able to specify and verify contractual application properties at the architecture level allows architects to build better architected and more reliable systems.

In this paper, we propose a model-based framework for designing contractualized architecture, independently of any paradigm (components or services). It enables a software architect to express the structural, behavioral, dataflow and QoS properties of his/her application. Our framework composes these properties in order to compute and check the properties of the assemblies incrementally. This allows architects to see the influence of their design decisions on the quality of his/her architecture and thus helps them to better design their systems architecture.

1 Introduction

Specifying software architectures has become a central and crucial activity in the design of complex and distributed systems. At design-time, software architects care about specifying the composition of the necessary functionalities, *e.g.,* components or services, to form the desired system, as well as verifying that the resulting assembly is coherent in order to maintain a certain level of system integrity.

System integrity relies on many kinds of properties. Typical properties are concerned with the structural and behavioral relationships among components or services. These properties are important because they allow architects to capture and understand the dependencies between the various components. Architects may also need to reason about quality properties, such as performance, security, reliability, *etc.* Finally, some requirements may concern the properties of the exchanged data, such as their values on the dataflow. However these information are rather used at runtime.

It is extremely important to identify early in the system development process that some component dependencies are incoherent or that a given quality property cannot be satisfied. Because systems are large and exhibit complex interactions, it is unrealistic to expect a software architect to be able to perform

* This work was partially funded by the French ANR TL FAROS project.

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 152–170, 2008.
© Springer-Verlag Berlin Heidelberg 2008

global compositional reasoning or to predict the overall architecture properties just on the basis of component properties. To provide architects with some support with respect to this issue, software architecture description languages are often coupled with analysis tools.

There exist several architecture description languages (ADLs) [1]. Each ADL has been designed to support the description and analysis of specific characteristics of a system's architecture. Consequently, an ADL is often tied up with its associated component or service model, which means that the design of an architecture is dependent of the underlying platform and must thus be re-thought and re-analyzed if the architecture shall be implemented on another platform. Some generic ADLs exist, such as Acme/Armani, but they mainly support the analysis of structural properties. Furthermore, to the best of our knowledge, there exists no ADL that supports structural, behavioral, dataflow and QoS application contraint specifications, and no associated analysis tools that enable the static global resolution of these contractual specifications over an architecture.

In this paper, we propose a framework to specify and analyze component or service-based architectures, independently of any given underlying platform. With this framework, a software architect can describe the configuration of his/her architecture using our structural model, which includes commonly used architectural elements, and can specify contractual application properties which capture the requirements on the structure, the behavior, the dataflow and the QoS of the application. These contractual specifications are based on the *assume-guarantee* paradigm and allow static analysis tools to check their global composition coherence. Verification can be performed incrementally, as the architecture elements get connected with each other. Furthermore if global coherence can not be validated, the specification that can not be satisfied is pointed to the architect. Overall our framework enhances the quality of the design of an architecture.

The rest of the paper is organized as follows. Section 2 illustrates with an architecture example the various contractual application specifications that an architect may need to express and gives an overview of existing approaches to specifying contracts. Section 3 presents our structural architecture model and its associated application contractual specifications are described in Section 4. Section 5 explains how specification composition is performed. Section 6 explains how platform-specific architecture characteristics can be handled in our model and presents the limitation of our approach. Finally Section 7 presents some related work and Section 8 concludes and provides some future work.

2 Example

To illustrate the design of an architecture with contracts, we use an example of architecture in the context of the French Personal Health Record system (PHR) [2]. We present the scenario and express the application specific properties that the software architect would like to be able to specify in order to design a more reliable PHR system. Finally, we illustrate through this example the capabilities and lacks of actual approaches for the design of safe architectures.

2.1 Overview of the PHR Example

PHR is the French personal health record system that will be able to provide health-care professionals with the information needed for their patients care. Figure 1 represents a possible architecture of the PHR system. All medical information, (such as biological analyses, X-rays, medications, *etc.*), will be stored in distributed databases (`Access`) and will be made accessible through an on-line interface (`Client`).

The first requirement of this system architecture is related to authentication issues. Indeed, of course, not everybody should have access to anybody's health records. The architecture of this system must thus provide some authentication mechanism. The `Authentication` architecture element logs a health-care professional in and returns a session ticket through the functionality `getTicket` that is offered by `SessionServer`. For security reason, the functionality `getTicket` can be used only by the element `Authentication` to avoid that an unauthenticated user get a session ticket. Finally the session ticket must be validated by the `SessionServer` before retrieving any medical data from the database.

The second requirement is the high reliability of the system. Such system has to be able to handle very heterogeneous medical information, going from lightweight text records to gigabytes of echographies. Furthermore, the devices used to display this information are also heterogeneous. They range from desktop computers with high-quality large-screen monitors and gigabyte network connexions to simple PDAs with small screens and low-bandwidth GPRS network connexion. Handling such data in a reliable way is critical because the system must be able to determine if a given data can be displayed appropriately with no loss of information, as well as in which time-frame, depending on the available resources and amount of information to display. Consequently, to enhance the reliability of the system, it is important to be able to take into account at design-time the device profiles and the type of the exchanged data.

2.2 PHR System Application Properties

The structure of the PHR system is shown in Figure 1. To design a more reliable PHR system, the architect needs to specify some *application properties*: one structural property, nine behavioral properties, four properties on the dataflow and one QoS property.

For security reason, the software architect needs to be able to add a structural property on the functionality `getTicket` of the architectural element `SessionServer` in order to restraint its use to the element `Authentication`.

For authentication reason, the architect has to express that the session ticket must be validated by the functionality `checkTicket` of the element `SessionServer` before calling the functionality `getData` in one of the `Access` elements. To do so, he/she needs to specify a behavioral property on each architectural element in order to describe their patterns of interaction with the environment, *i.e.*, their workflow. Theses properties can also be used to check that the overall behaviour of the PHR system has no deadlock.

Fig. 1. PHR's architecture

For reliability reason, the software architect wants to be assured that the device of the health-care professional never receives medical data that violate its profile and leads to its dysfunction. To do so, he/she needs to express some properties on the dataflow. He/she would like to specify that the medical data received by the `Client` is smaller than 100Mb and of type JPG, that the elements `Access` only send JPG medical data, and that the element `GlobalSearch` propagates medical data without modifying them.

For performance reason, the software architect needs to capture the fact that health-care professionals should not wait more than 10s the reception of medical data.

If the consistency of these application properties can not be guaranteed statically, due to a lack of information at design-time, the software architect would like to know what the inconsistencies are and when they appear as early as possible in the dataflow. With this information, the architect will be able to create a more responsive PHR system when property violation arises.

2.3 Specifying Application Properties with Existing ADLs

To best of our knowledge, there does not exist any ADL that handles all the kinds of properties that are needed by the PHR system. Table 1 presents an excerpt of some ADLs and classifies them according to the choices used to express the properties. Indeed, each ADL focuses on one specific aspect. Some

Table 1. Support of properties in ADLs

	Fractal, CCM, SCA	Wright	SOFA, Fractal-BPC	Acme / Armani	Confract	SafArchie	WebService
Structure	n/a	Style	n/a	Armani language	n/a	Type	n/a
Behaviour	n/a	CSP [3]	Behavioral protocol	n/a		SFSP [4]	BPMN/BPEL
Dataflow	n/a	n/a	n/a	pre/post conditions	pre/post conditions	pre/post conditions	n/a
QoS	n/a	n/a	n/a	n/a	n/a	n/a	SLA

ADLs and components models, such as Fractal [5], CCM [6] or SCA [7], are not concerned about the definition of application properties. The expression of application structural properties is present in many ADLs in different forms. They correspond to architectural styles in Wright [8], types of components in SafArchie [4] and a first order predicate logic language in Acme/Armani [9]. The specification of behavioural properties is well supported by ADLs, such as Wright, SOFA [10] or SafArchie, that care about providing some static guarantees, *i.e.*, deadlock detection. Some works have backported existing behavioral verification tools from other ADLs. For example, the BPC extension of Fractal is a rewrite of the support of behavioral protocol of SOFA. The specification of dataflow properties is less commonly offered. Dataflow specification is realized with pre and post-condition in Confract [11], Acme/Armani or SafArchie. Nevertheless, this kind of specification is not used at design-time to inform the software architect of the potential violation. For example, in Confract, this specification is directly transformed into executable assertions. Finally, few components model support the specification of QoS properties. WSLA [12] enables QoS specifications to be added on Web service description.

We propose a framework to capture and verify application properties. The architect improves the structural description of his/her architecture by writing the corresponding contractual specifications. These specifications are used to inform the architect if the desired level of application architecture integrity can be satisfied.

3 Designing the Structure of an Application Architecture

A well-known definition of the term software architecture is proposed by the IEEE Standards Association Systems: "Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." Although there is no universal definition, it is commonly accepted that the specification of a software architecture describes at least the entities contained in the architecture, which may be components or services depending on the paradigm used, as well as the relationships between these entities. Consequently, after studying many existing ADLs, we have chosen to describe the structure of a software architecture with five architectural concepts, common to component-based and service-oriented paradigms, and with which architects are familiar.

The architectural concepts available in our approach to an architect to describe the structure of an application are shown in Figure 2. An *Entity* is a computational element or a data store of a system, it is a component in an component-based architecture or a service in a service-oriented architecture. A *Connector* describes the interactions between entity, it is equivalent to a binding in many component-based approaches or a partnerLink in service-oriented architecture. A *CommunicationPoint* is an element of an entity that enables the entity to communicate with its environment, *i.e.*, the other entities. It is called a

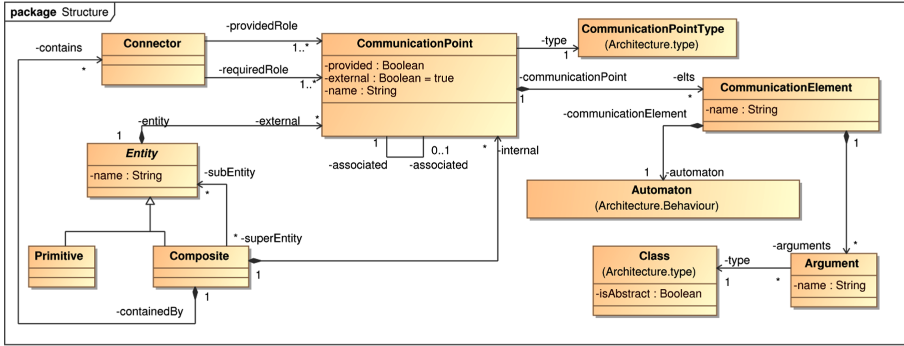


Fig. 2. Structural meta-model

port or interface in a component-based architecture and a portType in a service-oriented architecture. A *CommunicationElement* is an action. It can be called operation in some component models and service-oriented approaches. An *argument* is the element exchanged between the entity. It can be a typed object or a message.

Furthermore, we offer the possibility of hierarchically organizing entities by distinguishing *primitive* entities from *composite* entities. A primitive entity is a basic computational element, it can be viewed as a black box. A composite entity defines a given assembly of primitive and composite entities, it is equivalent to a composite component or can be seen as an orchestration.

Moreover, communication points must be provided or required, depending on whether they offer or require functionalities. The communication protocol of the communication elements are described with an automaton. It expresses the order of the sending and reception of the arguments. It can be used to specify synchronous or asynchronous communication protocols.

Finally, we have constrained our structural meta-model with some OCL [13] invariants in order to perform a minimal set of checks to guarantee that the architecture is well-formed. For example, we check the strict encapsulation of sub-entities or make sure that a connector is between a provided and a required communication points. We have reduced the number of these structural checks to its minimum. All the other constraints, may they be structural, behavioral, *etc.*, are considered application specific, and will be specified by the architect to satisfy his/her application context.

4 Contractual Specifications

In section 2 we have highlighted the application properties that the architect would like to capture and verify at the architecture level. In this section, we present the different contractual specifications that an architect can write in our approach. Section 5 will explain how these specifications can be used to inform the architect if the desired level of application architecture integrity is satisfied.

4.1 Definition

A contractual specification is always associated to a *participant* P and relies on the assume-guarantee paradigm [14] such that

- $assume(P)$ makes explicit what application properties P requires from its environment, *i.e.*, everything that interacts with P.
- $guarantee(P)$ makes explicit what properties P offers to its environment.

Based on the specification of these *assume* and *guarantee* properties, it will be possible to determine if the overall architecture satisfies all these properties such that $\text{assume}(P) \rightarrow \text{guarantee}(P)$, where the operator $\rightarrow$ is a form of logical implication that expresses the fact that if $\text{assume}(P)$ is true then $\text{guarantee}(P)$ is true [14].

This assume-guarantee approach has often been used in the context of QoS [12,15]. We generalize the use of this approach to the specification of structural, behavioral and dataflow contractual properties. Accordingly, a software architect can associate four kinds of contractual specifications on his/her architecture: structural specifications to constrain the structure of his/her architecture; behavioral specifications to describe the synchronization of the entities, *i.e.*, their workflow; dataflow specifications to restrict the values of the arguments in the dataflow; and QoS specifications to capture the entity extra-functional properties, such as the expected response-time.

All of these specifications are application specific and will be composed and analyzed to verify that the application architecture satisfies all the desired properties. The next sub-sections present these specifications, which are illustrated in the context of our PHR application example.

4.2 Structural Specification

Structural specifications are associated to communication points and allow the architect to express the structural requirements of his/her application. Their definition are expressed using the OCL language, which is a language well-known by software architects. We constrain the use of OCL to its invariants and provide a new keyword `other`, which makes reference to the *other* communication points that the contextual communication point, *i.e.,* the communication point that carries the structural specification. The architect only expresses application structural requirements, which corresponds to specifying an *assume* expression in order to constrain the possible connections of a given communication point.

PHR Example. To express the structural property presented in Section 2.2, *i.e.*, that the communication point `getTicket` of the entity `SessionServer` should only be used by the entity `Authentication`, the software architect can attach the following structural specification `S1` to the communication point `getTicket`:

```
S1 on SessionServer.getTicket : assume other.entity.name=‘‘Authentication’’
```

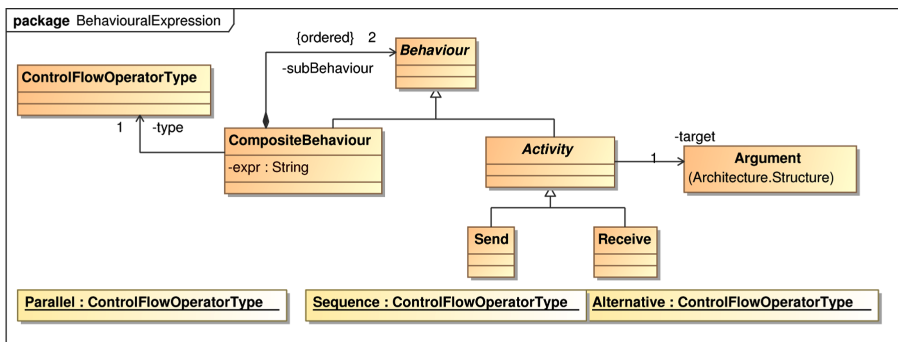


Fig. 3. Workflow meta-model

4.3 Behavioral Specification

Behavioral specifications are associated to communication points and allow the architect to capture the behavior of his/her architecture entities by describing the workflow going in and out of the communication points of these entities. They can be either an *assume* or a *guarantee* expression. The guarantee describes the behavior that the entity respects and the assumption expresses the behavior that the entity requires. The software architect can describe these specifications with the SFSP language or by designing a workflow conformed to the meta-model shown in Figure 3. The basic activities represent the sending or the reception of an argument and the control flow operators can be of three types: sequence, alternative or parallel.

PHR Example. We focus on the workflow that involves the entities `GlobalSearch` and `Access` and specify the *assume* and *guarantee* expressions to express the requirements presented in Section 2.2. For example the guaranteed behavior of the entity `Access` at the communication point `getData` is as follows:

```
B1 on Access.getData : guarantee ?getPicture.URL ; !getPicture.data$
```

This guarantee `B1` attached on the communication point `getData` of the entity `Access` expresses that the communication element `getPicture` in this communication point receives ('?') an argument URL of a medical data and then returns ('!') the argument data in response ('$'). Similarly, the architect expresses the guaranteed behavior of the entity `GlobalSearch`:

```
B2 on GlobalSearch.searchData, GlobalSearch.getData : guarantee
   ?searchData.searchPicture.URL;!getData.getPicture.URL;
   ?getData.getPicture.data$;!searchData.searchPicture.data$
```

The entity `Access` also requires that the communication element `verifyTicket` of the communication point `checkTicket` of the entity `SessionServer` be called before calling its communication element `getPicture` of its communication point `getData`. This can be specified with the following assumption:

```
B3 on Access.getData : assume
   ?SessionServer.checkTicket.verifyTicket.ticket;?Access.getData.getPicture.URL
```

4.4 Dataflow Specification

Dataflow specifications are associated to the workflow and allow the architect to restrict the values of the input and output arguments in the flow. Dataflow expressions are *assume* expressions when they apply to input argument and *guarantee* expressions in the case of output arguments.

PHR Example. To describe the profile of the `Client` entity presented in Section 2.2, the architect can specify the dataflow specifications `D1` and `D2` as follows:

```
D1 on Client.searchData : assume searchPicture.data.size<100;
D2 on Client.searchData : assume searchPicture.data.type=JPG;
```

The architect must also express that the entity `Access` only returns JPG pictures:

```
D3 on Access.getData : guarantee getPicture.data.type=JPG;
```

Finally, it is necessary to specify that the `GlobalSearch` entity does not modify the data that it receives:

```
D4 on GlobalSearch.searchData : guarantee
   searchData.searchPicture.data=getData.getPicture.data;
```

4.5 QoS Specification

QoS specifications cover a wide range of non-functional properties including performance, security, reliability and availability. Each non-functional property has its own metric and expression language. To handle this unbound set of properties, we have introduced the concept of QoS specification type. For each non-functional QoS property that an architect might want to reason about, there must exist its corresponding QoS specification type. We provide several pre-defined QoS specification types and our approach enables one to define new ones as needed.

A QoS specification is an *assume* (*resp. guarantee*) expression if it corresponds to a pre-condition (*resp.* post-condition) on the value of a non-functional property. A Qos specification is associated to the workflow.

PHR Example The architect needs to express the requirement that the healthcare professional must receive the medical data before a given maximal time. This can be done by adding a QoS specification `QoS1` of type `MaximalResponseTime` on the reception of the data by the communication element `searchPicture` of the communication point `searchData` of the entity `Client`.

```
QoS1:MaximalResponseTime on Client.searchData : assume Tmax(searchPicture)<10s
```

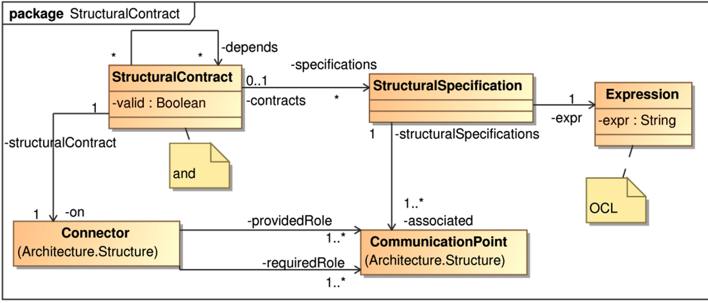


Fig. 4. Structural contract meta-model

5 Contractual Specification Composition

In this section, we explain how contracts are automatically computed by composing the specifications. This computation is generic and independent of any component models or service-oriented approach, as it is based on our generic models. For all kinds of contracts, we describe how the computation is performed. First, we present the computation of the structural, behavioral, dataflow contracts and QoS. Then, we discuss about the impact of the order of the composition.

5.1 Contract Computation

In this section, we define how the contractual specifications associated to participants are composed to form a contract.

Let c be a participant associated with the contractual specification $Spec_c$ and $P = \{c_i, ..., c_n\}$ a set of participants. We define the contractual specification $Spec_P$ of the set P as the pair $\{A_P, G_P\}$, where A_P is the assumption and G_P is the guarantee (cf. Section 4). The contract $C_{\{P_1 \star P_2\}}$ resulting from the interaction of P_1 and P_2 is computed such that $C_{\{P_1 \star P_2\}} = Spec_{P_1} \bullet Spec_{P_2}$ where $\bullet$ is the specifications composition operator and $\star$ is the interaction operator. The operator $\bullet$ and $\star$ are specific for each kind of contractual specification, *i.e.*, structural, behavioral, dataflow and QoS. The result of the contract is used to compute the specification $Spec_{\{P_1 \cup P_2\}}$ incrementally.

5.2 Structural Contracts

The structural contract (cf. Figure 4), which results from the interaction between a set of required P_1 and provided P_2 communication points connected together through a connector, is computed with the operator `and` of OCL such that $C_{P_1 \star P_2} = \{A\}$ where $A = A_{P_1}$ *and* A_{P_2}. The result of the composition is an OCL invariant, which can be checked statically by OCL tools.

PHR Example. When the communication point `getTicket` of `SessionServer` is connected with the communication point `P` of `Authentication`, the contract $C_{getTicket \star P}$ is computed by composing the specification `S1` (cf. Section 4):

$C_{checkTicket\star P}$ = `P.component.name=''Authentication''`.

This contract is valid.

5.3 Behavioral Contracts

The behavioral contract (cf. Figure 5) resulting from the interaction between communication points P_1 and P_2 is computed with the pair of operators $\{\cup, \|\}$, where $\|$ is the synchronization operator of SFSP, such that $C_{P_1 \star P_2} = \{A, G\}$ where $A = A_{P_1} \cup A_{P_2}$ and $G = G_{P_1} \| G_{P_2}$.

The resulting contract expresses the behavior of the assembly. A behavioral contract is checked statically in two steps. First, (i) the guarantee of the assembly must be valid, *i.e.*, there must be no deadlock. Then, (ii) the assumption is checked, it corresponds to checking the safety properties, *i.e.*, that it is not possible to enter into an error state.

The resulting contract $C_{P_1 \star P_2}$ becomes the behavioral specification $Spec_{P_1 \cup P_2}$ of the new assembly of entities, called a composite specification (cf. Figure 5).

PHR Example. When the communication points `getData` of `Access` and `getData` of `GlobalSearch` are connected together, the contract $C_{getData\star getData}$ is computed by composing the behavioral specification `B1`, `B2` and `B3` (cf Section 4).

```
guarantee ?GlobalSearch.searchData.searchPicture.URL;
  !GlobalSearch.getData.getPicture.URL;?Access.getData.getPicture.URL;
  !Access.getData.getPicture.data$;?GlobalSearch.getData.getPicture.data$;
  !GlobalSearch.searchData.searchPicture.data$
assume ?SessionServer.checkTicket.verifyTicket.ticket;?Access.getData.getPicture.URL
```

The behavioral expression that forms the guarantee has no deadlock but the assumption is not valid. It will be valid when the entity `SessionServer` will be taken into account in the assembly. The computed contract becomes the specifications of the new assembly associated with `Access.getData`, `Global-Search.searchData` and `GlobalSearch.getData`.

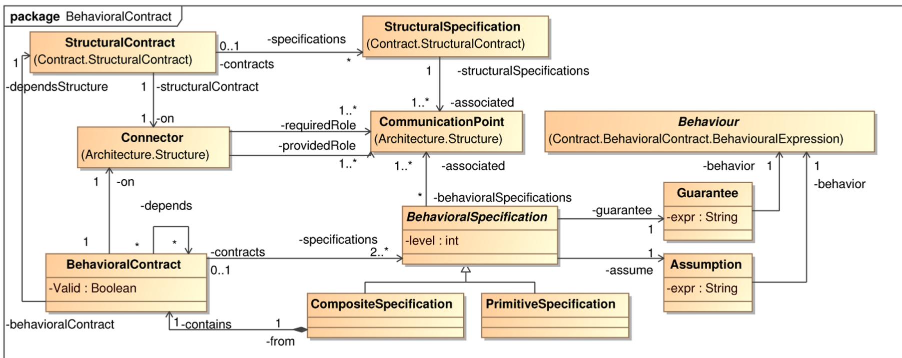


Fig. 5. Behavioral contract meta-model

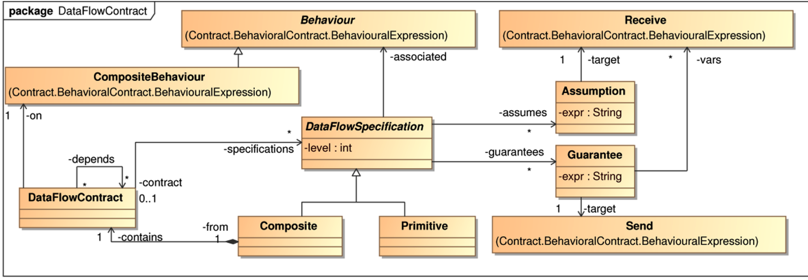


Fig. 6. Dataflow contract meta-model

5.4 Dataflow Contracts

A dataflow contract $C_{P_1 \star P_2}$ results from the composition of the dataflow specifications $Spec_{P_1}$ and $Spec_{P_2}$ associated to the workflow, such as a sequence or a reception of an argument (cf. Figure 6). It corresponds to resolving the set of pre-conditions A_{P_2}, where the post-conditions G_{P_1} are true if P_1 appears before P_2 in the workflow. It is similar to a partial validation like in program checking: $C_{P_1 \star P_2} = resolve(A_{P_2})$ where G_{P_1} is true if P_1 is before P_2 in the workflow.

To analyse the pre and post-conditions, our approach is similar to [16]. We attach the specifications on the nodes of the workflow. The output state of a node is a function of the state of this node entry. The input state of a node is a function of output state of the above node. The assumptions are resolved locally knowing the guarantees. The result is the equation $C_{P_1 \star P_2}$ that can be $true$, $false$ or not resolvable.

The dataflow contract is used to compute the dataflow specification $Spec_{P_1 \cup P_2}$ of the new workflow, such as shown in Table 2. Guarantees are propagated to the next nodes using a forward analysis.

Thanks to a backward analysis, the partially valid assumption is propagated back in the control-flow graph up to the first alternative. It prevents the system from entering into an execution path if an assumption in this path is not respected. Consequently, the software architect can take into account a possible contract violation at design time, which is detected as soon as possible in the dataflow path.

Table 2. Composition Operators for dataflow

$Spec_{P_1 \cup P_2}$ $= \{A, G\}$ where	Sequence $(P_1; P_2)$	Parallel $(P_1 \parallel P_2)$	Alternative $(P_1 \mid P_2)$
$A =$	$A_{P_1} \cup C_{P_1 \star P_2}$	$A_{P_1} \cup A_{P_2}$	n/a
$G =$	$G_{P_1} \cup G_{P_2} \cup A_{P_2}$	$G_{P_1} \cup G_{P_2}$	$G_{P_1} \cup G_{P_2} \cup A_{P_1} \cup$ condition

PHR Example. When the architect connect the communication points `getData` of `Access` and `getData` of `GlobalSearch`, the workflows of the two entities interact and produce the dataflow contract $C_{GlobalSearch \star Access}$. It is computed by composing the dataflow specifications `D3` and `D4` (cf. Section 4). There is no assumption,

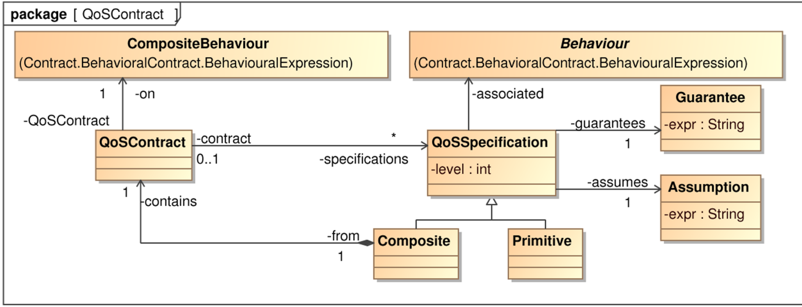


Fig. 7. QoS Contract meta-model

so the contract is valid. Finally, the specification $Spec_{GlobalSearch \cup Access}$ is computed in sequence such that:

```
on GlobalSearch.searchData;Access.getData :
 guarantee GlobalSearch.searchData.data.type=JPG and
 GlobalSearch.searchData.searchPicture.data=GlobalSearch.getData.getPicture.data;
```

When the software architect connect the communication point `getData` of `Client` with `getData` of `GlobalSearch` together, the assumptions `D1` and `D2` are checked knowing the guarantee $Spec_{GlobalSearch \cup Access}$: $Spec_{Client \cup GlobalSearch \cup Access}$ = `assume searchPicture.data.size<100`.

The assumption `D2` (JPG picture) is statically validated, but it is not possible to guarantee that data size is smaller than 100Mb. Thanks to a backward analysis, `D2` is propagated back in the workflow up to `!Access.getData.getPicture.data$`.

5.5 QoS Contracts

A QoS contract $C_{P_1 \star P_2}$ results from the composition of the QoS Specifications $Spec_{P_1}$ and $Spec_{P_2}$ associated to the workflow (cf. Figure 7). It is valid if the assumption required by the participants is fulfilled by the guarantee provided by the other participants that are appearing before in the workflow, as in QML [15].

The QoS specification $Spec_{P_1 \cup P_2}$ of the new workflow can be computed incrementally by composing the QoS specifications of P_1 and P_2. The QoS composition operators depend on the type of the QoS. Thus, we have defined a model of QoS type that enables a given QoS expert to create a new type of QoS by associating for each workflow operator the appropriate QoS composition operator. For example, [17] defines the QoS composition operators for the maximal and minimal time estimation and cost estimation (cf. Table 3). The use of the QoS concept type enables us to separate the definition of QoS composition operators and the algorithm to traverse the workflow, allowing the support for new QoS properties.

5.6 Contract Composition Order

Contracts $C_{P_1 \star P_2}$ are computed in the following order: structural, behavioral, dataflow and then QoS. Specification $Spec_{P_1 \cup P_2}$, based on these contracts, are

Table 3. Example of QoS Composition Operators

$Spec_{P_1 \cup P_2} = \{A, G\}$ where

Behavioural operators	Sequence	Parallel	Alternative
Maximal Response Time	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = max(A_{P_1}, A_{P_2})$ $G = max(G_{P_1}, G_{P_2})$	$A = max(A_{P_1}, A_{P_2})$ $G = max(G_{P_1}, G_{P_2})$
Minimal Response Time	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = min(A_{P_1}, A_{P_2})$ $G = min(G_{P_1}, G_{P_2})$
Maximal Cost	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = max(A_{P_1}, A_{P_2})$ $G = max(G_{P_1}, G_{P_2})$
Minimal Cost	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = sum(A_{P_1}, A_{P_2})$ $G = sum(G_{P_1}, G_{P_2})$	$A = min(A_{P_1}, A_{P_2})$ $G = min(G_{P_1}, G_{P_2})$

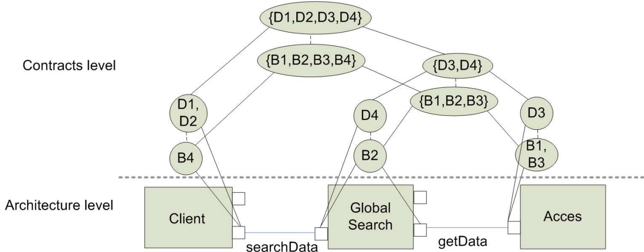


Fig. 8. Example of contracts composition

composed incrementally. The relation between these specifications is a tree. The leafs are the contractual specifications written by the architect and the nodes are the intermediate contractual specifications. The root is the specification of the entire architecture (cf. Figure 8). The incremental computation of the specification allow the recomputation of only the specifications of the modified part of the architecture, independently of the execution platform.

6 Discussion

This section first explains how plateform-specific properties can be handled in our model and then gives the limitations of our approach.

6.1 Handling Platform-Specific Properties

Our component model is built from architectural concepts that are common to many ADLs. However, each component model has its own platform-specific properties. For example, the notion of communication point compatibility is model specific, *e.g.,* two communication points are compatible : in Fractal only if they have the same type; and in CCM if they have the same type and the same communication protocol (synchronous or asynchronous). These platform-specific properties can be expressed on our model by writing contractual specifications. For example, Fractal specifications correspond to the following structural specification : `S_fractal on * : assume other.type = self.type`.

These platform-specific properties may be even more complex. For example, in AADL [18], there exist different types of components, such as `Processor`, `Memory` or `Device`. Thus some compatibility rules specific to AADL need to be checked, such as "It is forbidden to connect two `Device` together". All of these rules can be expressed on our model. Specific component types are expressed with an attribute in the `Entity` and properties that are specific to these attributes are expressed like any other contractual specifications. Furthermore, to relieve the architect from having to specialize his/her architecture to satisfy a given ADL each time another platform is considered, we plan to create platform-specific styles, which contain platform-specific properties that can be automatically loaded within our model.

6.2 Limitations

Our structural model is less generic than the one of Acme or xADL. Contrary to Acme or xADL, our model makes explicit the content of a communication point. It can be required only or provided only and must contain a set of communication elements with typed input/output arguments. Thus our structural model is compatible with any component model that exhibits bidirectional communication points (in/out or required/provided), which is a characteristic that is shared by many component models, such as Fractal, CCM, SCA or SOFA. The component model Unicon [19], however, is not supported since it has 14 kinds of communication points.

We have chosen to design meta-models with strong semantics in order to make them non-ambiguous, allowing thus analyses to be performed. Indeed the semantics of communication points must be clear to enable analysis tools to manipulate them. In Acme/Armani, the content of a communication point, called port, is not expressed. The only possibility to add information in a communication point is by defining properties, which have no clear semantics. For example, it is possible to perform Wright behavioral analysis on an Acme description by adding some Acme properties containing CSP expressions. However there is no relationship between these properties and the structure of the architecture. Consequently, it is not possible to guarantee that the messages declared to be sent or received structurally exist. In our approach, the behavioral meta-model is strongly associated with the structural meta-model. The sent and the received messages expressed in the behavioral model correspond to the argument in the structural model. So, for example, our approach checks that the sent (*resp.* received) messages in the behavioral model correspond to the output (*resp.* input) arguments in the structural model.

More generally, we have chosen to have a richer structural meta-model than Acme or xADL in order to be able to check the coherence between the different meta-models, *i.e.*, structural, behavioral, dataflow and QoS. In our approach, the syntactic coherence defined in [20], *i.e.*, the reuse of elements between different models, is guaranteed by the meta-modeling approach. Each element is a meta-class and the reuse of elements correspond to the reuse of meta-classes between the models. For example, the `Argument` used in the structural model (Figure 2)

is exactly the same `Argument` reused in the behavioral meta-model (Figure 3). The semantic coherence defined in [20] (*i.e.*, elements are designed using the "meaning" and interpretation of others elements) is guaranteed by the meta-modeling approach and the static validation tools. Indeed, each meta-class has a unique semantics and each concept with a same semantics is represented by a unique meta-class.

6.3 Implementation Status

We are implementing the tools as an *Eclipse* plugin. The models are created with EMF and the OCL constraints are implemented with the OCL-EMF validation framework.

7 Related Work

We have based the design of our structural model on several works that have proposed a generic ADL, such as Acme [21], xADL [22] and FIESTA [23], and on works that have, more recently, unified components and services [7,24]. With our model, our goal is not to provide yet another model, but to identify a meaningful subset for the architect to build on and reason about application contractual specifications.

Our contractual architecture framework, which supports four kinds of contractual specifications, brings together in a uniform way some contract concepts that were dispatched in various approaches.

Acme/Armani [9] enables the adding of structural constraints on an architecture and is supported as an extension in AcmeStudio, a graphical user interface for designing an architecture independently on any ADL. However, Acme/-Armani does not handle behavior specification, and thus is not appropriate for the design of service-oriented architectures, which focus on behavior description, *i.e.*, the orchestration.

With respect to behavioral specifications, we use a process algebra similarly like in many ADLs, such as CSP [3] in Wright, SFSP [4] in SafArchie or FSP [25] in Darwin. In service-oriented approaches, the behavior is described with BPMN [26] or BPEL [27]. At a high-level of abstraction, it is similar to an activity diagram, which can be described with an process algebra. Our current version of the FSM is not as expressive as CSP or FSP, we have however selected a meaningful subset by keeping the sequence, alternative and parallel composition operators. Moreover we offer a rich structural meta-model that enables us to define new behavioral meta-model representing an existing process algebra.

In [28], the authors express dataflow specifications on an architecture with Acme/Armani through the use of pre and post-conditions, which are all transformed into executable assertion at runtime. No static analysis is provided in order to perform partial validation. In our approach, we inform the software architect at design-time where potential inconsistencies may take place, as early as possible in the dataflow. This allows the architect to identify at which points

in the architecture runtime tests should be inserted. Because, these points are detected at the earliest in the dataflow, the resulting system should be more reactive to constraints violation.

QML [15] is a QoS specification language that introduces contract types. A contract type C defines each dimension type within C. A dimension type specifies a domain for a dimension, which can be a set, an enumeration or a numeric. However, the static resolution of QML contracts is only local. In our approach, QoS contracts are composed in order to compute the QoS contract of the entire architecture. To do so, we have extended the concept of contracts type of QML by describing also how the contract is composed using a set of QoS composition operators. A QoS contract composition operator is associated with each workflow composition operator.

[29] presents the needs for the second generation of ADLs. The second generation of ADLs must support the three concerns: technology, domain and business. The technological concern includes means for representing and reasoning about architectures. The domain concern includes means for representing and reasoning about problems in a given domain. The business concern corresponds to capturing and exploiting knowledge of the business context. Our approach takes into account the technology concern since it covers a wide range of existing ADLs and proposes some tools to check structural, behavioral, dataflow and QoS properties. The domain concern is supported by our approach because it provides a means to specify application specific-properties. Moreover, models can be personalized with domain-specific properties, such as the avionic domain properties expressed in AADL. The business concern is not handled by our approach since our framework does not offer the generation of components and the deployment of the system. However we plan to add runtime management to our framework.

8 Conclusion

We have presented a generic framework for designing contractualized architectures. Our approach is based on a generic architecture model and on four models of contractual specifications that address the structure, behavior, dataflow and QoS properties of an application. Our architecture model contains few concepts of communication points compatibility. Instead, compatibility is specified by the architect by writing contractual specifications to make explicit the application properties that he/she would like to capture and verify at the architecture level. Consequently, the description of the architecture contains only application-specific constraints. However, in order to check if an architecture is compatible with a given platform-specific property, platform properties can be also loaded as contractual specifications into the architecture.

Our framework handles the four kinds of specifications uniformly. Contractual specifications are based on the assume-guarantee paradigm. This enables the resolution and the incremental composition of the contractual specifications at design-time. This approach allows the architect to only verify the specifications of the modified part of the architecture and enhances the identification of

inconsistent or missing specifications. Overall our framework provides the architect with some support for global compositional reasoning and contributes thus to the definition of better architectured systems.

In our approach, we have separated the means to describe the architecture, *i.e.*, the meta-models, from the analysis tools. We provide rich meta-models with clear semantics that allow architects to create new analysis tools manipulating our meta-models. This separation enables our approach to be extensible since new analysis tools can be added without modifying the meta-models and new meta-models can be added without impacting the existing meta-models and tools.

In the near future, we plan to use our framework at runtime in order to verify the integrity of the architecture during adaptation. This way, even if the runtime platform does not handle any kind of contract assertion, the dynamic architecture will be contractualized since the verifications will be performed, in a generic way, at the architecture-level.

References

1. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. on Software Engineering 26(1), 70–93 (2000)
2. Nunziati, S.: Personal health record, `http://www.d-m-p.org/docs/EnglishVersionDMP.pdf`
3. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (2004)
4. Barais, O., Lawall, J., Le Meur, A.F., Duchien, L.: Safe integration of new concerns in a software architecture. In: Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS 2006), pp. 52–64. IEEE Computer Society, Los Alamitos (2006)
5. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
6. OMG: CORBA Component Model, v4.0, formal/06-04-01 (April 2006)
7. BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase: Assembly Component Architecture - Assembly Model Specification Version 1.00. (March 2007)
8. Allen, R.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon, School of Computer Science, CMU-CS-97-144 (January 1997)
9. Monroe, R.T.: Capturing Software Architecture Design Expertise with Armani (January 2001)
10. Kalibera, T., Tůma, P.: Distributed component system based on architecture description: The SOFA experience. In: CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519. pp. 981–994. Springer, Heidelberg (2002)
11. Collet, P., Rousseau, R.: Efficient Implementation Techniques for Advanced Assertion Languages. RSTI - Série L'Objet (RSTI-Objet) 5(3-4), 417–442 (1999)
12. IBM: WSLA Language Specification, V1.0. (2003)
13. OMG: Object Constraint Language (OCL). 2.0 edn. (May 2006)
14. Abadi, M., Lamport, L.: Composing specifications. ACM 15(1), 73–132 (1993)

15. Frolund, S., Koisten, J.: QML: A Language for Quality of Service Specification (1998)
16. Kildall, G.: A unified approach to global program optimization. In: 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973)
17. Xiangpeng, Z., Chao, C., Hongli, Y., Zongyan, Q.: A qos view of web service choreography. In: IEEE International Conference on e-Business Engineering, pp. 607–611 (2007)
18. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis & Design Language (AADL). SAE Standards nAS5506 (November 2004)
19. Zelesnik, G.: The UniCon Language Reference Manual (May 1996)
20. Roshandel, R., Medvidovic, N.: Multi-view software component modeling for dependability. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 286–304. Springer, Heidelberg (2004)
21. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
22. Dashofy, E.M., Van der Hoek, A.V., Taylor, R.N.: A highly-extensible, XML-based architecture description language. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). IEEE Computer Society Press, Los Alamitos (2001)
23. Waignier, G., Le Meur, A.F., Duchien, L.: Fiesta: A generic framework for integrating new functionalities into software architectures. International Journal of Cooperative Information Systems (IJCIS) 16(3/4), 367–391 (2007)
24. Van der Aalst, W.M., Beisiegel, M., Van Hee, K.M., König, D., Stahl, C.: An soa-based architecture framework. International Journal of Business Process Integration and Management 2(2), 91–101 (2007)
25. Magee, J.: Behavioral analysis of software architecture using ltsa. In: Proceedings of the 21st international conference on Software engineering, pp. 634–637. IEEE Computer Society, Los Alamitos (1999)
26. OMG: Business Process Model and Notation (BPMN) 2.0. (June 2007)
27. OASIS: Web Services Business Process Execution Language Version 2.0. (April 2007)
28. Jung, H., Rubio-Medrano, C.E., Wong, W.E., Cheon, Y.: Architectural Assertions: Checking Architecrural Constraints at Run-Time
29. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Journal of Information and Software Technology 49(1), 12–31 (2007)

Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant

Andres Diaz-Pace, Hyunwoo Kim, Len Bass, Phil Bianco, and Felix Bachmann

Software Engineering Institute, Carnegie Mellon University
4500 Fifth Avenue, Pittsburgh, PA-15213-2612, USA
{adiaz,hkim,ljb,pbianco,fb}@sei.cmu.edu

Abstract. Techniques and tools for specific quality-attribute issues are becoming a mainstream in architecture design. This approach is practical for evaluating the architecture in early stages but also for planning improvements for it. Thus, we believe that one challenge is the integration of the individual capabilities of quality-attribute techniques. This paper presents our research work on a design assistant called *ArchE* that, based on reasoning framework technology, provides an infrastructure for third-party researchers to integrate their own quality-attribute models. This infrastructure aims at facilitating the experimentation and sharing of quality-attribute knowledge in both research and educational contexts.

Keywords: Architecture-based analysis & design, quality attributes, design assistance, *ArchE*.

1 Introduction

The importance of tackling quality-attribute requirements (e.g., performance, modifiability, reliability and other "non-functional" issues) in early development stages has been widely recognized by the software community. The software architecture is an effective instrument to reason about the relationships between design decisions and quality attributes [4].

One mechanism for modeling quality-attribute issues is via reasoning frameworks. A *reasoning framework* [5] is an abstraction to encapsulate the knowledge needed to understand and estimate the behavior of a system with respect to a particular quality, so that this knowledge can be applied by non-experts. Having encapsulated models for quality attributes has advantages in terms of scale and level of detail, because it helps the architect to manage the relationships among multiple quality-attribute models when designing an architecture. Ideas of the same kind have been discussed by other researchers as well [7, 10, 14].

In this context, automated tool support is crucial to take advantage of quality-attribute knowledge. A particular category of tools is design assistants. A *design assistant* can be seen as an agent that supports the architect in decision-making, either by making suggestions on possible courses of action or by performing some computations autonomously on her behalf. For several years, the Software Engineering Institute

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 171–188, 2008.
© Springer-Verlag Berlin Heidelberg 2008

(SEI) has been developing an assistant for architecture design called *ArchE*[1] [1, 2, 16]. In a nutshell, this prototype performs a semi-automated search of the design space, using the outputs of reasoning frameworks to direct the search towards solutions with known quality properties. The initial release of *ArchE* consisted of a rule-based engine and examples of reasoning frameworks that allow the user to explore simple architectures for performance and modifiability.

However, the challenge is not only about sound reasoning frameworks able to link architectures to quality-attribute models individually. In order to fully realize the potential of this technology, we argue that a design assistant should allow people to put their own reasoning frameworks to work together. In this paper we describe an extension of *ArchE* called *ArchE Reasoning Framework Interface (ArchE-RF Interface)* to support such an objective. This new release consists of a collaborative infrastructure for third parties to contribute reasoning frameworks to *ArchE* as plugin modules. The approach is based on a blackboard organizational style, in which the *ArchE* engine plays the role of control component and the reasoning frameworks register themselves with *ArchE* through a publish-subscribe schema. *ArchE* has no semantic knowledge of quality-attribute models; it just manages the basic inputs such as scenarios and responsibilities, delegates the design work to the available reasoning frameworks, and then assembles their results.

The contribution of this approach is that a researcher can concentrate directly on the modeling and implementation of a reasoning framework for her quality of interest, and afterwards instantiate her reasoning framework easily on top of the *ArchE-RF Interface*. Furthermore, providing a platform for modular reasoning frameworks that are *ArchE*-compatible, we expect to support the development and integrated use of quality-attribute models by researchers, practitioners and educators.

The rest of the paper is structured around 5 sections. Section 2 describes the key concepts of the *ArchE* vocabulary for reasoning frameworks. Section 3 is devoted to the interactions between *ArchE* and the reasoning framework plugins using the *ArchE-RF Interface*. Section 4 briefly describes our experiences implementing two reasoning framework examples. Section 5 comments on related work. Finally, Section 6 presents the conclusions and discusses future lines of work.

2 Reasoning Frameworks: The Building Blocks

Conceptually, a reasoning framework is a modular entity that provides the capability to reason about specific quality-attribute behavior(s) of an architecture. In its original formulation [5], a reasoning framework only involved analytic theories (e.g., queuing networks for performance, change impact for modifiability, Markov chains for availability, etc.) to determine whether an architecture satisfies quality-attribute requirements. Later, this formulation was extended with the capability to transform an architecture using tactics [2] in order to satisfy unmet quality-attribute requirements.

The class of behaviors or situations for which the reasoning framework is useful is referred to as the problem description. A specification of a problem description can be a collection of scenarios along with an initial architectural model for the system. The

[1] http://www.sei.cmu.edu/architecture/arche.html

analytic theory needs also a representation to abstractly describe those aspects of the design we should reason about. This representation is referred to as the analysis model. In this context, a reasoning framework is expected to support three phases [2]:

1. **Interpretation:** The mapping procedure that converts the architectural model into the analysis model
2. **Evaluation:** The procedure used to solve the analysis model and compute quality-attribute measures for the scenarios. These measures help to determine whether the current architecture satisfies its scenarios.
3. **Re-design (optional):** In case some scenarios are unmet, tactics permit to adjust the structure/behavior of the current architectural model.

To accomplish these phases, the process of building a reasoning framework relies on a vocabulary of architectural concepts. The key concepts we have used for the development of *ArchE-RF Interface* include: general quality-attribute scenarios, concrete quality-attribute scenarios, quality-attribute models, responsibilities, architectural tactics, and architectural views. Figure 1 shows the ontology of concepts and the relationships among them.

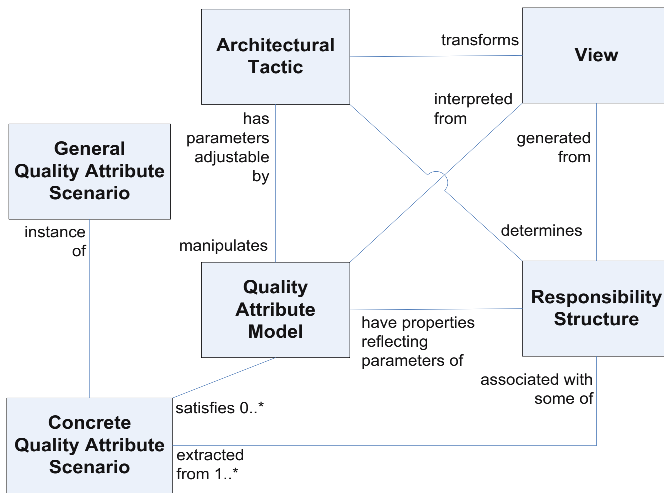


Fig. 1. Ontology of architectural concepts for reasoning frameworks

A summary of the concepts is provided below (see references for further details).

- *General quality-attribute scenario*. A system-independent table for deriving quality-attribute requirements. The table consists of six parts, namely: a stimulus, a stimulus source, an environment, an artifact being stimulated, a response, and a response measure; each part having different possible values. General scenarios for several quality attributes are discussed in [4].
- *Concrete quality-attribute scenario.* A system-specific requirement that is an instance of a general scenario. A concrete scenario for modifiability would look like "The operating system used by different customers may vary (stimulus).

Adaptation of the software to the different processors (response) should be done within 1 person-day (response measure)".

- *Quality-attribute model.* The result of interpreting an architecture design with an analytic theory. A quality-attribute model usually has a set of independent parameters that can be manipulated (in specific reasoning framework instances) to control the values of the measures produced by the evaluation procedure. See the chain impact analysis theory described in Section 2.1 for an example.
- *Responsibilities.* A responsibility is an activity undertaken by the software being designed [18]. We use responsibilities as a means to express functional requirements as a part of quality-attribute scenarios, and moreover, as a means to integrate the models produced by various reasoning frameworks. Responsibilities can be annotated with quality-specific properties or take part in relationships. All this information provides clues for a reasoning framework to create an initial architecture and reason about quality-attribute issues. See example of Section 2.1.
- *Architectural tactic.* A vehicle for satisfying a quality-attribute-response measure by manipulating some aspect of a quality-attribute model through design decisions. That is, a tactic is an architectural transformation based on a quality-attribute justification. A tactic comes with both analysis rules and design rules. The former rules specify how the independent parameters of a quality-attribute model can be controlled to achieve a desired measure (i.e., a scenario response). The latter rules codify architectural decisions to move from a given architecture to another one (variant) with a better fitness. See example of Section 2.1.
- *Architectural view.* A view is a design structure of the system that can be seen from a viewpoint [4]. In general, an architectural view can be seen as a typed graph that is composed of architectural design elements, their properties, and their relations for the viewpoint. Examples of common architectural views are: the module view, the process view, the component-and-connector view, etc.

Note that the ontology involves three types of model transformations. A first type of transformation generates the architectural model (i.e., a set of architectural views) from the scenarios and responsibilities. Then, a second type of transformation is the interpretation procedure, which translates the views to a representation that is more suitable for quality-attribute analysis. Finally, a third type of transformation is that of tactics, which modifies the current architectural view(s) to generate architectural variants. Here, it is assumed also that the tactics determine responsibilities and relationships for the architecture, which are consistent with the quality-attribute models manipulated in terms of its parameters.

In addition, we require every reasoning framework to publish a *manifesto*. This manifesto is used by *ArchE* to integrate the reasoning framework to the infrastructure, checking compliance of its modeling concepts and detecting possible conflicts with other reasoning frameworks. The manifesto specifies the quality attribute the reasoning framework is interested in, the scenario structure, and other architectural element types that the reasoning framework is able to process.

2.1 Example: A Modifiability Reasoning Framework

We briefly describe a modifiability reasoning framework based on change impact analysis (CIA) [6], as an example of the kind of quality-attribute models that can be integrated in *ArchE* [1][2]. Modifiability is seen as "the ability of a software architecture to accommodate changes". Given a set of change scenarios, the level of modifiability of an architecture is a function of how functionality is allocated to modules and how these modules interact with each other.

According to the CIA theory, the architecture is interpreted as a graph, in which the nodes correspond to "units of change" (e.g., responsibilities, modules, interfaces) while the arcs represent dependencies between the nodes (e.g., functional dependencies, data flows). A modification of a specific node is likely to propagate to a neighborhood of nodes. We assume that the effects of the change in the neighbors decrease as a function of the distance to the source of the change. So, we define an evaluation procedure that traverses the graph and returns cost estimations for the change. To do this evaluation, nodes and arcs are annotated with properties. The total cost of making a change is computed as a weighted sum that considers the costs of individual nodes and the probabilities of change rippling associated to the arcs. Furthermore, we allow manipulation of the graph via tactics, so as to affect the results of the evaluation function. This is accomplished either by adjusting the values of properties or by altering the topology of the graph.

Figure 2 outlines the manifesto for our modifiability reasoning framework. This manifesto is an XML file that lists the element types handled by the reasoning framework. The manifesto exposes structural information of the element types, but it is not concerned with their behavior. The first part of the manifesto identifies the reasoning framework itself (tag *<rf>)*. For CIA, the manifesto specifies a new type of modifiability scenario (section *<scenarioTypes>*) as well as modifiability-related elements for it (e.g., sections for responsibility parameters, responsibility relationship types, view element types, view relation types, etc.). *ArchE* will use this specification as "meta-information" of what is needed by the reasoning framework to operate. Additionally, *ArchE* will display appropriate GUIs and infer the data mappings to its database.

In the *<responsibilityStructure>* section, we specify that a responsibility can take part in a "functional dependency" relationship with other responsibilities. Besides, we decorate plain responsibilities and dependency relationships with modifiability-specific parameters. One parameter of a responsibility is the cost of changing that responsibility. Two parameters of a dependency are the probabilities for "incoming" and "outgoing" rippling of changes. The assignment of values to these parameters is done by the architect based on previous experiences or empirical data.

The *<view>* section specifies a module view [4] as a suitable architectural description for modifiability issues. A module can be thought of as a code or implementation unit that delivers some functionality. Modules have relationships with other modules. A common relationship between modules is dependency, which denotes coupling between two modules. Since *ArchE* relies on responsibilities, we have extended the module view to include allocation relationships, so that a module can support one or more responsibilities. Dependencies between modules are computed in terms of

[2] Although the CIA-based model is plausible to reason about modifiability, the model has not been fully validated yet.

responsibility dependencies and responsibility allocations. That is, if a responsibility A is dependent on a responsibility B and they are allocated to different modules MA and MB respectively, we will have then a dependency between modules MA and MB. The dependency relationship for modules behaves similarly to the responsibility dependency, having associated probabilities for incoming and outgoing change rippling. The <*model*> section is about the representation of the graph in terms of units of change and rippling probabilities. This section is optional in the manifesto, and it only serves to visualization purposes of the *ArchE* GUI.

```
<!--xml header -->
<rf   <!-- Reasoning framework identification -->
      id="ChangeImpactModifiabilityRF"            <!-- Unique ID -->
      quality="Modifiability"                     <!--Target quality attribute -->
      name="ModifChangeImpact  RF v0.1"           <!--Description -->
      version="0.1"                               <!-- Version of this reasoning framework -->
>
 <scenarioTypes> <!-- Specification of 6-part general scenario -->
 . . .
 </scenarioTypes>
 <responsibilityStructure > <!-- Information about responsibility parameters,  types of responsibility
relations  and parameters for those relationships,  e.g., dependency relationship, cost of change or
rippling properties -->
    <parameterTypes> . . . </parameterTypes>
    <responsibilityParameters> . . . </ responsibilityParameters >
    <relationshipTypes> . . . </ relationshipTypes >
 </responsibilityStructure >
 <view > <!-- Description of the design elements and relationships used in the architectural
representation, e.g., a module view-->
    <viewElementType> . . . </ viewElementType >
    <viewRelationType> . . . </ viewRelationType >
 . . .
 </view >
 <model > <!-- Description of elements and relationships of the model used  for quality- attribute
analysis, e.g., a dependency graph -->
    <modelElementType> . . . </ modelElementType >
    <modelRelationType> . . . </ modelRelationType >
 . . .
 </model >
</rf>
```

Fig. 2. Fragment of the XML manifesto

When the reasoning framework executes, its interpretation procedure will filter out those design elements and design relations of the module view that are related to scenario-specific responsibilities, in order to construct a graph for the architecture. This graph will be evaluated according to a cost formula. We used a cost formula derived from [1] for computing the cost of all the nodes impacted by a given scenario. The interpretation and evaluation are graphically exemplified in Figure 3. Finally, the design cycle is completed with two modifiability tactics [3], which are not included in the manifesto but supported by the reasoning framework implementation. The first tactic aims at reducing the cost of modifying a single (costly) responsibility by splitting it into children responsibilities. An instance of this tactic is shown at the bottom of Figure 3. The second tactic aims at reducing the coupling between modules by inserting an intermediary that breaks module dependencies. These tactics are materialized through

transformations that affect both the module view and the responsibility structure. The re-interpretation of the architectures generated by the transformations leads to slightly different dependency graphs, and consequently, the modifiability measures for these graphs vary. The process of *interpretation-evaluation-transformation* continues until the analysis of the scenarios reaches values that satisfy the architect's expectations.

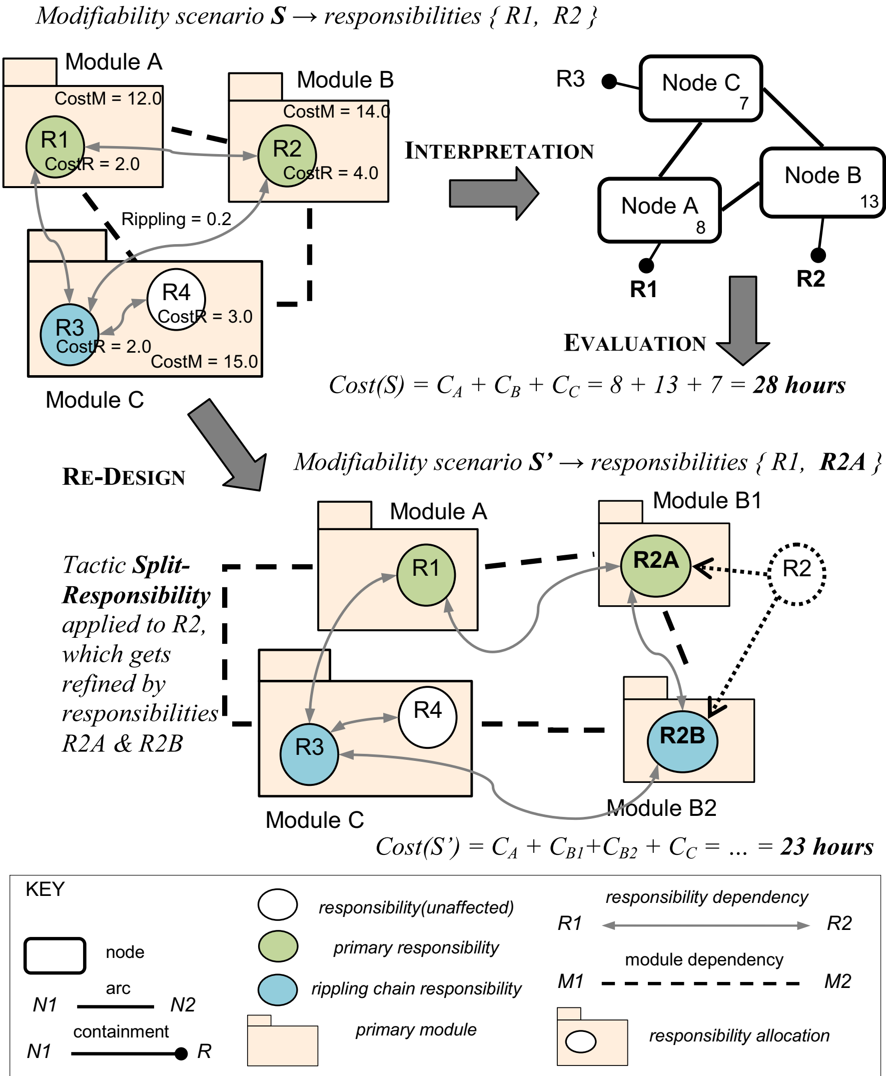


Fig. 3. Interpretation, evaluation and re-design for modifiability

3 ArchE-RF Interface: The Collaborative Infrastructure

The working of *ArchE* follows a blackboard style [8], in which different actors collaborate to produce a solution for a problem. Each actor can potentially read information from the blackboard that was developed by other actors; and conversely, each actor can introduce new information into the blackboard that could be of interest to anyone else. The reasoning frameworks can be seen as knowledge sources, and *ArchE* is the control component that manages the interactions among them, so as to ensure progress in the architecting process. Note that *ArchE* is an assistant to explore quality-driven architectural solutions, rather than being an automated design tool. Since not all the decisions can be made by *ArchE*, the user becomes an additional actor in the schema, who makes the final decisions. For instance, the computations of the reasoning frameworks need human intervention for specifying correct scenarios, entering the necessary parameters for analysis and tactics, among other tasks. This modality of assistance is known as *mixed-initiative* [17].

Enhancing the assistive capabilities of *ArchE* means to integrate different reasoning frameworks into the blackboard schema. To do so, we have re-designed the initial version of *ArchE* towards a collaborative infrastructure: the *ArchE Reasoning Framework Interface (ArchE-RF Interface)*. In this infrastructure, reasoning frameworks are considered as "external plugins". The term "external" means that a reasoning framework resides anywhere outside the *ArchE* process, even on a remote machine over networks. The term "plugin" means that a reasoning framework can be added or removed at runtime without disturbing the current tasks of *ArchE*. Thus, *ArchE* can take advantage of multiple computing resources by executing reasoning frameworks in parallel.

In Figure 4, we show a simplified view of the interactions between *ArchE* and the reasoning frameworks. A reasoning framework announces itself in the infrastructure via its manifesto, and *ArchE* enables the reasoning framework for operation. From that point on, the *ArchE* engine starts sending asynchronous interaction commands to the reasoning framework(s), and also communicating information through a database. Meanwhile, each reasoning framework acts as a "command listener", executing the received commands with its own logic and accessing the database. Once a reasoning framework has successfully executed a command, it sends the results back to *ArchE*. Examples of command results can be: analysis values, suggested tactics, or questions for the user. *ArchE* either waits for the results of a predefined command or proceeds with other commands, depending on the context.

The collaborative infrastructure relies on four main components:

- *ArchE Engine*. This component retains the functionality of the first release with respect to the general structure of the search for architectural alternatives. The only modification is that the design work is now delegated to "remote" reasoning frameworks. This engine has very little knowledge of either quality-attribute design techniques or semantics of the system being designed. The responsibilities of the engine are: processing of user inputs, update of GUI panels, parsing of the manifesto, coordination of reasoning frameworks, presentation of their results, and display of user questions.

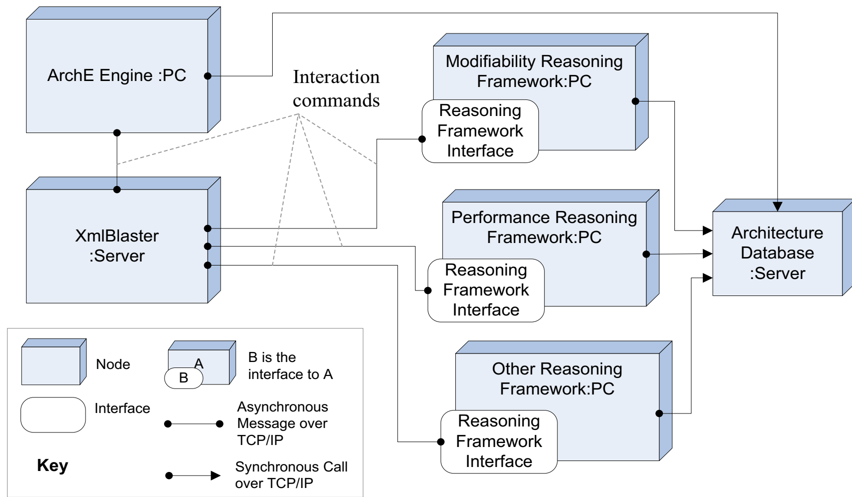


Fig. 4. Integration of external reasoning frameworks with the ArchE engine

- *XmlBlaster*[3]. This is a message-oriented middleware where implicit message invocations can take place among participants over networks. This middleware fosters extensibility in terms of adding (or removing) a participant without considering others.
- *Reasoning Framework Interface*. This is the actual interface to a reasoning framework. It abstracts the details about working with *XmlBlaster*, the communication protocol between *ArchE* and the reasoning framework, and also the algorithms executing the interaction commands.
- *Architecture Database*. This repository is used to manage any persistent data that need to be shared by *ArchE* and all participating reasoning frameworks. The data include both the original and the candidate architectures (e.g., scenarios, responsibilities, architectural views, and relationships among them).

The *ArchE-RF Interface* is implemented in Java, so the reasoning framework functionality must be implemented in Java as well. Anyway, given the *XMLBlaster* characteristics, the functionality could be implemented in other programming language (e.g., C or C++) and then assembled with the top-level Java code using JNI[4].

3.1 ArchE Interaction Commands

Basically, *ArchE* runs a search algorithm for finding promising candidate architectures. The search is divided between the *ArchE* engine and the available reasoning frameworks. On one side, the engine controls the main search cycle and makes a global evaluation of the proposals of the reasoning frameworks. On the other side, each reasoning framework should implement its own search algorithms to suggest tactics for the current architecture.

[3] http://www.xmlblaster.org/
[4] http://java.sun.com/javase/6/docs/technotes/guides/jni/

The search cycle is structured around five commands that govern the interactions with the reasoning frameworks.

- *ApplyTactics.* This command requests a specific reasoning framework to apply a tactic to the current architecture in order to refine it (*Re-design* phase). The tactic must come from a question that was previously shown to the user of *ArchE* and she agreed to apply (see command *DescribeTactic* below). The expected result is to have the refined version of the current architecture in the database.
- *AnalyzeAndSuggest.* This command requests a reasoning framework to analyze the current architecture regarding scenarios of interest, and to suggest new tactics if some scenarios are not fulfilled (*Interpretation* and *Evaluation* phases). The reasoning framework returns the analysis results and the tactics (if any) to *ArchE.*
- *ApplySuggestedTactic*. This command requests a reasoning framework to apply a tactic to the current architecture in order to create a new candidate architecture (*Re-design* phase on a new architecture instance). The tactic must be one of the tactics that the reasoning framework suggested when executing the *AnalyzeAnd-Suggest* command. The expected result is to have a candidate architecture in the database.
- *Analyze*. This command requests a reasoning framework to analyze a candidate architecture regarding scenarios of interest (*Interpretation* and *Evaluation* phases on a new architecture instance). The evaluation results returned by the reasoning framework will be used by *ArchE* to prioritize candidate architectures.
- *DescribeTactic*. This command requests a reasoning framework to provide *ArchE* with user-friendly questions that describe tactics or any other recommendations. This is actually the main mechanism to offer design advice to the user on how to improve its architecture. Again, *ArchE* does not know about the semantics of user questions, it just shows these questions in the GUI and let the user decide.

Whenever the user makes a change to some part of the design, *ArchE* starts a new cycle of its algorithm and executes the above commands in the following sequence:

1. If the change is a decision to apply a tactic, *ArchE* sends *ApplyTactics* to the reasoning framework that suggested the tactic, and then, the reasoning framework modifies the working architecture according to the tactic. For example, let's consider that our modifiability framework inserts an intermediary module upon user's request.
2. For every reasoning framework, *ArchE* sends *AnalyzeAndSuggest* sequentially. Each reasoning framework might modify the current architecture (if needed), in preparation for the following analysis task. This assures consistency on the responsibility structure and initialization of its architectural view. For example, our reasoning framework can decorate new responsibilities with costs (if that property is missing) and update the module view by allocating every new responsibility to a module. Then, each reasoning framework starts its analysis of the architecture. If the analysis results say that some scenarios are not fulfilled, it tries to find tactics suitable for the architecture. At last, it returns the analysis results and the list of suggested tactics. For instance, our reasoning framework may run its change impact analysis, detect a costly responsibility as a main contributor to the scenario response (total cost), and propose a responsibility splitting.

3. For every suggested tactic:
   a) *ArchE* sends *ApplySuggestedTactic* to the reasoning framework with the tactic under consideration. The reasoning framework creates a candidate architecture by modifying the architecture according to the tactic.
   b) For every reasoning framework, *ArchE* sends *Analyze* in parallel. Each framework analyzes the candidate architecture and returns the evaluation results to *ArchE*.
4. *ArchE* prioritizes all the evaluation results that came from applying suggested tactics. This ranking of evaluation results is displayed as a matrix of scenarios versus tactics called "traffic light". For every reasoning framework, *ArchE* sends *DescribeTactic* in parallel. Each reasoning framework provides *ArchE* with questions that describe suggested tactics (if applicable). For example, our reasoning framework would ask the user to apply the tactic of splitting on a particular responsibility, in order to satisfy a modifiability scenario.
5. *ArchE* shows to the user all the questions sent by reasoning frameworks. The cycle goes back to step 1.

3.2 Governing Reasoning Frameworks

When implementing the *ArchE-RF Interface*, a reasoning framework is expected to support six basic functionalities, which will hook into the search cycle described above. The functionalities are:

- Self Description (manifesto)
- Creating Initial Design
- Analyzing (for commands *Analyze* and *AnalyzeAndSuggest*),
- Suggesting Tactics (for command *AnalyzeAndSuggest*)
- Applying Tactics (for commands *ApplyTactic* and *ApplySuggestedTactic*)
- Describing Tactics (for command *DescribeTactic*)

ArchE does not require a reasoning framework to implement all the functionalities, but at least *Self Description* must be implemented to enable communication with *ArchE*. The implementation of the remaining functionalities is up to the researcher, depending on the type of reasoning framework wanted. The *Analyzing* functionality is generally present in any reasoning framework. For example, if we build our modifiability reasoning framework just to apply CIA on the module view, we can implement the *Analyzing* and *Creating Initial Design* parts and ignore other functionalities. However, if we would like our reasoning framework to be able to alter the architecture (after performing analysis), then we also need to implement the functionalities of *Suggesting Tactics*, *Applying Tactics* and *Describing Tactics*.

In addition to a command-based interface for interacting with *ArchE*, the *ArchE-RF Interface API* provides guidelines to implement the reasoning framework internals. These guidelines can be seen as a small object-oriented framework [11] that predefines the overall design of a plugin, its decomposition into Java interfaces and classes, the main methods to be overridden, and the general flow of control derived from the interaction commands. These features significantly reduce the design decisions that have to be made by a researcher when creating plugins for *ArchE*.

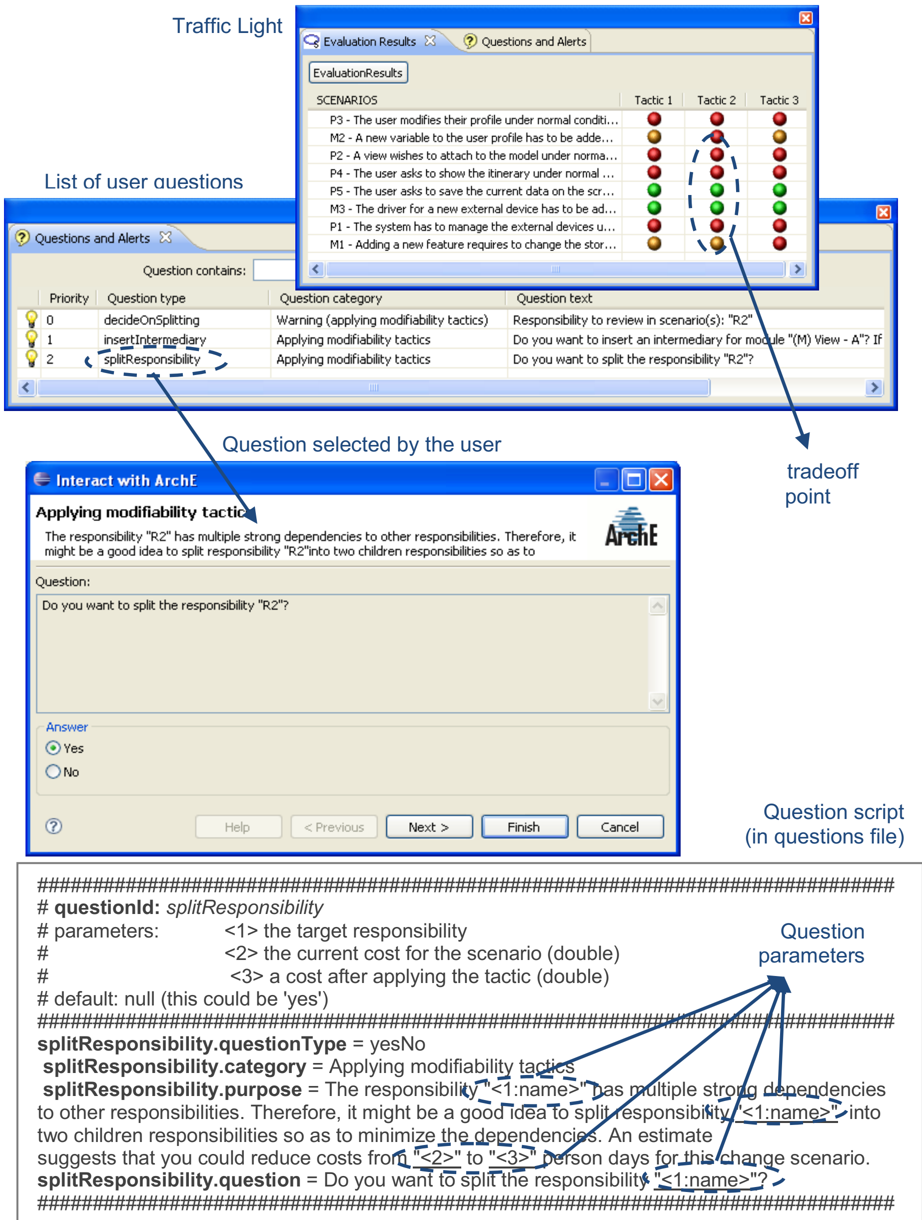
The *ArchE-RF Interface API* is structured into four layers. Each layer provides services for the upper layers, although there is no strict layering.

- *Communication layer.* It is the top-level layer that includes all the classes and interfaces related to interacting with *ArchE* via the *XmlBlaster*. It provides functionalities such as: registration of a reasoning framework with *ArchE* at runtime; reception of an interaction command from the *XmlBlaster* and delegation of its execution to the *Execution* layer; communication of progress messages and notice of command cancellations.
- *Execution layer.* It is equipped with a set of algorithms, each processing a different interaction command as forwarded from the *Communication* layer. Based on the services from the two layers below it, the *Execution* layer provides functionalities such as: restoring, saving and deletion of the architecture in the *ArchE Database*; exception handling, etc.
- *Reasoning Framework layer*. It provides the *ArchEReasoningFramework* class, which has to be extended by a researcher in order to implement a specific reasoning framework. It also provides other helping classes that she may use to handle inputs and outputs for an interaction command.
- *Data layer.* It is the bottom-level layer that provides the upper layers with the concepts shown in Figure 1. It includes the Java interfaces needed to manage the key concepts, which must be mapped to concrete classes and database tables.

3.3 Interaction with the User

The user gets to know about the reasoning framework proposals for the current architecture through two GUI mechanisms: the "traffic-light" metaphor and the user questions. Figure 5 shows a traffic light snapshot for modifiability and performance scenarios, along with potential scenario improvements when applying different tactics. The columns display color-coded ball icons that represent the tactics being evaluated by *ArchE*. A green ball indicates that the scenario will be satisfied if that tactic is applied, while a red ball indicates that the scenario will not be satisfied. Note also how the effects of the tactics on the scenarios lead to quality-attribute tradeoffs.

The snapshot below the traffic light shows a list of user questions. Typically, a question describes the purpose of a particular tactic. For instance, Figure 5 displays a question dialog for the tactic of splitting a costly responsibility. If the user enters a positive answer, then *ArchE* will trigger the corresponding architectural transformation. The types of questions associated to a reasoning framework must be specified by the reasoning framework developer in a questions file that supplements the manifesto. This questions file let *ArchE* know about the template and parameters of each possible question. The bottom part of Figure 5 shows how the question scripts look like. When the *ArchE* engine invokes the *DescribeTactic* command and the reasoning framework returns a question instance, *ArchE* loads its associated template and substitutes the placeholders of the text with specific question parameters. The *ArchE* GUI uses that information to display the question by means of predefined graphical widgets. Once the user picks and answers a particular question, *ArchE* translates the results into an *ApplyTactic* command for the reasoning framework that provided that question.




```
##########################################################################
# questionId: splitResponsibility
# parameters:        <1> the target responsibility
#                    <2> the current cost for the scenario (double)
#                    <3> a cost after applying the tactic (double)
# default: null (this could be 'yes')
##########################################################################
splitResponsibility.questionType = yesNo
 splitResponsibility.category = Applying modifiability tactics
 splitResponsibility.purpose = The responsibility "<1:name>" has multiple strong dependencies
to other responsibilities. Therefore, it might be a good idea to split responsibility "<1:name>" into
two children responsibilities so as to minimize the dependencies. An estimate
suggests that you could reduce costs from "<2>" to "<3>" person days for this change scenario.
splitResponsibility.question = Do you want to split the responsibility "<1:name>"?
##########################################################################
```

Fig. 5. Configuration and visualization of tactics in *ArchE*

4 Implemented Reasoning Frameworks and Lessons Learned

Currently, we have created two reasoning framework plugins using the *ArchE-RF Interface*. The first plugin is a full-fledged reasoning framework for modifiability (as outlined in sub-section 2.1), which served to test and tune the infrastructure. The

second plugin is a reasoning framework for real-time performance that takes advantage of an existing analytic solver called MAST[5]. MAST [12, 15] is a toolset for describing event-driven real-time systems and performing schedulability analysis. Figure 6 shows a snapshot of *ArchE* running the two plugins. In general, validating reasoning frameworks with respect to the scope and accuracy of their predictions is the job of the reasoning framework developer and not a portion of *ArchE*.

After writing its manifesto, the modifiability reasoning framework was implemented from scratch in Java. Initially, we defined subclasses for the responsibility dependencies and the responsibility structure. We also created a class to represent the module view. Then, we implemented a subclass of the *ArchEReasoningFramework* class that encapsulates the interpretation and the formulae for computing various metrics such as cost, coupling and cohesion. On this basis, we codified rules that looked at the values of these metrics to configure possible tactics. Finally, we equipped the reasoning framework with architectural transformations for the tactics, and we also wrote the corresponding questions file.

The performance reasoning framework was conceived as an "analyzer" with no support for tactics. The implementation steps were similar to the ones carried out for the modifiability plugin, except that we wrapped the MAST solver to supply the *Analyze* functionality. The MAST input is an ASCII file that consists of an arrangement of tasks with timing requirements (e.g., latency) and events linking the tasks. A worst-case analyzer processes this specification and outputs the timing behavior of the system. In our *ArchEReasoningFramework* subclass, the *Analyze* implementation converts the performance scenarios and their responsibilities to tasks, considering the responsibility relationships as event reactions between tasks. The task model is sent to a file and fed into the MAST toolset. The worst-case latency results are then compared against the timing requirements to determine the schedulability of the scenarios. We are now working on the addition of a set of performance tactics to this plugin.

The reliance of *ArchE* on reasoning frameworks favors integrability and modular reasoning about quality attributes. One of the research questions here is the extent to which the interactions (i.e., dependencies) among quality-attribute models can be reduced. The implementations above shed light on general issues about these interactions and also exposed some drawbacks of the blackboard approach.

In the current design, dataflow interactions arise because the reasoning frameworks often share (parts of) the architectural representation (e.g., responsibilities, elements of architectural views). Anyway, this architectural representation must be kept consistent at all times. Our plugins shared responsibilities but worked on separate architectural views (i.e., a module view and a task view respectively), and only the modifiability plugin had the capability of modifying the architecture. Because of these factors, the consistency checking was relatively simple. For instance, if a modifiability tactic splits a responsibility that appears in a performance scenario, then the performance reasoning framework is asked to update its task model and run the schedulability analysis again. We believe that a general treatment of opportunistic or harmful types of interactions would require more knowledge about the architectural representation, the effects of tactics or the user's inputs.

---

[5] MAST homepage: http://mast.unican.es/

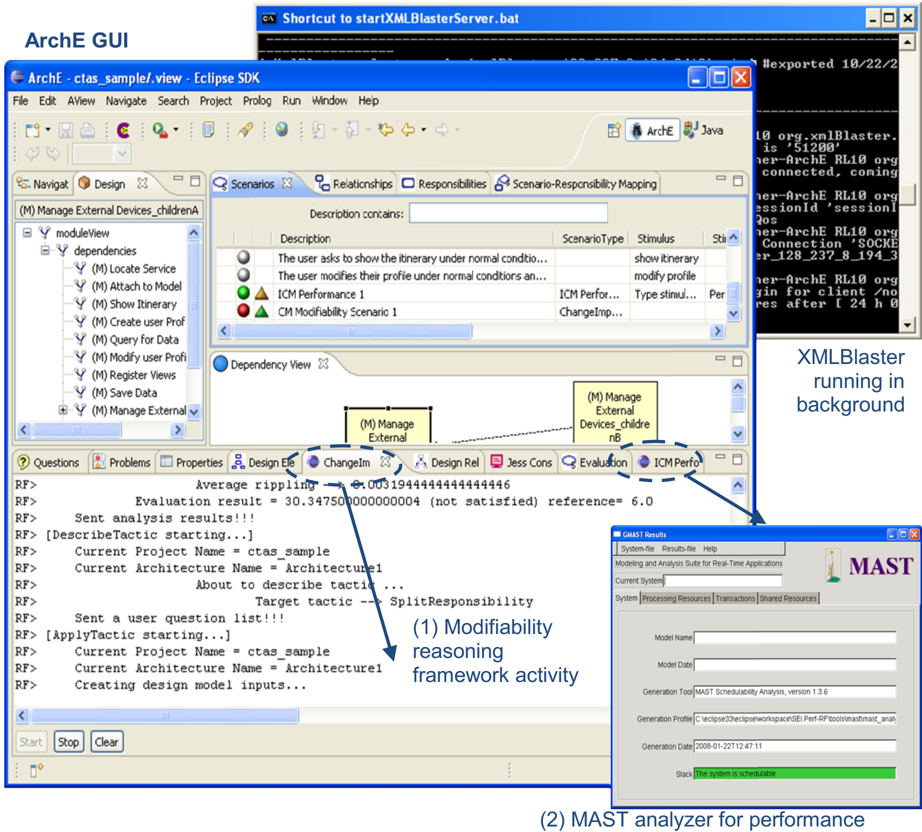


Fig. 6. The *ArchE* prototype executing two reasoning frameworks as plugins

The management of tradeoffs is decoupled into two aspects. The first aspect has to do with the "traffic light" metaphor, so that the user must decide on a tactic making a quality-attribute balance that is good enough for her scenarios of interest. The second aspect comes from the opportunistic/harmful interactions discussed above. A simple source of tradeoffs is the parameters of responsibilities [2]. For instance, when inserting an intermediary due to modifiability reasons, the modifiability reasoning framework can impose a minimum execution time for that responsibility, but this constraint on the execution time parameter later impacts on the schedulability analysis of the performance reasoning framework. Putting mechanisms in place for *ArchE* to support this second aspect of trade-offs is a topic for further research.

Regarding search, each reasoning framework looks locally for tactics that change the architectural structure. However, the resulting architectural transformations do not always guarantee an improvement of the evaluation function, because that function depends on both the architectural configuration and tactic-specific parameters. For instance, when applying the tactic of splitting a responsibility, we must set the costs for the children responsibilities and set the rippling probabilities for their dependencies. Different choices for these values lead to different instances of the same tactic,

some of which reduce the cost of the change and some others do not. The problem of finding an adequate configuration of values for a given tactic is not trivial, and it often needs heuristic search.

We additionally observed some side-effects of the blackboard architecture on usability. A first issue is the processing overhead forced by the main control strategy, because the *ArchE* engine does not know the semantics of the user's actions. A second issue (related to the control strategy) is that the reasoning framework activities for responding to the *ArchE* commands have limited visibility through the GUI. Therefore, while *ArchE* is running, the user can only handle or inspect reasoning framework features at specific points of the exploration process. Future developments should provide a more flexible user-interaction schema.

5 Related Work

The analysis of component-based systems by applying quality-attribute techniques has been an active field of research and technology transfer for many years. Several quality-specific approaches have been developed [7, 10, 14, 15], although few of them have tackled the integration of models and analysis tools. To begin with, the Predictable Assembly from Certifiable Components (PACC) initiative at the SEI has focused on building component-based systems that have predictable behaviors prior to implementation [15]. PACC uses the notion of reasoning frameworks in combination with model checking to analyze performance and safety properties but also to enforce the assumptions required by each analysis technique when applied to the systems. This technology can be applied to predict other properties as well (e.g., reliability, security). As evidenced by the MAST example, we think these techniques can be integrated into *ArchE* with little effort.

The DeSiX approach [7] provides tools for component-based systems on multiprocessor architectures that allow for design space exploration. Here, scenario-based analyses for performance, reliability and cost serve to focus the design on particular architectural configurations. The developer can map usage profiles to simulation tasks, and then visualize the resulting architectures using Pareto curves. When compared to *ArchE*, a drawback of DeSiX is that it does not support automated search, and the developer manually selects configurations to be evaluated by the tool.

Other researchers have proposed a view of software engineering as a search problem [9], in which automation is supported by optimization techniques. Along this line, Grunske [13] has investigated the integration of quality-attribute techniques using genetic algorithms for some experiments involving reliability and cost requirements. Also, he has proposed a generic model for quality-attribute evaluation [14] that contains four elements, namely: encapsulated evaluation models, composition algorithms for these evaluation models, operational/usage profiles, and evaluation algorithms to determine relevant quality measures from the evaluation models. This perspective is similar in spirit to that of reasoning framework, although it does not consider explicitly the aspect of architectural transformations. Nonetheless, Grunske has pointed out challenges for the combined use of quality-attribute models and tool support, such as composability, analyzability and complexity issues.

More recently, Edwards et al. have [10] coined the term "model interpreter" as a vehicle to transform component-based models into analysis models by means of model-driven engineering (MDE) techniques. Consequently, they have developed a "tool-chain" called XTEAM that supports and integrates different types of model interpreters. These interpreters are able to implement transformations between high-level component models (amenable to architectural reasoning) and low-level analysis models (amenable to prediction of component assembly properties). This approach is still experimental and has many analogies with the PACC work, but unlike *ArchE*, it does not seem to focus on the exploration of the design space.

6 Conclusions

In this paper, we have described a tool approach for incentivizing the use of quality-attribute models in architectural design. The *ArchE* approach relies on having a collection of reasoning frameworks that are each specialized for a single quality attribute but that work together in the creation and analysis of architectural designs. *ArchE* is not intended to perform an exhaustive or optimal search in the design space; rather, it is an assistant to the architect that can point out "good directions" in that space. Along this line, the contributions of this work are the encapsulation of quality-attribute knowledge and the tool infrastructure to accommodate that knowledge.

The *ArchE-RF Interface* constitutes an important step towards improving the design of the *ArchE* prototype. Nonetheless, there are issues that need further discussion and implementation efforts. Some of these issues are:

- Incorporation of UML features for architectural modeling, and linking *ArchE* to other development tools.
- Management of tradeoffs between solutions proposed by individual reasoning frameworks, under multiple criteria (e.g., cost, utility, uncertainty).
- Experiments with searching techniques and more powerful solvers (e.g., simulated annealing, planning, SAT, etc.).
- Support for recording design decisions, as an extension of quality-attribute analysis results and tactic proposals.

Finally, we believe that the more reasoning frameworks that are available, the broader the reasoning capabilities of *ArchE* will be. Thus, we hope this work will stimulate researchers, educators and practitioners to plug in and share analysis/design models for various quality attributes, in order to foster architecture-centric practices.

References

1. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Experience Using an Expert System to Assist an Architect in Designing for Modifiability. In: Proceedings 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), Oslo, Norway, p. 281 (2004)
2. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing Software Architectures to Achieve Quality Attribute Requirements. Software IEE 152(4), 153–165 (2005)
3. Bachmann, F., Bass, L., Nord, R.: Modifiability Tactics. Technical report CMU/SEI-2007-TR-002. Software Engineering Institute, Pittsburgh, PA (2007)

4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
5. Bass, L., Ivers, I., Klein, M., Merson, P., Wallnau, K.: Encapsulating Quality Attribute Knowledge. In: Proceedings 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), Pittsburgh, PA, pp. 193–194. IEEE Computer Society, Los Alamitos (2005)
6. Bohner, S., Arnold, R.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
7. Bondarev, E., Chaudron, M., de With, P.: Quality-Oriented Design Space Exploration for Component-Based Architectures. Computer Science Report. University of Technology, Eindhoven, The Netherlands (2006)
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons, Chichester (1996)
9. Clarke, J., Dolado, J., Harman, M., Hierons, R., Jones, R., Lumkinm, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating Software Engineering as a Search Problem. Software IEE 150(3), 161–175 (2003)
10. Edwards, G., Seo, C., Medvidovic, N.: Construction of Analytic Frameworks for Component-Based Architectures. In: Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). Campinas, Sao Paulo, Brazil (2007)
11. Fayad, M., Schmidt, D., Johnson, R. (eds.): Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, Chichester (1999)
12. Gonzalez Harbour, M., Gutierrez García, J.J., Palencia Gutiérrez, J.C., Drake Moyano, J.M.: MAST: Modeling and Analysis Suite for Real Time Applications. In: Proceedings 13th Euromicro Conference on Real-Time Systems (ECRTS), IEEE Comp. Society, Washington (2001)
13. Grunske, L.: Identifying "Good" Architectural Design Alternatives with Multi-Objective Optimization Strategies. In: International Conference on Software Engineering (ICSE), Workshop on Emerging Results, pp. 20–28, 849–852. ACM Shanghai, China (2006)
14. Grunske, L.: Early quality prediction of component-based systems - A generic framework. Journal of Systems and Software 80(5), 678–686 (2007)
15. Ivers, J., Moreno, G.A.: Model-driven development with predictable quality. In: Companion 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion (OOPSLA 2007), Montreal, Quebec, Canada (2007)
16. McGregor, J., Bachmann, F., Bass, L., Bianco, P., Klein, M.: Using an Architecture Reasoning Tool to Teach Software Architecture. In: Proceedings 20th Conference on Software Engineering Education & Training (CSEE&T 2007), pp. 275–282. IEEE Computer Society, Los Alamitos (2007)
17. Wilkins, D., des Jardins, M.: A Call for Knowledge-based Planning. AI Magazine 22(1) (Spring, 2001)
18. Wirfs-Brock, R., McKean, A.: Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, Boston (2003)

Middleware Architecture Evaluation for Dependable Self-managing Systems

Yan Liu[1], Muhammad Ali Babar[2], and Ian Gorton[3]

[1] National ICT Australia, Australia
Jenny.liu@nicta.com.au
[2] Lero, the Irish Software Engineering Centre, University of Limerick, Ireland
malibaba@lero.ie
[3] Pacific Northwest National Laboratory, USA
ian.gorton@pnl.gov

Abstract. Middleware provides infrastructure support for creating dependable software systems. A specific middleware implementation plays a critical role in determining the quality attributes that satisfy a system's dependability requirements. Evaluating a middleware architecture at an early development stage can help to pinpoint critical architectural challenges and optimize design decisions. In this paper, we present a method and its application to evaluate middleware architectures, driven by emerging architecture patterns for developing self-managing systems. Our approach focuses on two key attributes of dependability, reliability and maintainability by means of fault tolerance and fault prevention. We identify the architectural design patterns necessary to build an adaptive self-managing architecture that is capable of preventing or recovering from failures. These architectural patterns and their impacts on quality attributes create the context for middleware evaluation. Our approach is demonstrated by an example application -- failover control of a financial application on an enterprise service bus.

1 Introduction

Dependability is defined as the ability of a system to avoid failures that are more frequent and more severe than is acceptable [3]. It encompasses a set of attributes, including availability, reliability, safety, integrity, and maintainability. Mechanisms for achieving dependability can be categorized into four groups, namely fault prevention, fault tolerance, fault removal and fault forecasting. Dependable software systems are an integral part of many large scale mission critical systems such as financial systems, defence applications, health care and water management systems [10]. Dependability is therefore a key quality attribute in the design of such systems.

Traditional approaches to achieving dependability mostly focus on system design and implementation with a fixed configuration or human-configurable operations. However, these approaches are less effective for systems that are required to be adaptive and responsive to dynamic changes in real-time. For example, not all potential runtime errors and failures can be known during system design. In addition, in many autonomous systems (such as self-organized sensor networks) minimizing the degree of manual reconfiguration is the central doctrine [1]. This has motivated recent

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 189–204, 2008.
© Springer-Verlag Berlin Heidelberg 2008

research efforts on constructing adaptive and self-managing software architecture, which enable automatic problem diagnosis and recovery [10][11].

Adaptive self-managing systems are designed to constantly monitor and reconfigure themselves in response to changing environmental or operational conditions, including errors or failures resulting from unexpected hardware or software faults, network problems, system overloading, and other runtime conditions.

A range of domain specific self-managing strategies and various intelligent mechanisms have been developed for constructing autonomic computing systems. Underlying these mechanisms is a core set of common, domain-independent architectural patterns [16][17], from which best practices can be extracted to form new design patterns for constructing adaptive self-managing architectures. These new design patterns can improve architecture design, and importantly can establish the context for evaluating middleware architectures.

However adaptive self-managing architectures introduce a complicating factor in developing a dependable system. This is because an adaptive architecture requires not only suitable architectural patterns but also infrastructure mechanisms to incorporate the intelligence needed for self-healing or self-configuration. Both demand support from the underlying infrastructure, including the middleware. This tight coupling of middleware and the self-managing architecture running on top of it introduces complexity in managing dependability in an adaptive manner. Therefore, evaluating middleware architectures can provide insights into the design and development of an implementation to meet dependability requirements.

Most architecture evaluation methods [1][2] do not explicitly evaluate the middleware to be used in a particular implementation; rather middleware is either considered at the implementation level or the deployment level. Nor do these methods explicitly consider the design patterns used in a proposed architecture. However, design patterns are an integral part of middleware evaluation practices [6]. Hence, design patterns should be considered as explicit factors in architecture evaluation methods. To this end, we have developed a Method for Evaluating Middleware architectureS (MEMS) [12], which measures middleware architectures by rating multiple quality attributes. The output of MEMS helps to determine the suitability of alternative middleware architectures to meet an application's quality goals. However one limitation of MEMS is that it does not consider design patterns in the specific context of self-managing architectures.

We have therefore made substantial extensions to MEMS for supporting pattern-based middleware architectures for dependable self-managing systems. The extended version of MEMS fulfils the basic requirements, including:

- Capturing key self-managing scenarios in additional to functional scenarios;
- Evaluating patterns and their implementations;
- Conducting both qualitative and quantitative evaluations.

In this paper, we present MEMS and apply it to evaluate adaptive self-managing middleware architecture. We focus on two attributes of dependability, namely reliability and maintainability. In particular, we are interested in the means of fault prevention and fault tolerance of these two attributes. We identify the architectural design patterns necessary to build an adaptive self-managing architecture that is capable of fault tolerance and fault prevention. The two patterns studied in this paper are the

M-A-P-E pattern and policy point pattern. They create an application context for middleware evaluation. We then explain different aspects of MEMS by applying it to evaluate Mule [15], an open source enterprise service bus middleware technology, in supporting adaptive failover control of financial services integrated by Mule.

2 Self-managing Architecture Patterns

A primary concern with self-managing architectures is preserving safety after changes are made to the architecture, behavior and structure of a system [10]. The assumption is that self-managing abilities can help to prevent failure and thus reduce the probability of violating dependability requirements. However, there is the possibility that a poorly designed self-managing mechanism might degrade system dependability in unanticipated ways. The rigor and accuracy of a self-managing strategy plays a key role in attaining dependability. Best practices can inform new design patterns for constructing adaptive self-managing architectures, which are extracted from successful autonomic computing systems that share common architectural patterns [16][17][18]. The use of these patterns creates the context for middleware architecture evaluation, in which patterns can be examined against an individual middleware implementation to evaluate its ability to support a self-managing application.

2.1 M-A-P-E Pattern

The M-A-P-E pattern is derived from IBM's Autonomic Computing architecture blueprint [10][17]. It was designed as reference architecture to construct autonomic systems. M-A-P-E is suitable for adaptive architecture, which is an integral element in autonomic computing. We have applied the template in [6] to describe the pattern, in terms of its implications for dependability attributes.

Context: M-A-P-E stands for the four core functions to organize the structure of an autonomic system, namely *monitor*, *analyze*, *plan* and *execute*. It models the fundamental concepts, constructs and behaviors for building self-managing capability into the environment.

Problem: The creation of self-managing capabilities requires extensible software architectures to provide flexible monitoring and feedback mechanisms, with minimal intrusion.

Solution: The function of a self-managing capability is a control loop that collects details from the system and acts accordingly.

Structure: A control loop interacts with the managed element and consists of tasks for monitoring execution, analyzing collected data, planning the necessary responses based on adaptation policies, and executing actions to enforce the adaptive behavior.

Implementation: M-A-P-E does not specify how the control loop should be implemented. For example, in Java component-based systems, the communication between core entities can be provided by the RMI and JMX protocols [13], while in Web services, the communication can be replaced by SOAP and WSDM protocols [14]. The mechanisms, services and configuration of the underlying middleware utilized are the tactics to realize the architecture.

Variants: There can be several architecture alternatives to implement the M-A-P-E pattern. For example, in a tightly coupled design, the M-A-P-E entities can explicitly invoke one another; while a loosely coupled system can use an intervening layer of abstraction to encapsulate inter-component connection and communication and to separate them from application functionality. This tightly coupled implementation is simpler to implement, but this comes at the cost of flexibility and scalability.

Affected Attributes: The M-A-P-E can be adopted to improve dependability by means of monitoring the system and automatically responding to changes that may result in failures. However, extra processing from M-A-P-E can also introduce overhead, and a poor design can degrade dependability.

2.2 Policy Point Pattern

The policy point pattern is a design pattern that is important in constructing self-managing architectures. A policy is a representation of desired behavior or constraints on behaviors defined in a standard external form. In a self-managing system, a decision to take an action depends on policy specifications. This pattern is derived from QoS control in network systems [1], and can be applied to general policy management in self-managing architectures.

Context: Policies specify the goals of self-managing strategies. A decision to take an action depends on information collected and policy specifications. The policy point pattern models the basic constructs and behaviors for managing and coordinating policies.

Problem: In a self-managing architecture, adaptive behavior is managed by transforming policies into configuration changes and applying those changes to the managed element or resource. When a policy is enforced at a particular point of execution, it is necessary to structure entities or components involved in the policy management. A simple policy-management architecture has been effective in developing QoS control of complex systems [1].

Solution: Policy points are defined as individual entities that are deployable to the managed systems. These entities collect information, make decisions, and enforce policy.

Structure: the policy point pattern consists of three types of policy points: policy decision point (PDP), policy information point (PIP), and policy enforcement point (PEP). A PDP usually needs to refer to more than one policy when making a decision. The information is collected by a PIP and put into a repository, but the PIP does not return any decision. The PDP retrieves information it needs from the repository. A policy enforcement point (PEP) actually performs actions and enforce polices.

Implementation: A PIP, PDP and PEP can be implemented as an *interceptor*. An interceptor is a mechanism to intercept requests, capture execution context and inject actions.

Variants: A self-managing architecture may involve multiple policies. The set of PDP, PIP and PEP can be structured into hierarchical layers. For example, a higher layer PDP may produce decisions depending on multiple PDPs and PIPs from the lower layers.

Affected Attributes: The policy point pattern is complementary to the M-A-P-E pattern. A correctly applied policy point pattern can help structure M-A-P-E entities to attain dependability attributes. However, an incorrect application can be misleading and even incur negative effects on quality attributes including those of dependability.

2.3 Pattern Integration in Self-managing Architecture

The M-A-P-E and policy point patterns can be used together in a self-managing architecture. A set of policy points can help to structure control components and customize an M-A-P-E control loop. Likewise, the policy point pattern relies on those M-A-P-E entities to enable the function of each policy point, such as monitoring changes or detecting internal or external conditions. The policy enforcement is actually performed by the execution entity of the M-A-P-E pattern.

The composition of these patterns can produce various solutions that exert different demands on the underlying middleware infrastructure. This requires that patterns and their implementation should be integral factors in the middleware evaluation context. Moreover, it is also expected that the implementation of these patterns will not degrade the dependability attributes required in highly dependable systems. This imposes extra requirements on an evaluation method to include risk analysis. We depict the relations between patterns, dependability requirements and attributes, metrics and criteria into an evaluation context shown in Figure 1. The arrows in this context indicate dependency between different elements of this context.

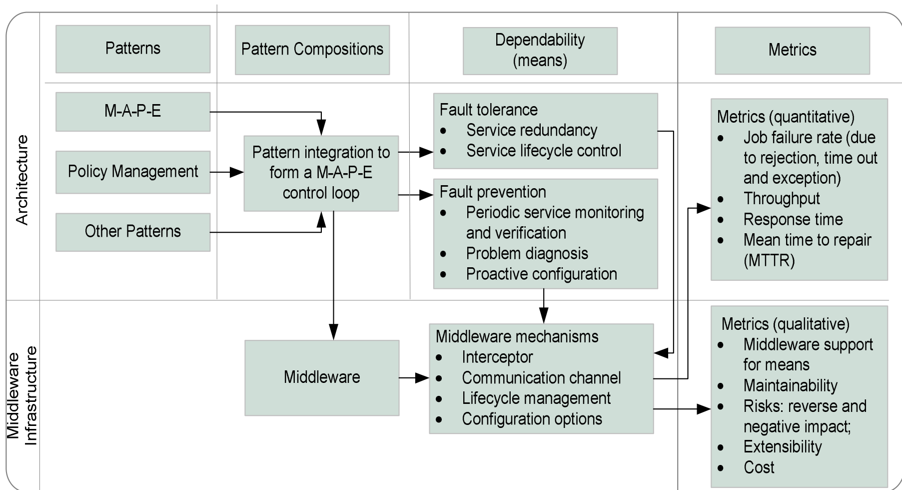


Fig. 1. Middleware Architecture Evaluation Context

3 Evaluation Method

The core of our evaluation method is to treat design patterns as explicit factors for evaluating middleware architectures. Our method to evaluate middleware architectures

has been developed in the context of Figure 1, in which design patterns play a critical role in driving the evaluation and having implication on quality attributes. To achieve this goal, we have extended MEMS [12] to incorporate design patterns. Extended MEMS is a scenario-based architecture evaluation method, consisting of six steps:

1. **Determine quality attributes.** *The outcome provides a set of criteria used by an evaluator to evaluate an architecture. As shown in the metrics column in Figure 1, we focus on reliability and maintainability.*
2. **Generate key scenarios** *to refine the context of reliability and maintainability by means of fault tolerance and fault prevention. The design pattern's description is mapped to the key scenario context.*
3. **Determine pattern alternatives.** *This step has three sub steps.*
   - 3.1. List quality attributes affected by the design pattern. This is achieved by examining the design pattern descriptions through its structure or variants.
   - 3.2. Determine metrics for measurement. The metrics can be either qualitative or quantitative. For example, performance can be described by a quantitative metric such as response times, while programmability is more often evaluated by qualitative metrics.
   - 3.3. Identify alternative implementation. A single pattern can have alternative implementations. The pattern variants can help identify alterative architecture decisions using patterns.
4. **Identify middleware mechanisms,** *which determine the feasibility and efficiency of the architecture solution – how the design patterns can be realized by a middleware.*
5. **Define quality attribute scale,** *which is necessary to evaluate qualitative attributes using a rating scale, such as the level of severity of risks.*
6. **Evaluate quality attributes.** *At this stage, qualitative values are evaluated to assess quality attributes. For quantitative attributes, the rating scale defined earlier will be used to gauge each quality attribute.*

Finally, ratings for each quality attribute are visually presented. The results of the evaluation can either be used as feedback for developers or an architect can iterate through step 2 to further refine the quality attributes.

4 Case Study

In this paper, we use a loan processing integration system to illustrate the use of MEMS for evaluating middleware architecture. The architecture design of this application is discussed in detail in [9] (see chapter 9). Different versions of the application have been implemented on several ESB platforms including Mule [15]. We have chosen this application because it satisfies the scenarios, which illustrate the basic requirements for the underlying middleware such as EJB, Messaging, Web Services and Enterprise Service Bus (ESB), to support dependable SOA design and integration. Therefore, evaluating the ESB middleware to support the dependable loan application can give insights into research issues involved in dependability analysis, and help us identify solutions in middleware-based architecture design.

4.1 Determine Quality Attributes

The goal is to enhance the existing loan processing application with adaptive failover capabilities. Please refer to Figure.1 for the means and metrics of interests.

4.2 Generate Key Scenarios

The loan processing workflow is as follows shown in Figure2. A customer requests a loan handled by a Loan Broker. The Loan Broker is responsible for firstly checking the credit rating of the customer and secondly contacting all the banks, querying the interest rate of each bank and returning the lowest interest rate to the customer. Individual entities, customers, loan broker, credit bureau and banks are all deployed in separate physical domains and integrated by messages transferred over Mule ESB. Technical details of configuring message channels on Mule are addressed in [15]. This system has dependability requirements on two key scenarios related to credit bureau operations:

Verify the Credit Bureau Operation. The credit bureau is an external service provided by a third party. The correct operation of this service should be ensured by periodically sending test messages. The test messages verify both the correctness of the data returned (such as the data should be within a range) and the response time of the external credit bureau operations.

Credit Bureau Failover. If the credit bureau malfunctions as the result of the verification, credit request messages should be temporarily routed to another service provider by a controller. While the message traffic is redirected, the monitor still keeps sending test messages to the primary provider. When the monitor confirms the correct operation of the service, the request messages are rerouted to the primary service provider.

These two key scenarios lead to further implicit architecture requirements:

- The test messages should not disturb the existing message flow. It is also required that the usage of network bandwidth sending test messages should be minimized.
- The correct credit bureau operations should be specified in policies, which are consulted at runtime for management and control decisions.

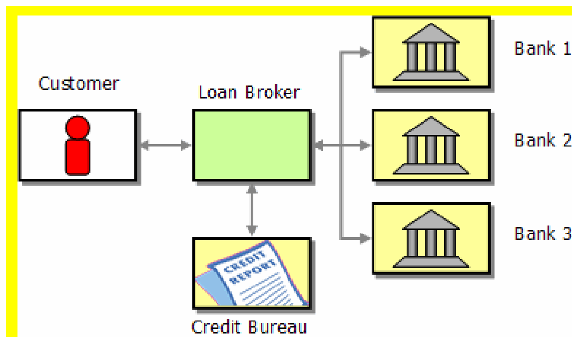


Fig. 2. Loan Processing System on ESB

The entities in the loan processing system should not be aware of the existence of those extra controlling entities that perform self-management. This helps to achieve separation of concern for promoting extensibility and maintainability. Changes to the quality of service management and control can be realized without modifying the business logic implementation and deployment.

4.3 Determining Pattern Alternatives

These key scenarios and their requirements motivate a self-managing architecture that enables periodic monitoring and adaptation according to the state of external services. The context and problem descriptions of M-A-P-E and policy point patterns fit in the solution space of this self-managing architecture. We therefore follow the evaluation steps described in section 3 to refine the solution based on these two patterns.

Quality attributes affected. Fault prevention and fault tolerance are the two means adopted to achieve reliability and maintainability. For fault prevention, periodic monitoring can help diagnose and detect service failures on-the-fly. The malfunction of a service can trigger a proactive action to recover a system from failures and faults. For fault tolerance, service redundancy is adopted to failover the malfunctioned services to backup ones. However, the failover service requires adaptive capabilities to be able to flexibly switch back to the primary service when the service is back to normal status. This may be because, for example, the primary service provider may provide substantial discounts under certain usage quotas.

In realizing these mechanisms, two more quality attributes are affected, programmability and risk. Programmability is concerned with the availability of the required mechanisms in the middleware, which determines the feasibility of a pattern implementation. Risk covers computing overhead, the availability of the control management, and the consequences of control management failure.

Metrics for measurement. We consider the qualitative metrics listed in Figure 1. Quantitative metrics are not within the scope of this paper because the evaluation is performed in a lab environment to illustrate the method itself. This is because the measurements of quantitative metrics, such as job failure ratio, throughput or response times, will have limited value when translated into a real system in a production environment.

Pattern implementation alternatives. The M-A-P-E and policy point patterns are realized using mechanisms and techniques from the Mule ESB. Figure 3 shows the resulting architecture that enables the self-managing capability. The grey colored components are from the original loan processing systems. The rest are extra components introduced to realize the self-managing architecture. The arrows indicate data flows.

In this architecture, the *Test Data Generator* generates test request messages and sends them to the *primaryCreditRequest* queue. A test message includes the reply address so that a *testReply* queue is dedicated for directing the replying messages to the *Test Data Verifier* component. The verifier not only checks the accuracy of the data but also measures the response time for processing a request message by an external credit bureau. It notifies the controller by sending a request message for controlling the action through a dedicated *controlRequest* queue.

One requirement for the test message generation is that the injection rate should not introduce significant workload to the primary service. A presetting of the injection

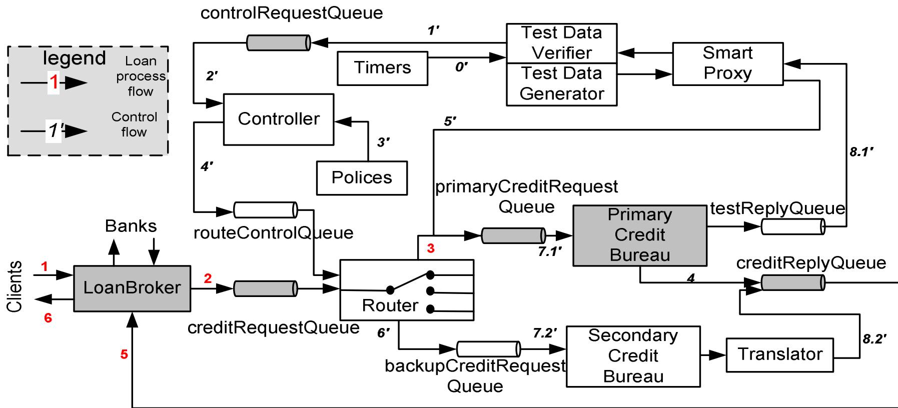


Fig. 3. Self-managing architecture of failover control

rate may serve well under light workload. However, when the primary service is stressed the test data generator should adjust to a slower rate or even stop generating any further test messages until the workload is reduced. This intelligence helps achieve the reliability of the self-managing service. The smart proxy design pattern [9] can be applied which employs a smart proxy component between the test data generator and the primary credit bureau.

The important control conditions are defined in policies. The controller component is responsible for interpreting the policy specification and making decisions on actions according to the run-time information collected by the test data verifier. The decision message is sent to a context-based router through the *routeControl* queue. The router has a connection with a secondary credit bureau through the *backupCreditRequest* queue. The reply message from the secondary credit bureau passes through the *creditReply* queue the same as the primary service. A translator in front of the *creditReply* queue translates the data format from the secondary service into the primary one, if there is difference in data format between the primary and secondary services.

Apart from the M-A-P-E and policy point patterns, this architecture also uses other integration patterns, including smart proxy, return address, context-based router and message translator [9]. These integration patterns are embedded in M-A-P-E and policy point patterns. They help to customize these two patterns to realize the key scenarios in the context of fault tolerance and fault prevention. Table 1 provides a summary of mapping between patterns and the architecture components.

With regards to the pattern variants, the M-A-P-E pattern forms a loosely coupled self-managing architecture. Components involved in the pattern do not directly invoke each other, rather the Mule ESB routes messages from one component/service to another. The policy management in this case study is simple. There are two policies, one for data correctness and one for response time. These policies are dependant on each other.

4.4 Identify Middleware Mechanisms

The middleware mechanisms that can support the flexible plug-in of new components are required to implement the design patterns shown in Table 1. Of most interest to this evaluation are the infrastructure components provided by Mule to assist self-managing architecture designers to monitor, intercept messages and adapt the system behavior based on the current environment. Mule provides a set of default routers which control and manipulate events received and dispatched by a component. In the router configuration file, Mule supports a property *<reply-to address/>*, which can be employed to realize the return address design pattern.

Table 1. Design Patterns and Architecture Components

Pattern		Components
M-A-P-E	Monitor	Test data generator, timer, testReplyQueue
	Analyse	Test data verifier, controlRequestQueue
	Plan	Controller, policy
	Execute	routeControlQueue, router, backupCreditRequestQueue, translator
Policy Point	PIP	Test data verifier
	PDP	Controller
	PEP	routeControlQueue, rounter
Smart Proxy		Smart proxy
Return Address		testReplyQueue, Test data verifier
Context-based Router		Router
Message Translator		Translator

In Mule, a content based router *FilteringOutboundRouter* can use filters to determine whether the content of the event matches a particular criteria, and if so it will route the event to one or more endpoints of the configured destination services on the router. Any outbound routers that extend *FilteringOutboundRouter* can apply a filter to it.

Mule supports the implementation of user specified router interfaces in a system. The deployment of a router can simply be specified in a Mule configuration file. This allows new components to be connected through messages directed by a router in between.

With regard to a monitor's runtime status, Mule provides interceptors, which allow a developer to intercept message processing on a Mule Unified Message Object (UMO) component and potentially alter the processing and outcome. They also allow various crosscutting concerns such as logging to be unobtrusively woven into components that were not originally designed to have such features. A UMO interceptor completely overrides control of the UMO component and takes control of the event. The UMO interceptor has the freedom of manipulating the event before and after the execution of the UMO component.

With these mechanisms, patterns in Table 1 can be implemented through configuration or extension at the API level. The smart proxy is the most complicated component. It implements a workload control algorithm, which uses a window size to control the

pace that test messages are sent to the primary credit bureau. Details of the smart proxy design and implementation are described in [7]. However its integration with the rest of the self-managing architecture also employs the above mentioned mechanisms.

4.5 Define Quality Attribute Scale

The attribute scale definition is listed in Table 2. We can perform qualitative impact analysis of maintainability by estimating how easily changes can be made to the architecture and its components. The risk attribute evaluates the reliability and availability of self-managing service itself. It has three sub-categories, namely overhead, availability and failure severity. For example, the attribute *failure severity* is dedicated to rank the severity of impact if the self-managing service fails. In this paper, we adopt the severity classification used in [17].

Table 2. Scale Definition of Metrics

Quality Attribute		Scale Definition		
		High	Middle	Low
Maintainability		Only changes in configuration or implementation for setting properties are required	Changes for configuration and out-of-box implementation are required	Customized implementations are required using middleware templates, patterns or frameworks
Programmability		Middleware has full support	Supported but with limitation	Not supported
Risk	Overhead	Overhead ratio is over 15%	Overhead ratio is less than 15%	Overhead ratio is lower than 5%
	Availability	No single point failure	only one single point of failure	The number of single points of failures > 1
	Failure Severity	(aka critical) A failure may cause major system damage or loss of production.	(aka margin) A failure may cause minor system damage, or delay or minor loss of production.	(aka minor): A failure may not cause system damage, but will result in unscheduled maintenance or repair.

We use a coarse grained scale, defined as high (H), middle (M) and low (L). We did not adopt an ordinal scale (e.g., 1 to 5) because the values do not truly represent the differences between scales in ratio or distance. In fact, the differences in their values only give indications of their relative rankings. If needed, the scaling definition can be refined later to be more fine-grained or use ordinal scale. The rating scale definition is applicable to each of the sub categories of risk and programmability. For programmability, we consider its evaluation for individual design patterns and aggregate the results. In order to aggregate the overall rating from those ratings given to individual sub categories, each sub-category has to be weighted. The value of the

weight represents how much the sub category contributes to the overall risk assessment. The scale definition as High, Middle and Low must be assigned some value and then the overall rating can be calculated. Finally the rating in the form of number is converted back into the descriptive form as High, Middle or Low. The value assigned can be arbitrary as it is just used as a means for calculation. A simple algorithm is used for calculation as shown in Table 3.

Table 3. Simple algorithm for calculating rating

1. Assign values to High, Middle and Low to convert the rating of each sub-category into values correspondingly.

$$H \leftarrow 1, M \leftarrow 2/3, L \leftarrow 1/3$$

2. Calculate the overall rating based on weight and the converted rating of each sub-category

$$Rate = \sum_{i=1}^{n} Weight_n \times Rate_n \qquad (\sum_{i=1}^{n} Weight_n = 1)$$

where n is the number of sub-categories

3. Convert back the value of the overall rating into the descriptive form.

$$Rate \leftarrow \begin{cases} H & (2/3 < Rate \leq 1) \\ M & (1/3 < Rate \leq 2/3) \\ L & (0 < Rate \leq 1/3) \end{cases}$$

4.6 Evaluation

The overhead of risk analysis is evaluated by measuring the computing overhead introduced by extra components in the self-managing architecture. However it is technically complicated to measure an accurate roundtrip time for a test message going through the whole execution path, because the messages are sent asynchronously between queues. Although time stamps can be attached to messages, the extra overhead of correlating messages will eventually lead to inaccurate measurement. Instead we evaluate the overhead using an approximation. It is insignificant that the time spent on sending a control message to the router to direct business messages to the secondary credit bureau. Therefore, most computing of self-management is in the test data generator, test data verifier and the smart proxy.

Our overhead measurement collects data of the overall throughputs and the CPU usage when these self-managing components are enabled and disabled respectively. The injection rate of the test messages is approximately 5% to 10% of the workload. As can be seen in figure 4, when the self-managing components are running, the CPU utilization (around 40%) and throughput (around 6 requests/second) averages are consistent with those measurements when self-managing components are disabled. We can therefore conclude that there is no significant performance impact in running our adaptive solution under the condition that injection ratio is less than 10% of the normal workload.

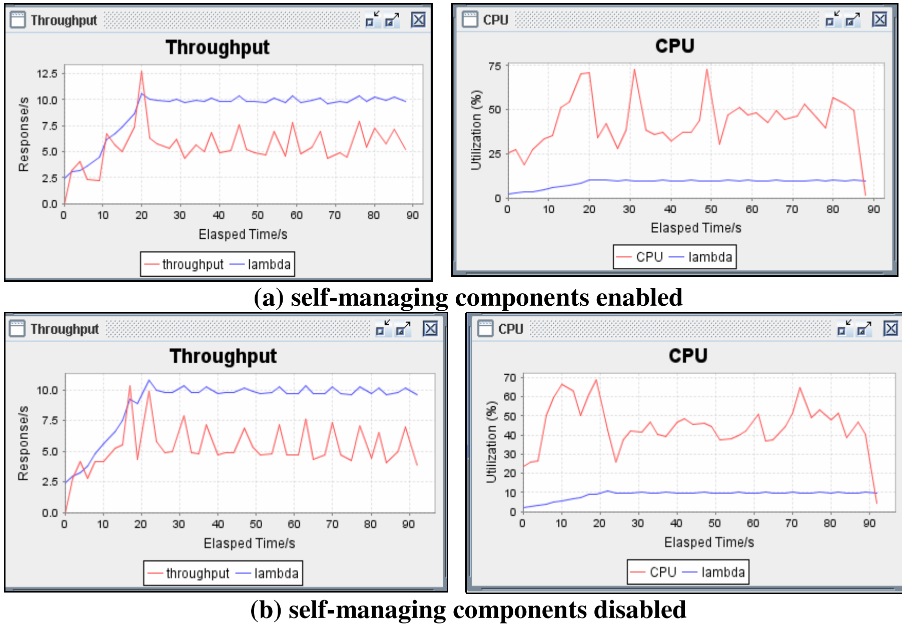


Fig. 4. CPU overhead of computing adaptation logic

For the rest of the qualitative evaluation, of quality attributes, the development team gave ratings for each quality attribute based on the detailed architecture descriptions, the rating scale definition and the middleware-based implementation in section 4. In this case study, the developer who participated in the architecture design and prototype implementation provided ratings. The correlation of scores between/among two or more evaluators who rate on the same scale can be estimated to measure the homogeneity of their ratings using an inter-rate reliability analysis method [8].

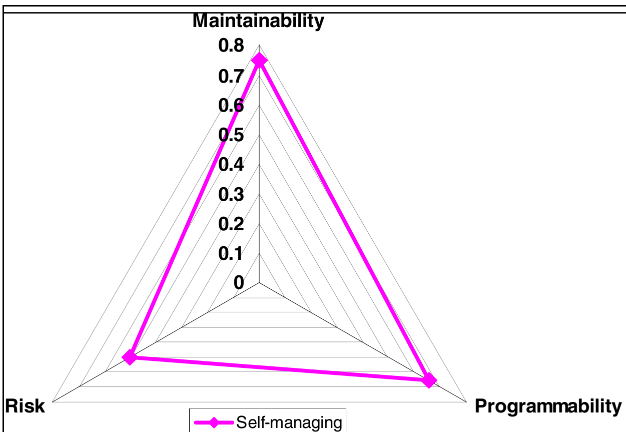


Fig. 5. Overall evaluation results

The overall results are presented in a radar diagram in Fig. 5 with each axis representing an individual quality attribute. Fig. 5 demonstrates how a self-managing architecture based on Mule performs with respect to attributes of dependability. The architecture has a high maintainability because Mule has provided efficient mechanisms to easily plug-in and out components in order to form different service structure. The programmability is at the high end of the middle ranking range. This means design patterns can be implemented using Mule with reasonable programming efforts to extend its default routers and interceptors. For the reported case study, it took less than 100 hours per person to implement the most complicated self-managing component, the smart proxy. The result shows this architecture has middle range risks because the availability of this self-managing architecture depends on the reliability of message queues (hosted by other messaging middleware). The malfunction of the self-managing architecture is less likely to result in critical damages to the whole system, however it can affect the service level agreement of the loan request processing.

5 Related Work

Many research efforts have been dedicated to ensure conformance between architecture quality attributes and implementation. Yacoub and Ammar [17] proposed a quantitative method for reliability risk assessment at the architecture level. The method used dynamic complexity and dynamic coupling metrics of implementation to define risk factors for the architecture elements. This method is based on component-based systems in which implementation entities explicitly invoke each other. The analysis method for this tightly coupled architecture is not suitable for loosely coupled service oriented architectures without extension. It remains our future work to integrate this method to do quantitative evaluation.

Our approach does not ignore or conflict with existing scenario-based architecture evaluation methods [5] or other approaches to analyzing dependability and its risk factors [4]. Rather, it extends and complements them by decomposing design pattern contexts and solutions into the architecture evaluation steps.

6 Conclusion

The design patterns described in this paper support the design of self-managing architecture that have fault tolerance and fault prevention as first class requirements. Implementation of these design patterns is heavily dependent on middleware mechanisms and techniques at the platform level. In this paper we propose an architecture evaluation method that focuses on middleware and design pattern integration. The contribution of this paper is a middleware evaluation method dedicated for pattern centric self-managing architectures. It has three main characteristics:

- It evaluates middleware architecture in the context of patterns and their variants, which allows fine-grained evaluation of middleware architecture;
- It explicitly captures key self-managing scenarios as functional scenarios, which encompasses the middleware's ability to preserve safe changes.
- Conducting both qualitative and quantitative evaluations.

The directions for future work include integrating further quantitative risk factor analysis for self-managing architectures and tools to document the evaluation results as reusable knowledge.

Acknowledgement

National ICT Australia is funded through the Australia Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

[1] A Primer on Policy-based Network Management, Hewlett-Packard Company (1999), http://www.openview.hp.com/Uploads/primer_on_policy-based_network_mgmt.pdf

[2] Ali Babar, M., Gorton, I.: Comparison of Scenario-Based Software Architecture Evaluation Methods. In: 11th Asia-Pacific Software Engineering Conference, pp. 600–607 (2004)

[3] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on dependable and secure computing 1(1), 11–33 (2004)

[4] Bass, L., Nord, R., Wood, W., Zubrow, D.: Risk Themes Discovered through Architecture Evaluations. In: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture WICSA, vol. 1. IEEE Computer Society, Los Alamitos (2007)

[5] Bosch, J., Bengtsson, P.: Assessing Optimal Software Architecture Maintainability. In: Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR), vol. 168. IEEE Computer Society, Washington (2001)

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester (1996)

[7] Chiu, R.: Enhance the Adaptivity of Integrated Services in SOA, Undergraduate Thesis, University of New South Wales, Australia (October 2007)

[8] Cronbach, L.J.: Coefficient alpha and the internal structure of tests. Psychometrika, 297–333 (1951)

[9] Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Reading (2003)

[10] IBM online article, An architectural blueprint for autonomic computing, (October 2004), http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html

[11] Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: 2007 Future of Software Engineering. ICSE, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)

[12] Liu, Y., Gorton, I., Bass, L., Hoang, C., Abanmi, S.: MEMS: A Method for Evaluating Middleware Architectures. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214. pp. 9–26. Springer, Heidelberg (2006)

[13] Liu, Y., Gorton, I.: Implementing Adaptive Performance Management in Server Applications. In: International Workshop on Software Engineering For Adaptive and Self-Managing Systems (SEAMS 2007), ICSE, pp. 12–21. IEEE Computer Society, Los Alamitos (2007)

[14] Martin, P., Powley, W., Wilson, K., Tian, W., Xu, T., Zebedee, J.: The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. In: International Workshop on Software Engineering For Adaptive and Self-Managing Systems (SEAMS 2007), ICSE, pp. 9–18. IEEE Computer Society, Los Alamitos (2007)
[15] Mule Enterprise Service Bus, http://mule.mulesource.org/wiki/display/MULE/LoanBroker
[16] Srivastava, B., Bigus, J.P., Schlosnagle, D.A.: Bringing Planning to Autonomic Applications with ABLE. In: Proceedings of the First international Conference on Autonomic Computing (ICAC 2004), pp. 154–161. IEEE Computer Society, Los Alamitos (2004)
[17] Yacoub, S.M., Ammar, H.: A Methodology for Architecture-Level Reliability Risk Analysis. IEEE Trans. Softw. Eng. 28(6), 529–547 (2002)
[18] White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Segal, A., Kephart, J.O.: Autonomic computing: Architectural approach and prototype. Integr. Comput.-Aided Eng. 13(2), 173–188 (2006)

Comprehensive Architecture Evaluation and Management in Large Software-Systems

Frank Salger[1], Marcel Bennicke[2], Gregor Engels[1,3], and Claus Lewerentz[2]

[1] sd&m AG, Carl-Wery-Straße 42, 81739 München, Germany
frank.salger@sdm.de
[2] Brandenburg University of Technology, Postbox 101344, 03013 Cottbus, Germany
{mab,cl}@informatik.tu-cottbus.de
[3] University of Paderborn, s-lab, Warburger Str. 100, 33098 Paderborn, Germany
engels@upb.de

Abstract. The architecture of a software system is both a success and a failure factor. Taking the wrong architectural decisions may break a project, since such errors are often systematic and affect cross-cutting aspects of the system to be built. Moreover, software projects get more and more challenging due to the rising complexity and dynamics of business processes, large team size and distributed development. As the software architecture is the common platform for many project activities, it constitutes a critical success factor. Thus, a comprehensive method for evaluating a software architecture and propagating important properties of it downstream to code is needed. At sd&m, we designed a comprehensive architecture evaluation and management framework in order to satisfy these needs. In this paper, we derive a list of requirements, such a framework should fulfill. We then present the components of our architecture evaluation method and demonstrate, how it fulfills these requirements.

1 Introduction

Conceptual errors in software development can get very expensive quickly: They can slow down the process due to badly defined responsibilities of components. They can make the system difficult to be changed, extended, maintained, and tested. Or they can give rise to inaccurate estimations, which themselves lead to wrong project plans and failed deadlines. In the context of ever growing complexity of business processes, project sizes, time pressure as well as massive distributed development, the soundness and appropriateness of both, development artefacts and processes are more important than ever for project success. Hence, comprehensive methods to evaluate and continuously manage the maturity of all aspects of the development process are needed.

One of the business areas of sd&m is the design and the development of information systems for business-critical processes of our customers. The soundness, appropriateness and maintainability of our software architectures is one major success factor of such projects. Many methods like ATAM, ARID or SAAM [1] have been devised in order to assess software architectures. By using dedicated approaches and viewpoints, some of the methods particularly enable analyzing specific aspects or artifacts in detail, neglecting others. Also these methods need to be tailored towards a particular context

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 205–219, 2008.
© Springer-Verlag Berlin Heidelberg 2008

in order to be efficient and effective. For example, the quality of results of an ATAM evaluation largely depends on discovering relevant architectural decisions. Eventually such evaluations are intended to be executed at a given point in time, but do not continuously monitor the development process. Hence, a single of these methods is not sufficient in order to fully address all architectural concerns of a software project and enabling the evaluation to be performed efficiently.

The high-level objectives of a comprehensive architecture evaluation and management framework are:

- To evaluate, whether the needs of the customer are supported by the software architecture.
- To determine, whether the architecture constitutes an appropriate solution for the problem.
- To secure, that the architecture serves as a robust fundament for the following development activities.
- To enforce the defined fundament for the following development activities.

In this paper, we present the two architecturally relevant components of our architecture evaluation and management framework - the *architecture quality gate* and the *architecture management process*. We show how the architecture quality gate and the architecture management can be embedded into the software development process and how they interact.

In the next section, we elaborate on the requirements that have to be fulfilled by a comprehensive architecture evaluation method. In section 3, we describe our concept for quality gates at sd&m in general and the architecture quality gate for business information systems in particular. Section 4 explains the architecture management process and its interrelations with the architecture quality gate. We then report on successful applications of our framework in large projects at sd&m. Finally, we draw our conclusions and indicate future work.

2 Requirements for Software Architecture Evaluation Methods

In this section, we discuss and motivate the requirements that should be fulfilled by a comprehensive evaluation method for software architectures.

1. The method checks whether the architecture addresses all architecturally relevant requirements in a balanced manner

The method should support the assessor to spot incomplete, inconsistent or ambivalent architecturally relevant requirements. As architecturally relevant requirements are often interrelated with each other, the method should check whether the architecture addresses the requirements in a balanced manner on which all stakeholders can agree.

2. The method should question the reasons behind the architecturally relevant decisions

Software architecture can be seen as the earliest available set of design decisions in the engineering process. As such, the chance should be taken to validate the decisions

against the requirements as early and rigorously as possible. The method should provide a comprehensive set of key questions, in order to systematically address all critical areas of a software architecture.

3. The method should mitigate the problem of "getting routine blinded"
Everybody finds an error easier than the one who made it. Thus, it is essential, that the method incorporates external expertise and draws on consolidated knowledge.

4. The method should include the evaluation of prototypes
In most of the process models for software development it is recommended to build evolutionary prototypes, e.g.[2]. Usually, many coding templates are derived from the evolutionary prototypes. Since these templates will be used by dozens of developers, their quality can be critical to the success of the whole project.

Hence, in case of developed prototypes, the method must verify whether they are really good enough to serve as team-wide templates. For explorative prototypes it must be verified whether the predictions drawn from them are valid (e.g. the estimated throughput concluded from prototype performance measurements).

5. The method must support the evaluation of both, processes and products
Obviously, the quality of a product is influenced by the process that creates it and an architectural design must be aligned with the process that implements it (for example with respect to assigning components to developers for implementation). Thus, both, the product and its development process must be evaluated.

6. The method must assess artefacts generated in earlier activities which serve as input to the design of the architecture and the following activities
For example it is reasonable to assess early in the requirements specification activity, whether an appropriate specification method is used systematically. However, we cannot be sure, whether this specification method is actually used in the iterations that follow this assessment. Hence, we have to evaluate the requirements specification before we start construction. Of course, the focus will be different this time: We should mainly assess, whether the "good practices" firmed by the specification assessment where continued.

7. The method should foster consistency of artefacts downstream in the development process
Due to time pressure, mid-course corrections or lacking overview, artefacts tend to drift away from each other with negative effects on software product quality. The method should include further processes that help to perpetuate the quality of the software architecture.

We present an integrated framework which considers all of these requirements. Requirements 1 through 6 are addressed by the specific design of the architecture quality gate (section 3). Requirement 7 is addressed by the architecture management process (section 4).

3 Quality Gates at sd&m

sd&m AG – software design & management – develops high-performance custom software solutions and provides consultancy services for all aspects of IT. Our customer base boasts major enterprises from all sectors and public institutions, particularly from the automotive, banking and insurance industries.

In order to securely finalize large scale development projects to the satisfaction of our customers, we developed a quality framework consisting of selective and continuous evaluations.

3.1 Basic Concepts

At sd&m, software projects follow the process as indicated in figure 1. Quality gates are comprehensive evaluations executed at specific points in time assessing the maturity and sustainability of produced artefacts (milestones) and the processes followed to produce them. We have developed quality gates for securing the quality of the tender, the requirements specification, the architecture and the integration test. With our quality gates, we pursue the following high-level objectives:

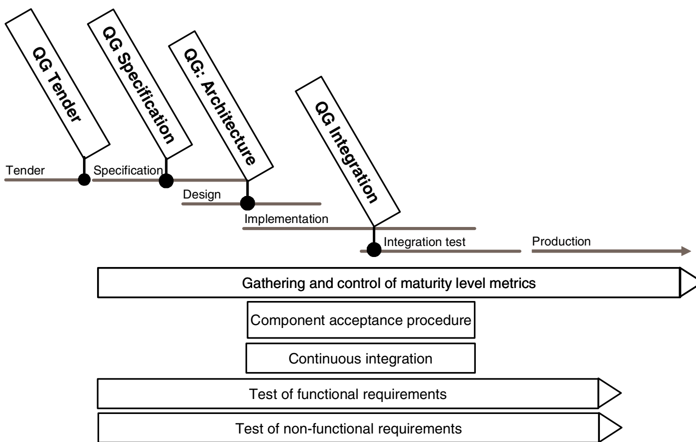


Fig. 1. Quality gates in context

- To make the maturity of development artefacts and processes transparent.
- To derive effective countermeasures for major problems encountered.
- To “standardize” audits to a reasonable extent.

The main characteristics of applying quality gates are:

- They are executed according to a defined quality gate process and end with a decision („Am I ready for the next step?“).
- They are no formal checks but evaluate the content of artefacts.
- They are conducted by sd&m-experts (senior developers not involved in the project).

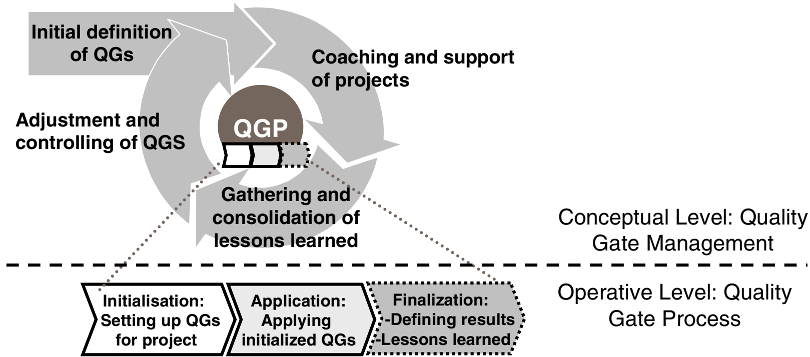


Fig. 2. Quality gate processes

The quality gates together with the ongoing processes indicated in figure 1 constitute our integrated framework used to develop high quality software. The continuous gathering and control of maturity level metrics complements the milestone evaluations by increasing the transparency of the quality of the produced results. In particular the architecture management process which addresses continuous architectural concerns is part of the metrics collection process.

When we defined our quality gate concept, we considered international standards and further literature that address the topic of defining audits, inspections or reviews [3, 4, 5 6, 7]. However, we do not use ongoing formal statistical evaluation of the quality gate results as suggested for inspections by [3] and [6].

As shown in figure 2, the quality gate concept at sd&m consists of two process levels. The first level addresses the lifecycle aspects of quality gates, like the definition and improvement of the quality gates and the associated processes. The second level addresses the operative aspects of the quality gates, like the initialization, the application and the finalization of quality gate within one specific project.

3.2 The Architecture Quality Gate

Having presented the key ingredients of our quality gate concept, this section discusses the architecture quality gate in more detail. For our architecture quality gate we considered the following evaluation methods and standards:

1. The architecture tradeoff analysis method (ATAM) of the SEI, Carnegie Mellon University [1]. ATAM is a very powerful scenario-based method for evaluating software architectures .We adapted it for our purposes.
2. The CMM(I), Version 1.2 of the SEI, Carnegie Mellon University [8]. We used this process assessment model to derive questions to assess the architecturally relevant processes.
3. ISO/IEC 15504 Part 2 and Part 5 [4, 9]. We used this standard to check the completeness of our “process questionnaire” (which will be presented below).

4. ISO/IEC 9126 Part 1 [10]. The structure of our "architecture questionnaire" (which will be presented below) has been inspired by this standard.

We now present the different evaluation steps that make up our architecture quality gate. Not all steps need to be applied necessarily. By default, all steps are mandatory, as long as it can be reasonably argued, that it can be omitted. Criteria exist upon which the responsible assessor may decide to omit specific steps.

To some extend, the steps build upon each other in the sense, that the steps successively dig deeper into the actual quality of the architecture. However, their execution order can be changed, if the projects context suggests it: In a project which consists of multiple sub-projects, it could be sensible to check the effectiveness of cross-project processes (like a centralized change management process) in the first step. If such processes are not in place, the overall project has a serious problem, regardless of the concrete software architecture.

Evaluation step 1: Requirements specification questionnaire
Motivation: Even a perfect software architecture will not help developers code the use cases or functional requirements, if these are incomplete, contradictory or in a very bad formal shape. Hence, after a software requirements specification (SRS) has initially been checked in the specification quality gate, we check its correct continuation in the architecture quality gate by means of this questionnaire.
Objectives: To check whether (1) functional requirements be understood by the developers, and (2) the SRS was continued in the way, it was approved by our specification quality gate.
Benefits: Approval that (1) the functional requirements are specified in a way understandable by the developers, (2) the SRS does not contain contradictions and can be realized, and (3) the results of the specification quality gate have been considered.

Evaluation step 2: Architecture questionnaire
Motivation: We check a software architecture at a conceptual level as well as whether the architecture complies with our sd&m standard QUASAR (quality software architecture, see [11]) for business information systems. However, this questionnaire is no dogma. We accept answers which do not match the preferred answer but are reasonably motivated.
Objectives: (1) To get a high-level overview of the architecture, (2) To pose detailed questions about the architecture along three dimensions: 1) The "object" to be assessed like "whole system", "client tier", "business tier", "database architecture", etc. 2) Development principles like "separation of concerns", "low coupling", "testability", etc. 3) The quality attributes of ISO/IEC 9126, like "usability", "reliability", etc. This supports different views onto the architecture.
Benefits: (1) Impression of the quality of the specific parts of the system as well as the system as a whole, (2) Confidence whether the architecture complies to QUASAR, the sd&m standard for architectures of business information systems, and (3) Hypothesis, were the real problems are, as input for the following methods like the lightweight-ATAM and the hotspot-analysis.

Evaluation step 3: Lightweight-ATAM
Motivation: The attribute utility tree[1] plays a central role within ATAM. This tree is generated from scratch in the "investigation and analysis" step of the ATAM session.In order to speed up this time consuming step, we demand the full, project specific attribute utility tree (which has also been checked and prioritized by the customer) from the architect as input for the quality gate. Further, we demand a mapping (called the "A-matrix") of the key architectural decisions onto this attribute utility tree. Based on these two inputs, the ATAM session can be conducted very efficiently. Further, generating the utility-tree and the A-matrix forces the architect to reason explicitly about his key architectural decisions. Basically, the results of the lightweight-ATAM correspond to the results of the standard ATAM. However, the A-matrix formalizes the ATAM-concepts of sensitivity- and tradeoff points as will be explained in the case study section. To help the projects setting up their specific attribute utility tree quickly, we provide a "standard" attribute utility tree geared to the development of business information systems. We tailored ISO/IEC 9126 (Part 1) to the type of systems we evaluate with our architecture quality gate.
Objectives: (1) By means of the creating the A-Matrix, the project creates precise cross-cutting requirements and a mapping of the cross-cutting requirements to the architectural decisions. Another goal is (2) the verification of the architecture against the quality model of the customer.
Benefits: (1) Confidence, which architectural decisions bear high risks (like tradeoff-points, sensitivity points and risks), (2) Identification of inconsistencies between requirements, and (3) Confidence, that the system is build, the customer actually pays for.

Evaluation step 4: Prototype questionnaire
Motivation: Prototypes are used to explore requirements, estimate performance impacts or serve as templates for dozens of developers. They should therefore be assessed.
Objectives: (1) Check, whether an evolutionary prototypes is sufficiently mature to serve as a template (e.g. compliance to coding style, robustness), (2) Check whether the behaviour of the system to be built can really be inferred from the exploratory prototypes.
Benefits: (1) Projects are motivated to build exploratory prototypes, (2) Confidence, whether the code of evolutionary prototypes can be safely used as a template, and (3) Statement about the validity of predictions made using the prototype or yet uncovered risks,

Evaluation step 5: Hot spot analysis
Motivation: Having applied the aforementioned methods, the assessor should have a very good grasp of the main problem areas allowing her to focus specific items afterwards. The quality gate assessor (which is a senior architect) should use her experience to sharpen the problems. This crisp problem description will then serve as the basis for the definition of precise countermeasures.

[1] The attribute utility tree basically is a quality model for the software system to be built, consisting of quantifiable, prioritized requirements and consolidated by the stakeholders of the architecture.

Objectives: (1) Generating a list of the main architectural problems, and (2) Detailed investigation of these problems.
Benefits: (1) Determination which risk these problems pose on the overall success of the project, and (2) Definition of countermeasures by senior architects.

Evaluation step 6: Process questionnaire
Motivation: The previous steps concentrate on the evaluation of artefacts. However, projects can also fail due to weak change management processes, or unrealistic project plans for the development activities It is therefore necessary to check the processes used for the development of the architecture as well. We used CMM(I) and ISO/IEC 15504 as an orientation to derive questions for this evaluation step.
Objectives: (1) Assessing the project management with respect to architectural concerns and (2) assessing the quality management with respect to architectural concerns.
Benefits: Assuring (1) the appropriateness of the processes in place, (2) the appropriateness of the processes for the following activities, and (3) a safe transition to the following development activities and phases.

The effectiveness of these steps has been evaluated with several large projects. The concerted definition of the above steps and their application provides valuable synergies. The architecture quality gate allows the assessor to question all decisions made so far within the project and evaluate them in the full context. It is designed such that it allows suspicious issues discovered in one step may be scrutinized by another step. Yet the evaluation is structured and well-integrated with the constructive development process to be efficient. Section 5 gives some examples from the evaluations of the architecture quality gate.

4 Architecture Management

4.1 Motivation

The overall goal of the entire evaluation framework sketched in section 3.1 is to eliminate risks jeopardizing delivery of a high-quality product within time and budget. Upon completion of the architecture quality gate, there is general approval that the so-far designed artifacts based on the available information will fulfill the end-user requirements as well that the preconditions for efficient execution of the remaining process phases are met. Particularly, the quality gate verifies whether common goals such as performance, maintainability, reusability and team buildability are supported by architectural decisions. However the execution of the following phases poses new risks. For example, there remain risks of not meeting performance and reliability goals when the system operates under real-world conditions or there remains a risk that badly developed source code may slow down the entire process (due to bad documentation, frequent changes or bug fixes). These and other risks are handled by the outlined continuous gathering and control of maturity level metrics. This process activity collects metrics from static and dynamic tests of the product as well as supporting systems that characterize results of the process (e.g. error tracking system, configuration management, time tracking system). In the following we outline one such measurement process, that deals with the static module view onto a system.

4.2 Dependency Management

A particular risk that endangers the conclusions drawn from architectural decisions is that the implementation may drift away from an initially developed design and the system documentation. The drift surfaces in structural mismatches between as-planned/as-documented and as-implemented modules and the uncontrolled creation of dependencies between modules during the implementation phase. The detrimental effect of dependencies on the quality of software and their interactions with organisational structures has long been identified by several researchers (see e.g. [12, 13, 14]). Keeping the code in sync with its design is important since many evaluations carried out at the architecture quality gate and the setup of the following phases build upon the designed structure and become invalid as the actual structure diverts from it. On the other hand, the drift certainly does not occur suddenly and deliberate, but in the light of time pressure and changing requirements rather gradually and unintended.

Dependency management is about controlling the consistency between an abstract high level model describing planned modules and dependencies and the implemented software system realized as packages, classes and (source code) files stored in a file system. In the past, several conceptual approaches and tools have been proposed to describe and perform this kind of consistency checks ([15, 16, 17, 18, 19, 20] to name a few). Which of these approaches fits best in a particular context, depends on factors such as intended major use case, capability to describe the system structures to be monitored including their evolution, scalability and ease of integration with the existing tool environment.

Performing consistency checks requires the following parameters as inputs:

- An as-designed high-level model of the software architecture. The model defines all subsystems and their allowed or required dependencies. The dependencies specification covers inter-subsystem relationships as well as relationships with external components that a project chooses to use.
- A mapping between each architectural subsystem and the code that implement this subsystem.

Using this information a tool extracts an as-implemented model from the source code and compares it to the as-designed model. The method unveils any differences between the as-designed and as-implemented models of a software system.

If consistency checks should be applied, the required as-designed model and its mapping to code should be fixed before entry into the implementation phase (cf. figure 1) with two goals

- It serves developers as a guideline on how the code must be structured.
- The code can be checked against the model right from the first line.

Thus, the process questionnaire of the architecture quality gate checks whether, this information has been prepared during the design.

We employ consistency checks in two modes:

- Within the development environment: From our perspective consistency checks must be executed within the development environment of every developer since this generates prompt feedback. It prevents differences to occur right in the first

place and to cumulate over time. However, this application mode has only local effects as each developer cleans up only those errors he is responsible for. We also do not expect developers clean up every error instantly.

- During continuous integration: Second, we execute consistency checks as part of our regular continuous system integration build and collect metric values about the results of the checks. These metrics are made available to the architect to help him asses the overall architectural quality and make decisions e.g. about where to schedule refactorings next. The metrics quantify how much code is covered by the analysis and indicate the degree of non-consistency. For example, the number of unplanned dependencies is collected for the entire system and for each subsystem.

The resolution of a detected inconsistency requires either a change to the code or even a change to the architecture and its documentation. Where code changes can easily be dealt with during implementation, a change of the architecture is handled within the larger scope of a change management process if necessary.

To summarize, the interleaving of the architecture quality gate and the dependency management process has the following benefits:

- The project is in a state to be able to perform consistency checks.
- The project closely follows the design such that assumptions made for the implementation phase continue to hold.
- Correctness of development documentation

5 Case Studies

In the following we present the application of the architecture quality gate and the architecture management to demonstrate the effectiveness of the evaluation methods. We selected methods of different types for demonstration: a questionnaire-type evaluation, the lightweight ATAM evaluation method and the continuous architecture management method.

5.1 Case Study 1: Effectiveness of the Architecture Questionnaire

The goal of this study was to confirm the architecture questionaire's ability to pinpoint substantial problems. From a real world project, we sketch two of the posed questions and the discovered problems in more detail:

Question: In which way does the customer define "quality" with respect to the architecture?

Rationale: If the architect does not know the consolidated quality model shared by all stakeholders the customer, it is not clear whether the "right" system will be built, and there is a high risk that the customer will not accept the built system. At least a standard like ISO9126 should be used to check the requirements for completeness.

Actual Answer: "We mainly talk with the IT-department in order to gather the requirements and determine the quality attributes."

Since it is a requirements specification anti-pattern not to consider all stakeholders of the application (in this case the user) this answer indicated a potential problem with the requirements. A deeper investigation in the course of the quality gate in fact showed, that both, the use case model and the non functional requirements where flawed and had to be reworked. We note, that it would not have been possible to use ATAM in this project since the end users were not available during this specific period. Nevertheless, we found the problem since the architecture quality gate uses different methods to evaluate architectures.

Question: How much functionality of the technical COTS are actually used?

Rationale: Functionality of COTS which is not used may develop into a product lifetime burden. The unused parts can be flawed. Due to compatibility reasons COTS need to be updated even if the used functionality is not affected from the flaw. A COTS update can be expensive as it may trigger a cascade of redeployments of components into the end-user environment. If not the whole functionality of the COTS is used, other COTS should be evaluated. Any unused functionality of a COTS should be hidden by using an adapter.

Actual Answer: “We only use the ‘reminder functionality’ of the workflow component.”

A deeper investigation showed, that this workflow component mandated an asynchronous interaction with the application layer of the system adding a lot of complexity. Later, the workflow component was removed and the “reminder functionality” was implemented as a technical service within the system itself. It would be very hard to cover this situation by use case-, growth- or exploratory scenarios. Hence, it seems to be unlikely that we would have discovered this problem using e.g. ATAM only.

5.2 Case Study 2: Effectiveness of Lightweight ATAM

The goal of this case study was to verify whether our lightweight ATAM would still enable us to uncover substantial risks such as inconsistencies of requirements, and inadequate architectural decisions.

Figure 3 shows a fragment form the full A-matrix. Requirements are arranged in rows, architectural decisions in columns. The architectural decisions are related to requirements by marking the appropriate cell with a “p” or “n”, meaning, that the decision has either a positive or a negative impact on the according requirement. A “0” means no impact, a “W” indicates a conflict. The A-matrix formalizes the concepts of sensitivity- and tradeoff points: Each “p” or “n” corresponds to a sensitivity point. A column which contains at least one “p” and one “n” is a tradeoff point, since the architecture decision in this column affects at least two adjacent requirements in an opposite way. This is the case the requirement U-1 “Resolution of 1280 x 900 pixel must be possible” conflicts with the architectural decision IN-3 “Usage of clients "standard-client", since the “standard-client” did not support this resolution. However, this architectural decision was explicitly stated as a requirement (ADC-3). Hence, by means of ATAM we revealed that the requirements ADC-3 and U-1 together lead to a contradiction. A further benefit for this project was the further sharpening of requirements done during the ATAM session.

	Architecturally relevant decisions	100MBit networks at developer workstations	Network to processing plant: 100 Mbit	Usage of clients "standard-client"	...	...	Frameworks	XML Parsing	Document workflow with COTS	...	Integration	Legacy Systems read directly from core database	COTS for document workflow encapsulated	...	...
	Requirements	IN-1	IN-2	IN-3				F-1	F-3			INT-1	INT-2		
...	...														
Usability (Usability Compliance)															
U-1	Resolution of 1280 x 900 pixel must be possible			**W**											
...															
...															
Non ISO-Characteristics															
Cross cutting architecture and design constraints															
ADC-1	Usage of a component architecture											**n**	**p**		
ADC-2	Encapsulation of data access											**n**	**p**		
ADC-3	Usage of clients "standard-client"			**0**											
Cross cutting project constraint															
PC-1	Very early employable, production stable first version of system											**p**	**n**		
...															

Fig. 3. Fragment of the A-matrix

5.3 Case Study 3: Architecture Management

Being able to perform consistency checks as outlined above requires tools. Hence part of this case study was to evaluate several consistency checking tools for their appropriateness in our context. A second goal was to quantify the drift between design and implementation for a real-world system to verify there is actual need for this technique. The analysis has been conducted as a post mortem analysis after completion of a system representative in terms of its size and architecture.

The as-designed model has been built according to the available design documents and with the help of the project's architect. The module structure is described by two perspectives both of which are a direct result of the Quasar software design method [11] used at sd&m. The so-called application architecture (A-architecture) defines distinct topics and their dependencies from a problem domain perspective. The technical architecture (T-architecture) defines a technically motivated subdivision of the system into distinct layers. Actual components arise from combining both perspectives. Usually a component stretches over one problem domain topic and several or all layers.

The resulting (simplified) overall architecture is shown in figure 4. The leading column and line of the matrix show the T- and the A-architecture subdivision and the allowed dependencies in each of these views (<<public>> means the module may be used by any other module). Each of the matrix elements is a subsystem to which actual code can be assigned. Both, the A-architecture and T-architecture dependency

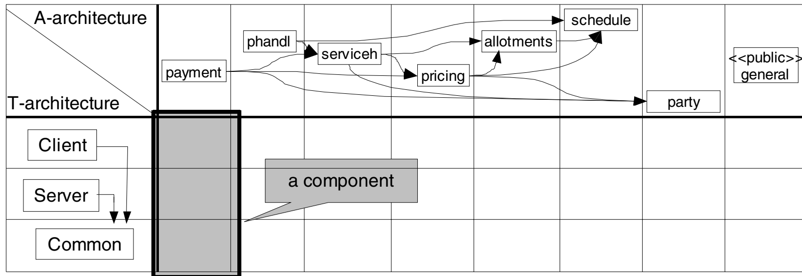


Fig. 4. Architecture of the passenger transportation system

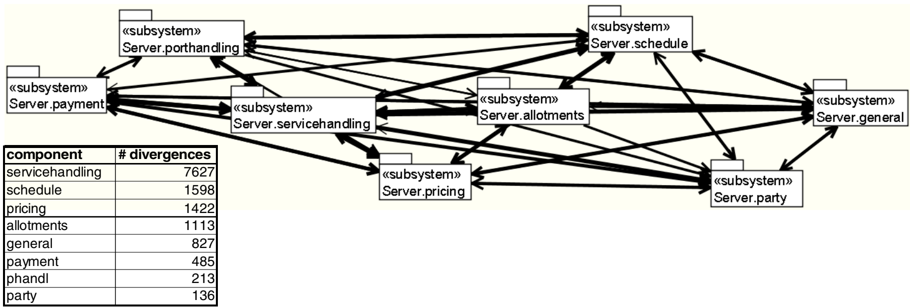
definitions apply to the code of each subsystem. In this example the T-architecture defines a client part, a server part and a common part which establishes communication between the client and server parts. Each of these parts has further sub-layers.

The entire system is about 1.45 Million lines of code, where the above model covers about 1.2 Million lines of code (82 %). There were no high-level modules without code.

The above matrix can almost directly be modelled using a consistency checking tool called SonarJ [18]. The mapping of code to each of the subsystems was sometimes difficult, as SonarJ can only assign entire packages whereas some packages contained code that would have to be assigned to different subsystems. Also, from the source code it became visible that some domain components actually contain further sub-components which could be used to refine the above model. However, they were ignored for this analysis.

The result of comparing the T-architecture was that the division of the client, server and common parts has been implemented as planned. No single divergence has been found in this perspective. This is not surprising as the build and deployment process for the software forbids anything else. However, when the detail of the T-architecture is increased and also sub-layers are modelled, unplanned dependencies within the server-part became visible. Results are much different for the A-architecture. In total, there are 13421 unplanned dependencies between the above mentioned modules (see figure 5). All domain components have unplanned dependencies. In particular, the component “general” which was intended to be independent from other subsystems depends upon all other subsystems. Second, many mutual dependencies have developed which can make components hard to understand or test independently.

The architecture model of this system is typical for the business information systems sd&m builds for its customers. Thus, there are good chances, other systems can be covered with this matrix-like style to describe architectures. The case clearly illustrates that without constant monitoring or other forces in place the dependency structure erodes already during the initial implementation of a system. It also illustrates the need to have the source code organized such that it is suitable for consistency checks – a point that will be included in the architecture quality gate to ensure applicability of this technique.



component	# divergences
servicehandling	7627
schedule	1598
pricing	1422
allotments	1113
general	827
payment	485
phandl	213
party	136

Fig. 5. As-implemented A-architecture of the passenger transport system and number of divergences caused by each subsystem.

6 Conclusion and Future Work

We presented the basic ideas of the overall quality assurance framework at sd&m in general and highlighted the architecture quality gate and the dependency management process in particular. By applying our architecture quality gate we ensure that the critical architectural decisions are resolved such that

- the business need of the customer is addressed by the architecture,
- the non functional requirements are supported,
- the architecture can be used as a firm and stable platform for the developers and testers,
- the current processes are precise, communicated and followed,
- and the processes for the following activities are in place and reasonable.

As such, we make sure the architecture supports the project to be finished in time and budget satisfying the customer. The architecture management process ensures the system properties fixed in the architecture continue to hold throughout the following process phases. Keeping the implementation in sync with its design facilitates efficiency of the process. As the design is usually part of a system's development documentation, the transition into future maintenance activities is also improved.

We reported on some applications of parts of the architecture quality gate and the architecture management to large scale systems and showed which benefits can be obtained.

The presented architecture quality gate is optimized for the evaluation of architectures of business information systems. Part of our future work will be the definition of an architecture quality gate for business intelligence applications as well as quality gates for the requirements analysis phase and the integration testing. We also aim at further refining the architecture management aspect. One goal is to define standard T-architectures for recurring types of components such as user dialogs, data management or reporting and develop a model based consistency checking approach that can cover such component types. Second, we intend to include analyses in the architecture management process that address further fundamental system properties defined by the architecture (e.g. error handling strategy).

References

1. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures Methods and Case Studies. Addison-Wesley, Reading (2002)
2. Rational Unified Process. IBM Coporation. Version 7.0.1. 2000 (2007)
3. Fagan, M.: Advances in Software Inspections. IEEE Transactions on Software Engineering 12(7), 744–751 (1986)
4. ISO/IEC 15504-2:2003(E). Software engineering — Process assessment — Part 2: Performing an assessment
5. IEEE Std. 1028-1997 – Standard for Software Reviews. The Institute of Electrical and Electronics Engineers, Inc. New York (1997)
6. Gilb, T., Graham, D.: Software Inspection. Addison-Wesley, Reading (1993)
7. Wiegers, K.E.: Peer Reviews in Software – A Practical Guide. Addison-Wesley, Reading (2002)
8. CMMI® for Development, Version 1.2. CMU/SEI-2006-TR-008, ESC-TR-2006-008. Carnegie Mellon, Software Engineering Institute (2006)
9. ISO/IEC 15504-5:2006(E). Information technology — Process assessment — Part 5: An exemplar process assessment model
10. ISO/IEC 9126-1:2001(E). Software Engineering — Product Quality — Part 1: Quality Model
11. Haft, M., Humm, B., Siedersleben, J.: The Architect's Dilemma – Will Reference Architectures Help? In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712. pp. 106–122. Springer, Heidelberg (2005)
12. Conway, M.: How Do Committees Invent? Datamation 14(4), 28–31 (1968)
13. Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems into Modules. CACM 15(12), 1053–1058 (1972)
14. Lakos, J.: Large-Scale C++ Software Design. Addison-Wesley, Reading (1996)
15. Murphy, G.C., Notkin, D., Sullivan, K.: Software Reflexion Models. Bridging the Gap Between Source and High-Level Models. In: Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 18–28. ACM Press, New York (1995)
16. Koschke, R., Simon, D.: Hierarchical Reflexion Models. In: Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), pp. 36–45. IEEE Computer Society, Los Alamitos (2003)
17. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: Proceedings of the 20th annual ACM SIGPLAN conference on object oriented programming, systems, languages, and applications, OOPSLA 2005, pp. 167–176. ACM Press, New York (2005)
18. SonarJ vendor homepage, `http://www.hello2morrow.com`
19. SotoArc vendor homepage, `http://www.software-tomography.com`
20. Becker-Pechau, P., Bennicke, M.: Concepts of Modeling Architectural Module Views for Consistency Checks Based on Architectural Styles. In: Smith, J. (ed.) Proc. of the 11th IASTED International Conference on Software Engineering and Applications (SEA 2007), Acta Press (2007)

Sharing the Architectural Knowledge of Quantitative Analysis

Anton Jansen[1], Tjaard de Vries[1], Paris Avgeriou[1], and Martijn van Veelen[2]

[1] University of Groningen, Department of Mathematics and Computing Science,
P.O. Box 800, 9700AV Groningen, The Netherlands
anton@cs.rug.nl, tjaard@tjaard.nl, paris@cs.rug.nl

[2] ASML, DE-SSD, Litho Systems Architecture
5504DR Veldhoven, The Netherlands
martijn.van.veelen@asml.com
Previously employed by ASTRON
P.O. Box 2, 7990AA Dwingeloo, The Netherlands

Abstract. Sharing the architectural knowledge of architectural analysis among stakeholders proves to be troublesome. This causes problems in and with architectural analysis, which can have serious consequences for the quality of a system being developed, as this quality might be incompletely or wrongly assessed. This paper presents a domain model, which can be used as a common ground among analysts and architects to capture and explicitly share such knowledge. This enables a way to overcome some of the obstacles imposed by the multi-disciplinary context in which architectural analysis takes place. To apply the domain model in practice, we have created a tool implementing (part of) this domain model for capturing and using explicit architectural knowledge during analysis. We validate the tool and domain model in the context of an industrial case study.

1 Introduction

Expectations, and therefore demands, on the quality of systems are ever increasing. More and more systems become software-intensive systems, in which software plays a crucial role in the delivery of the required functionality. Consequently, the quality of these systems is greatly influenced by the quality of the software. Software architectures offer the ability to predict the expected quality of a software system before it is actually implemented or changed [1]. This architectural analysis gives engineers a tool to find out which kind of software system is optimal for their system needs, without implementing the (changes to the) software beforehand.

Architectural Knowledge (AK) [2,3,4] plays a crucial role in architectural analysis, as it is this knowledge an analyst consumes and produces [5]. System analysts are often experts in certain domains and use or *consume* AK to analyze (parts of) an architecture. During the analysis, AK is *produced* by analysts, which ranges from individual analysis results to new insights into the overall behavior of the design space.

S. Becker, F. Plasil, and R. Reussner (Eds.): QoSA 2008, LNCS 5281, pp. 220–234, 2008.
© Springer-Verlag Berlin Heidelberg 2008

However, sharing an architectural analysis among system analysts and other relevant stakeholders proves to be problematic. To fully understand the results of an architectural analysis, the AK *consumed* to produce these documents and models is required as well, since this AK explains the reasoning path the analyst used to come up with the analysis result. Often not all of this knowledge is shared. This incomplete AK sharing can have repercussions for the quality of the analysis, thus negatively affecting the quality of the realized or changed system. Typically, these repercussions include: (1) difficult communication among stakeholders/analysts, (2) troublesome integration of analysis results of different analysts and (3) incomplete documentation and traceability of the analysis itself.

The incomplete AK sharing has two major causes. First, awareness is often missing of which AK is relevant to share. Second, the multi-disciplinary context, i.e. the very different backgrounds of the stakeholders and analysts, creates an obstacle to sharing this knowledge.

In this paper, the focus is on sharing the AK of quantitative analysis. First, the needs are identified for sharing this AK. Based on this, the AK relevant to share is identified and described in a domain model. The domain model describes this knowledge independently of the background of the stakeholders and analysts, thus creating a common ground, which to some extent can prevent issues arising from the multi-disciplinary context.

The rest of this paper is organized as follows. Section 2 presents the needs to share the AK of quantitative analysis. Section 3 presents a domain model describing the AK of quantitative analysis relevant to share. This is followed by section 4, which presents a tool for sharing this knowledge for one of the identified needs. The domain model and tool are validated with a case study in section 5. Related work is discussed in section 6 and the paper concludes with conclusions and future work in section 7.

2 Sharing the AK of Quantitative Analysis

In software architecting, the evaluation of software architectures forms an important activity [6]. Based on evaluation results, informed architectural decisions can be made. Not only does this increase the confidence in the architectural design, but also makes the design easier to defend, as objective rationale and alternatives for the architectural decisions are available. One way to obtain such results is by quantitative analysis. In quantitative analysis, one or more analysis models are created to quantitatively analyze various potential architectural solutions. The results of an analysis model are quantifications of one or more quality attributes.

Sharing the AK of quantitative analysis is required for the following three cases:

- **Integration.** For complex systems, a divide and conquer strategy is often used for quantitative analysis. The system is divided into subsystems, for each of which analysis is performed by analysts, each with their own area of expertise. Another way to divide the system is based on the relevant quality

attributes, each of which is to be analyzed by its corresponding experts. Consequently, numerous analysis models are created.

However, to evaluate a system as a whole, the different perspectives the analysts have on the architecture need to be combined. Hence, *integration* of their analysis models is required. This requires the analysis models to be compatible with each other, i.e. use the same kind of quantification and terminology for the various quality attributes. In addition, assumptions made about the solutions being evaluated in the analysis models should be synchronized to ensure that all analysis share a common ground. Thus, considerable knowledge sharing of the consumed and produced AK is required among the analysts.

- **Verification.** Another need for AK sharing comes from *verification*, i.e. the need to verify the correctness, completeness, and consistency of the analysis models. Typically, analysts let their models be reviewed by others, as to find problems with their analysis. This is especially useful if the analysis is performed on the boundary of the expertise of different analysts.
- **Validation.** A third need for AK sharing comes from stakeholders who want (or need) to *validate* a design, i.e. want to have a clearer understanding of the rationale used for an architectural decision. This knowledge can be used to convince stakeholders of the appropriateness of an architectural decision, e.g. in the form of traceability. Additionally, this can create further insight, allowing the discovery of new and potentially better alternative designs.

3 Domain Model for Quantitative Analysis

3.1 Introduction

Before the AK of quantitative analysis can be shared, we need to know what this knowledge entails. To discover this knowledge, we have investigated architectural analysis in a particular organization: Astron, the Dutch national foundation for research in Astronomy. One of their activities is performing quantitative architectural analysis of radio and optical telescopes. In the new generation of telescopes, software has become a dominant design factor. Astron analysts mainly perform quantitative performance and cost analysis (see e.g. [7]), although qualities like reliability and maintainability are quantitatively analyzed sometimes as well.

We closely cooperated with Astron analysts to find out which AK is consumed and produced during their analysis. To describe this knowledge, we have developed a domain model. The model describes the concepts and the relationships of this knowledge. It is based on informal interviews, inspection of various analysis models, software architecture documents and system analysts meetings. In addition, we were inspired by some of the concepts and insights gained from the Massive project, which delivered a tool for quantitative cost and performance analysis of embedded systems of telescopes [8]. Furthermore, the domain model has been improved in multiple iterations with the system analysts to make sure that it reflects their practice.

The aim was to come up with a unified domain model, which was independent of the individual analysts, their quantitative modeling approaches, and the qualities being analyzed, since only such a domain model would become effective in tackling the multi-disciplinary boundary by providing a common ground. Although the naming of many concepts in the domain model are specific to Astron, most of the concepts will be likely found back under different names in other organizations as well. To which extent this is the case is still an open research question.

In the remainder of this section, we present this domain for quantitative analysis. Due to the many concepts and their relationships, the model is presented in parts to enhance readability. For the full details of the domain model and a complete figure, we refer to [9].

3.2 AK Basis Model

The AK basis model presented in figure 1 is not part of the domain model for quantitative analysis. Rather, it presents the basic concepts that individual domain models extend for their particular case. In this case, it forms the foundation upon which the domain model for quantitative analysis is built. At the heart of the model is the concept of a *Knowledge Entity*: a concept, which represents the different knowledge entities found in a particular domain. For example, a requirement is a *Knowledge Entity* in a domain model for architectural design documentation. In a domain model, *all* the domain concepts inherit from *Knowledge Entity*. A domain model can define specific relationships among *Knowledge Entities*, thereby relating them to each other. How and what the semantics of these relationships are is not known to the AK basis model.

Knowledge Entities typically have one or more creators called *Authors*, who express a *Knowledge Entity* in one or more *Artifact Fragment*s. An *Artifact Fragment* identifies which part of an *Artifact* contains a (partial) description of a *Knowledge Entity*. For example, a paragraph (i.e. an *Artifact Fragment*) describing a specific stakeholder (i.e. a *Knowledge Entity*) is contained in a Word document (i.e. an *Artifact*).

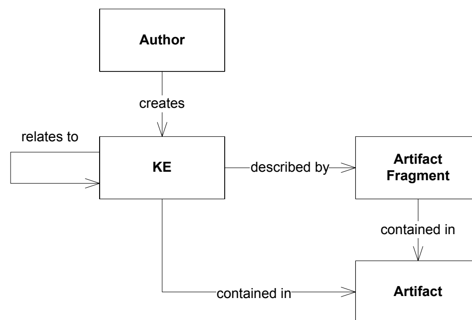


Fig. 1. The AK basis model

3.3 Quantitative Analysis Process

The domain model part that models the concepts of the quantitative analysis process is presented in figure 2. The goal of quantitative analysis is to investigate the quality of different design options, or *Alternative*s. At Astron, *Alternative*s are stratified. The first classification is in *Design Concept*s; the basic type of design. A *Design Concept* outlines a basic solution direction, thus defining a scope in the (large) design space. For radio telescopes, examples include: single dish (i.e. make one big dish), phased arrays (i.e. combine multiple antennas using beamforming), and aperture arrays (i.e. correlate the signals of telescopes and phased arrays). A *Design Concept* is specialized in the analysis in one or more *Scenario*s, which are quantitatively analyzed in one or more *Analysis Model*s.

An *Analysis model* consists of *System Parameter*s, which are the input and output of *Analysis Function*s. A *System parameter* describes the state of part of the *Analysis Model*, which is expressed in a *Number* of a unit defined by the *System Parameter*. An *Analysis Function* describes a *System Parameter*'s behavior and relationships. For example, the cost of a dish is a *System Parameter*, which can be the output of an *Analysis Function* that takes as inputs the *System Parameter*s of the costs of the various parts of a dish. Some of the *System Parameter*s can be related to one or more *Quality Attribute*s, which are defined by the quality model(s) used in the analysis. This relationship is used to classify the *System Parameter*s based on the quality they contribute to.

An *Analysis Model* is an aggregation of *System Parameters, Analysis Functions, Quality Attributes* and *Numbers*. The input of an *Analysis Model* is set by

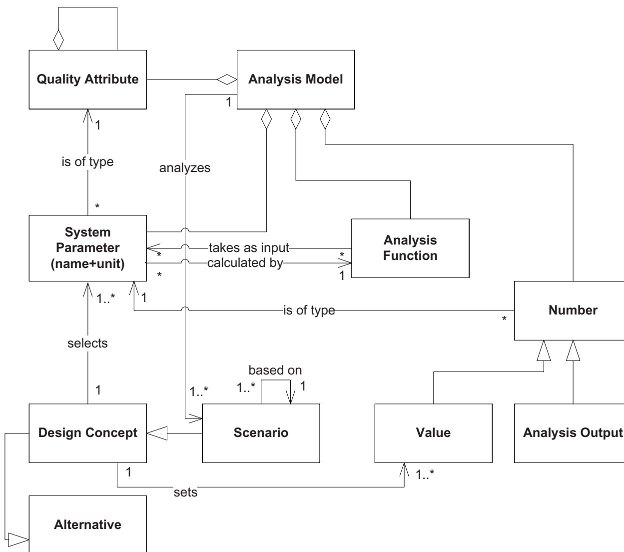


Fig. 2. Quantitative Analysis Process Concepts

assigning *Values* to input *System Parameters*. The output of an *Analysis Model* are *Analysis Outputs*, which are *Numbers* of *System parameters* calculated by *Analysis Functions*. In the example of the dish costs, the costs in dollars of the dish is an output *System Parameter* associated with the *Quality Attribute* costs and their value (i.e. a *Number*) is an *Analysis Output*. In a similar fashion, the parts of a dish are modelled as input *System parameters* and *Values*. Please remark that a distinction between input and output *System Parameters* is not explicit in the model, but rather is inferred from the relationships they have with the *Analysis Functions*.

Analysis functions have an additional property, which is not visualized in figure 2. *Analysis Models* are typically incomplete, as analysts will utilize shortcuts in their models based on their (tacit) domain knowledge and intuition. Hence to address this issue, the domain model defines a *Confidence* property for an *Analysis Function*. This property indicates to which extent the *Analysis Function* is reliable and whether the *Analysis Function* is based on intuition, trends, fact, or simply a wild guess. Consequently, points for improvements in an *Analysis Model* are made explicit.

3.4 Integration of Analysis Models

The previous subsection explained the concepts involved in quantitative analysis for a single analysis model. However, as identified in section 2 there often is a need to share individual analysis models, due to the divide and conquer strategy being used for analysis. Only when the analysis models of the parts are unified (and therefore shared), a complete architectural evaluation of an *Alternative* becomes feasible.

Figure 3 presents the concepts involved with the integration of *Analysis models*. First of all, the figure visualizes that each *Analysis Model* has its own *Namespace*. It is in bridging these *Namespaces* of various *Analysis Models* that the sharing

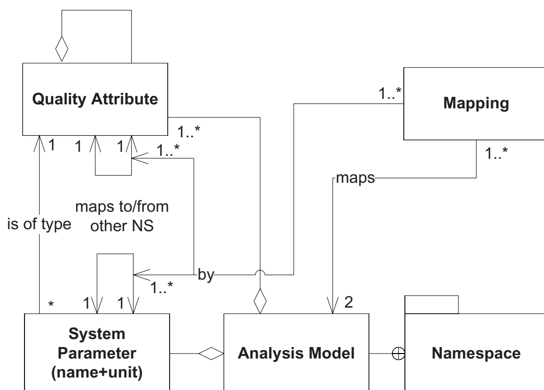


Fig. 3. Concepts for integrating Analysis Models

of AK takes place among different analysts. To integrate two analysis models, a *Mapping* should be defined between them. This involves defining mapping relationships between the *Quality Attribute*s (i.e. the quality model) and the *System Parameter*s of the two *Analysis Model*s.

One effect of these *Mapping*s is that it makes the dependencies among *Analysis Model*s visible. For example, one could identify which output *System Parameter*s of one *Analysis Model* are used in other *Analysis Models*. However, more important is the fact that an analyst can follow the translation made from his/her *System Parameter*s and *Quality Attribute*s terms into the terms used in an *Analysis Model* created by a colleague. For example, the cost of a dish could be called costs in one model and dish cost in another model. Mappings between these two *System Parameter*s identify that both denote the same concept.

Another effect of these *Mapping*s is that the overlap and gaps between *Analysis Model*s are identified. This helps analysts identify semantic inconsistencies among the *Quality Attribute*s and the *System Parameter*s of different *Analysis Model*s. For example, the *System Parameter* costs of a dish could be expressed in one *Analysis Model* in terms of millions of dollars, whereas another *Analysis Model* assumes these costs to be in thousands of dollars. *Mapping*s or their absence, makes such inconsistencies visible.

3.5 Verification of Analysis Models

Another important form of AK sharing during architectural analysis is for the purpose of verification, which is achieved in Astron by reviewing. In reviews, analysts look for problems with the *Analysis Model*s. The feedback gathered from reviews can improve the analysis quality and thus (hopefully) improve the quality of the resulting system. Figure 4 presents the concepts involved in reviewing an *Analysis Model*.

At the heart is the concept of a *Reviewable*, which denotes a concept that is relevant during reviewing. A *Reviewable* has a *Review State*, which is determined by the judgment of one or more *Reviewers*. Concepts that are *Reviewable* include: *Analysis Function*, *Mapping*, *System Parameter*, and *Components*. *Components* are a special type of *System Parameter* useful during reviewing. They provide a mechanism to group large numbers of *System Parameter*s together in a hierarchical fashion (i.e. *Components* can contain other *Components*). Different composition strategies can be used (e.g. process or deployment) to group *System Parameter*s together to create a *View/Topology* on the analysis.

The concepts of *Component* and *View/Topology* allow a *Reviewer* to judge a group of *System Parameter*s as a whole, thereby providing an explicit mechanism to review at different levels of abstraction.

3.6 Validation of Designs Using Analysis Models

Quantitative analysis is not done without a reason. The outcome plays an important role in the architectural design process. This process can be seen as a decision making process, in which an architectural decision has to be made among

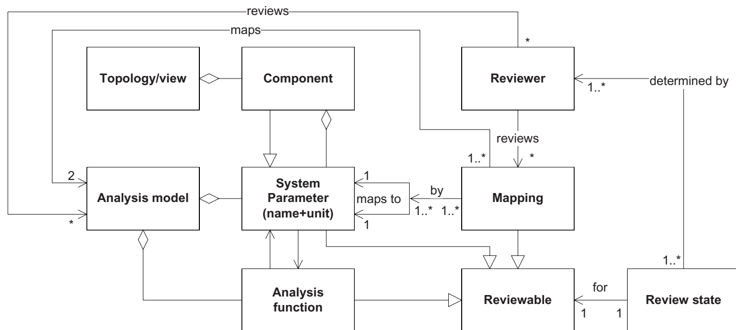


Fig. 4. Concepts of Reviewing and Quantitative Analysis

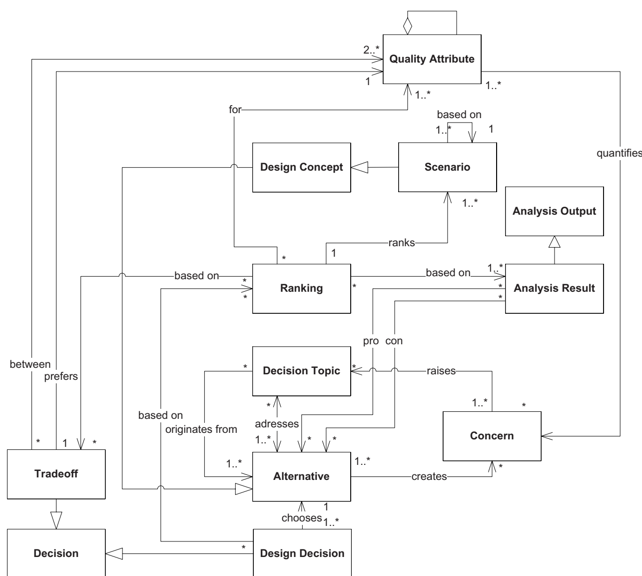


Fig. 5. The Quantitative Analysis and Design Decision process

different *Alternatives* [2,10]. The concepts involved with this decision making process and the role quantitative analysis concepts play in it are visualized in figure 5.

At the core of the decision making process are the concepts of *Decision Topics*, *Alternatives*, and *Concerns*. *Concerns* (such as requirements) raise *Decision Topics* to deal with them. The *Decision Topics* originate from various alternatives that could address the *Concern*. *Alternatives* themselves usually come with side effects, which cause new *Concerns*. When a decision is made for a certain *Alternative*, this is called a *Design Decision*.

When performing quantitative analysis, this *Design Decision* is made on the basis of quantitative knowledge. More precisely, it is based on a quantitative *Ranking* of *Scenario*s with respect to one or more *Quality Attribute*s. The values associated with these *Quality Attribute*s stem from the *Analysis Result*s of the *Scenario*s.

It is not uncommon for quality attributes to be in conflict in the context of a *Ranking*. To come to a *Design Decision*, a *Tradeoff* is made between them. The *Tradeoff* expresses a preference for a certain *Quality Attribute* for a particular *Ranking*. How these tradeoffs are exactly done can vary greatly from case to case and the design method being used.

4 The Knowledge Architect Tool Suite

Astron analysts create analysis models using both general purpose and domain specific tools. General purpose tools include Matlab, Python, Microsoft Excel, white boards and pieces of paper. An example of a domain specific tool is the Massive tool [8] for cost and performance analysis of embedded systems.

The domain model of section 3 should describe all the AK consumed and produced in these tools. To validate the domain model with respect to *verification* (see section 2), we have created the Knowledge Architect Excel plug-in tool [11], which implements the domain model for one of these general purpose tools (i.e. Microsoft Excel). The tool supports analysts in making the AK produced during analysis explicit. The aim is to facilitate the sharing of AK for verification of Excel analysis models by other analysts. The other AK sharing purposes (i.e. integration and validation) are supported by other tools, which are part of the larger Knowledge Architect tool suite.

In Excel, *System Parameter*s, *Analysis Function*s, and *Number*s have a strong relationship, as these three concepts are joint together in the form of a cell. The visible representation of the cell is normally the *Number* of a *System Parameter*. The equation bar presents the *Analysis Function* of a *System Parameter* if the appropriate cell is selected. By design, Excel does not allow for separation between these concepts. Thus the only way to have multiple *Scenario*s share the same *Analysis Function* for a *System Parameter* is by duplicating a cell.

Often labels surrounding the cell denote the semantic meaning of a cell (and thereby of a *System Parameter* and its associated *Number*). However, the texts of these labels are not formally related to any cells or *System Parameter*s. The tool allows analysts to make special annotations to make these relationships explicit. Figure 6 presents how a cell is annotated. For a *System Parameter* of a cell, a name, symbol, unit, and description can be defined. In the same window, an analyst can define the confidence of the *Analysis Function* as well. The *Author* of these *Knowledge Entities* is automatically determined.

The window presented in figure 6 does not show all of the KE types covered by the tool. A similar window exists for relating *System parameter*s to *Scenario*s. Another window allows different *Reviewer*s to define the *Review State* of a *System Parameter* and *Analysis Function* and give comments on them.

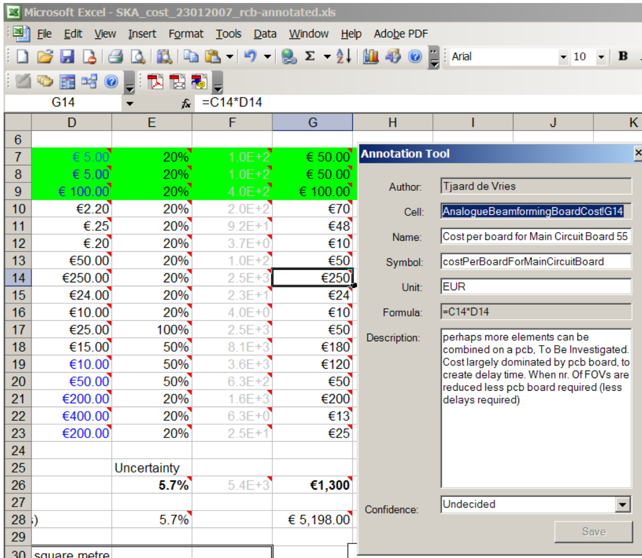


Fig. 6. Knowledge Architect Excel Plugin annotating AK in Excel

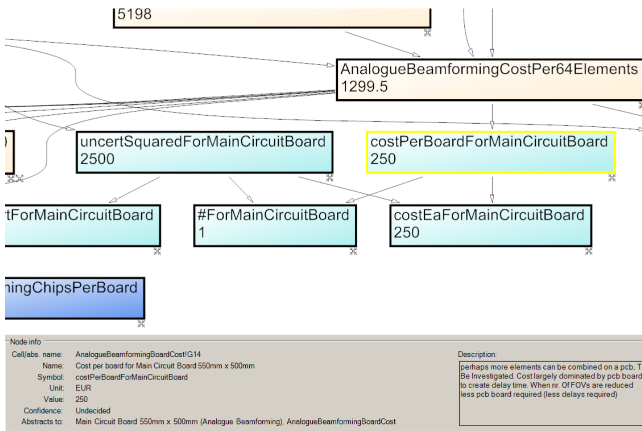


Fig. 7. An excerpt of the system parameter dependency graph

Furthermore, since annotating cells one by one is time-consuming, a feature is available to annotate whole tables of cells without much effort.

To facilitate verification, the tool offers a visualization of the dependency graph. An example of this graph is presented in figure 7. A node in the graph represents a *System Parameter* or a *Component* , i.e. a set of *System Parameters*. An arrow between two nodes indicates that the *Analysis Function*(s) of the *System Parameter*(s) of one node uses the *System Parameters* in its calculations,

thereby creating a dependency between the *System Parameter*s. The name of a node comes either from a user annotation or a default cell name. The bottom part of the figure visualizes the details of a selected node.

There is a correspondence between the nodes in the dependency graph and the cells of the Excel worksheets. Making a selection of cells in a worksheet, also selects the related nodes in the dependency graph and vice versa. An analyst can use this selection mechanism to easily create new *Component*s. The color of a node in figure 7 indicate the *Component* a *System Parameter* belongs to. A user can expand or collapse nodes, thereby providing a way to view either more abstract *System Parameter*s or more detailed ones. Orthogonal to this, is the ability to filter *System Parameter*s based on the *Scenario*s they are involved in.

5 Experiment: Sharing a SKA Cost Model

5.1 Introduction

In this section we present an experiment to find out to what extent the Knowledge Architect Excel plug-in helps in sharing AK for verification purposes. Especially, we want to know whether an analyst understands someone else's analysis model better and more quickly when aided by the tool. Furthermore, we want to know the reasons for any differences found.

The experiment takes place in the context of sharing a cost model of the Square Kilometre Array (SKA) [12]. SKA is a world wide international scientific instrument, which is still in its design phase and will consist of a radio signal collectable area of one square kilometre. In cooperation with the Jodrell Bank Centre for Astrophysics in Manchester, UK, Astron has devised a cost model for SKA in Excel. The cost model is rather complex and consists of over 1500 different *System Parameter*s spread over 12 different Excel worksheets. The various designs evaluated have costs in the order of billions (10^9) of dollars.

In this experiment, five Astron analysts were the subjects among which this cost model was shared for verification. First, the analysts were given a training to familiarize themselves with the tool. After this they were given tasks in the form of time-boxed questions, one set to be completed with and one to be completed without the tool. Besides timing the participants, we observed and debriefed the analysts to gather additional qualitative data. The questions used are representative of questions analysts may ask during the verification of analysis models. The questions are divided into the following three classes that denote activities done during verification: (1) **Sensitivity analysis**. This type of task requires an analyst to investigate the dependencies among *System parameter*s to verify their correct interaction. (2) **Consistency analysis**. This type of task requires an analyst to verify whether the same kind of *Analysis Function*s are used for similar *System Parameter*s for different *Scenario*s. (3) **Defect analysis**. This type of task requires an analyst to spot errors in the *Analysis Function*s, their relationships and the assumed *System Parameter*s for a *Scenario*.

In the remainder of this section, we present the lessons learned based on our qualitative results. For more information about the experiment, we refer to [9].

5.2 Lessons Learned

The experiment confirmed that verifying complex analysis models in detail is not an easy task. The analysts mentioned having difficulties with understanding the used terminology inside the cost model. Although the tool did help the subjects to understand the concepts used, it failed to overcome differences in terminology used for the names and descriptions of the *System Parameter*s. Integrating of analysis models, as described in section 3.4, might help in addressing this issue. Since such an integration creates a deeper understanding among the analysts of the differences in terminology they use within their models.

The navigation capabilities of the dependency graph were found useful to find relevant parts of the analysis model. However, for understanding the analysis itself, the vast multitude of arrows and nodes in the graph was too confusing. Surprisingly enough, this did not matter to the co-author of the cost model; he could directly spot some defects simply by looking at the dependency graph. This indicates that he has some tacit knowledge for filtering out irrelevant information so that he was able to use the tool to quickly get new insights into his own work.

It appeared that the cost model heavily relied on tables. For two-dimensional data, tables in a spreadsheet are a pretty good representation, but for more complex relations between system parameters the tool had a clear edge over plain Excel. It made non-trivial relationships explicit, clearly visible and thus easier to inspect for the subjects than in a spreadsheet.

Overall, it appears that the tool has potential to give extra insight into analysis models, but in its current form fails to deliver in all cases. The domain specific naming and description of the *System Parameter*s is one cause for this. Another cause is that the system parameter graph is too complex, i.e. it fails to reduce the complexity of the analysis model. Improvements in the visualization and the way non-relevant information is filtered out should help with addressing this issue. On a positive side, the tool did help analysts quickly locate relevant parts during verification and authors of a domain model with locating defects and inconsistencies.

6 Related Work

For software architecture evaluation, two types of approaches can be discerned [13]: scenario-based [14], and quantitative model based methods. In scenario-based methods (e.g. ATAM [15], ALMA [16], SALUTA[17], PASA [18] (performance), scenarios are used to define use-cases of the typical and expected future uses of the system. Based on these scenarios one or more architectural designs are evaluated for the qualities of interest. ATAM [15] is a framework method that incorporates the results of other scenario based evaluation methods and focusses on making tradeoffs between different qualities. From a knowledge perspective, this is the decision part of the domain model.

ALMA [16] provides an analysis method for modifiability, likewise SALUTA [17] does this for usability. PASA [18] combines a scenario-based approach with a

quantitative model for performance. PASA requires quantitative goals, performs the analysis quantitatively where possible, and includes a cost-benefit analysis as one of its steps.

A general approach to quantitative architectural analysis for multiple quality attributes is proposed by Bachmann et al. in the form of so-called *reasoning frameworks* [19]. A reasoning framework is a quantitative analysis model with respect to a certain quality attribute. It proposes architectural tactics to improve the architecture with respect to this quality attribute. In essence, a reasoning framework is a quantitative analysis model. Hence, our domain model should describe the AK concepts used in them.

Massive [8,20] is an analytic Layered Queueing Network method aimed at the design of large and complex embedded systems, which focusses on quantitative cost and performance analysis. The approach emphasizes reuse of components and the topology of the system being designed. Our domain model abstracts some of the core concepts of this approach and presents a more precise identification of the concepts involved.

Compared to other AK meta-models and tools, e.g. the Core model [2], AREL [21], Pakme [22], ADDSS[23], the domain model has an extensive description of quantitative AK. The other meta-models exclusively focus on qualitative rationale with no to little attention on how this rationale relates to quantitative AK.

Farenhorst et al. [24] identify difficulties that arise when sharing AK, as well as prerequisites for sharing to be successful. They define incentives for people needed to be willing to share AK, as well as the lesson that striving for completeness is infeasible. However, they do not define the actual knowledge to be shared, as is done in our domain model.

7 Conclusions and Future Work

This paper identified three different needs (i.e. integration, verification, and validation) for sharing the AK of quantitative analysis. For each need, the presented domain model describes the concepts and relationships of the relevant AK. Based on this domain model, a tool was created to facilitate AK sharing for verification. In an experiment, this tool and the domain model were tested.

Based on the lessons learned in this experiment, we can conclude that there is a close relationship between the knowledge needed for verification and integration. Although the domain model provides a common language for the concepts, it does not provide a common language for the instances. They are still domain specific, thus synchronization at this level is still needed for both integration and verification. To what extent this is also the case for validation is an open question for future work.

Another open question has to do with the perceived complexity of an analysis model. An interesting starting point for this is an observation made during the experiment. The co-author of the analysis model used was capable of dealing with this complexity, whereas others were not. The question is which tacit knowledge was involved in this.

How well the presented domain model is generally applicable is an open question as well. For this, we plan to investigate how concepts found in another organization map to ones in the presented domain model. This will provide us with insight into the extent to which the domain model is generally applicable.

In future work, we plan to improve our tool based on the outcomes of these questions. In addition, we plan to create tools, as part of the Knowledge Architect platform, for sharing AK for validation and integration purposes.

Acknowledgements

This research is sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. We thank the people from Astron who participated in this research. In particular, Kjeld van der Schaaf and Albert-Jan Boonstra.

References

1. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2nd edn. Addison-Wesley, Reading (2003)
2. de Boer, R.C., Farenhorst, R., Lago, P., van Vliet, H., Jansen, A.G.J.: Architectural knowledge: Getting to the core. In: Overhage, S., Szyperski, C.A., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880. pp. 197–214. Springer, Heidelberg (2008)
3. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214. Springer, Heidelberg (2006)
4. Habli, I., Kelly, T.: Capturing and replaying architectural knowledge through derivational analogy. In: SHARK-ADI 2007: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent, Washington, DC, USA, p. 4. IEEE Computer Society, Los Alamitos (2007)
5. Lago, P., Avgeriou, P.: First workshop on sharing and reusing architectural knowledge. SIGSOFT Software Engineering Notes 31(5), 32–36 (2006)
6. Clements, P., Rick Kazman, M.K.: Evaluating Software Architectures - Methods and Case Studies. The SEI Series in Software Engineering. Addison-Wesley, Reading (2002)
7. Alliot, S.: A performance cost estimation model for large scale array signal processing system specification. In: Proc. of the Third International Samos Workshop on Synthesis, Architectures, and Simulation, pp. 156–160 (July 2003)
8. Alliot, S., Nicolae, L., van Veelen., M.: A tool for exploring the large scale signal processing systems specifications. In: IEEE International conference on parallel computing in electrical engineering, pp. 341–348 (September 2004)
9. de Vries, T., Jansen, A.G.J.: Knowledge architect excel plug-in technical report. Technical Report IWI preprint 2008-7-01, Department of Mathematics and Computing Science, University of Groningen, PO Box 800, 9700 AV The Netherlands (March 2008)

10. Jansen, A.G.J., Bosch, J., Avergiou, P.: Documenting after the fact: recovering architectural design decisions. Journal of Systems and Software 81(4), 536–557 (2008)
11. The Griffin project website, http://search.cs.rug.nl/Griffin
12. The Square Kilometre Array project website, http://www.skatelescope.org/
13. Babar, M.A., Gorton, I.: Comparison of scenario-based software architecture evaluation methods. In: Software Engineering Conference, 2004. 11th Asia-Pacific, 30 November-3 December, pp. 600–607 (2004)
14. Dobrica, L., Niemela, E.: A survey on software architecture analysis methods. IEEE Trans. Softw. Eng. 28(7), 638–653 (2002)
15. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures, Views and Beyond. Addison-Wesley, Reading (2002)
16. Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H.: Architecture-level modifiability analysis (ALMA). J. Syst. Softw. 69(1-2), 129–147 (2004)
17. Folmer, E., van Gurp, J., Bosch, J.: Software architecture analysis of usability. In: 9th IFIP Working Conference on Engineering for Human-Computer Interaction, pp. 321–339 (July 2004)
18. Williams, L.G., Smith, C.U.: Pasa: a method for the performance assessment of software architectures. In: WOSP 2002: Proceedings of the 3rd international workshop on Software and performance, Rome, Italy, pp. 179–189. ACM Press, New York (2002)
19. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing software architectures to achieve quality attribute requirements. IEE Proceedings - Software 152(4), 153–165 (2005)
20. Alliot, S., M.: Modelling and system design for the lofar station digital processing. In: SPIE Astronomical Telescopes and Instrumentation, Modelling and System Engineering (June 2004)
21. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
22. Babar, M.A., Gorton, I., Kitchenham, B.: A framework for supporting architecture knowledge and rationale management. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (eds.) Rationale Management in Software Engineering, pp. 237–254. Springer, Heidelberg (2006)
23. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. SIGSOFT Software Engineering Notes 31(5) (2006)
24. Farenhorst, R., Lago, P., van Vliet, H.: Prerequisites for successful architectural knowledge sharing. In: ASWEC 2007: Proceedings of the 2007 Australian Software Engineering Conference, pp. 27–38. IEEE Computer Society, Los Alamitos (2007)

Author Index

Adámek, Jiří 71
Ardagna, Danilo 1
Avgeriou, Paris 220

Babar, Muhammad Ali 189
Bachmann, Felix 171
Bass, Len 171
Bennicke, Marcel 205
Bianco, Phil 171

Cortellessa, Vittorio 86

de Vries, Tjaard 220
Diaz-Pace, Andres 171
Duchien, Laurence 152

Engels, Gregor 205

Gallotti, Stefano 119
Ghezzi, Carlo 1, 119
Gorton, Ian 189

Han, Jun 28

Jansen, Anton 220

Kim, Hyunwoo 171
Kruchten, Philippe 43

Le Meur, Anne-Françoise 152
Lee, Larix 43
Lewerentz, Claus 205
Liu, Yan 189

Mallet, Julien 55
Merson, Paulo 135
Mirandola, Raffaela 1, 119
Moreno, Gabriel A. 135

Pierini, Pierluigi 86
Plšek, Aleš 71

Raadt, Bas van der 103
Rouvrais, Siegfried 55

Salger, Frank 205
Spalazzese, Romina 86

Tamburrelli, Giordano 119
Tang, Antony 28
Tran, Minh H. 28

van Veelen, Martijn 220
van Vliet, Hans 28, 103
Vianale, Alessio 86

Waignier, Guillaume 152