Volker Gruhn
Flavio Oquendo (Eds.)

LNCS 4344

Software Architecture

Third European Workshop, EWSA 2006
Nantes, France, September 2006
Revised Selected Papers

Springer

Lecture Notes in Computer Science 4344

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

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

Moshe Y. Vardi
Rice University, Houston, TX, USA

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

Volker Gruhn Flavio Oquendo (Eds.)

Software Architecture

Third European Workshop, EWSA 2006
Nantes, France, September 4-5, 2006
Revised Selected Papers

Springer

Volume Editors

Volker Gruhn
University of Leipzig
Applied Telematics / e-Business
Klostergasse 3, 04109 Leipzig, Germany
E-mail: volker.gruhn@informatik.uni-leipzig.de

Flavio Oquendo
University of South Brittany
VALORIA – Formal Software Architecture and Process Research Group
B.P. 573, 56017 Vannes Cedex, France
E-mail: flavio.oquendo@univ-ubs.fr

Library of Congress Control Number: 2006938908

CR Subject Classification (1998): D.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-69271-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-69271-3 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 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11966104 06/3142 5 4 3 2 1 0

Preface

Following the successful workshops held in St. Andrews, Scotland, UK in 2004 (EWSA 2004, Springer LNCS 3527) and in Pisa, Italy in 2005 (EWSA 2005, Springer LNCS 3047), the 3rd European Workshop on Software Architecture (EWSA 2006) held in Nantes, France during September 4–5, 2006 provided an international forum for researchers and practitioners from academia and industry to present innovative research and discuss a wide range of topics in the area of software architecture.

Software architecture has emerged as an important subdiscipline of software engineering encompassing a broad set of languages, styles, models, tools, and processes. The role of software architecture in the engineering of software-intensive systems has become more and more important and widespread. Challenging applications include support for dynamic, adaptive, autonomic and mobile systems.

The workshop focused on formalisms, technologies, and processes for describing, verifying, validating, refining, building, and evolving software systems, in particular based on component and service-oriented architectures. Topics covered included architecture modeling, architectural aspects, architecture analysis, transformation and synthesis, quality attributes, model-driven engineering, and architecture-based support for assembling components and developing component and service-oriented systems.

EWSA 2006, distinguished between two types of papers: research papers, which describe authors novel research work, and position papers, which present concise arguments about a topic of software architecture research or practice.

The Program Committee selected 18 papers (13 research papers and 5 position papers) out of 53 submissions from 25 countries (Algeria, Austria, Belgium, Brazil, Canada, Chile, China, Czech Republic, Finland, France, Germany, India, Ireland, Italy, Norway, Oman, Poland, Portugal, Russia, Spain, Switzerland, The Netherlands, Tunisia, UK, USA). All submissions were reviewed by three members of the Program Committee. Papers were selected based on originality, quality, soundness and relevance to the workshop. Credit for the quality of the proceedings goes to all the authors. In addition, the workshop included an invited talk by Richard N. Taylor (University of California, Irvine, USA) and a panel on Research Directions.

We would like to thank the members of the Program Committee for providing timely and significant reviews and for their substantial effort in making EWSA 2006 a successful workshop.

As with EWSA 2004 and 2005, the EWSA 2006 submission and review process was extensively supported by the Paperdyne Conference Management System. We are indebted to Clemens Schäfer for his outstanding support.

EWSA 2006 was held in conjunction with the 1st French Conference on Software Architecture (CAL 2006). We acknowledge Tahar Khammaci, Dalila Tamzalit and the other members of both Organizing Committees for their excellent service.

Finally, we acknowledge the prompt and professional support from Springer, which published these proceedings in printed and electronic volumes as part of the *Lecture Notes in Computer Science* series.

September 2006

Volker Gruhn
Flavio Oquendo
Program Committee Chairs

Mourad Oussalah
Organizing Chair

Organization

Program Committee

Program Chairs

Volker Gruhn	University of Leipzig, Germany gruhn@ebus.informatik.uni-leipzig.de
Flavio Oquendo	University of South Brittany – VALORIA, France flavio.oquendo@univ-ubs.fr

Committee Members

Dharini Balasubramaniam	University of St. Andrews, UK
Thais Batista	University of Rio Grande do Norte – UFRN, Brazil
Rafael Capilla	Universidad Rey Juan Carlos, Spain
José A. Carsí	Technical University of Valencia, Spain
Carlos E. Cuesta	Universidad Rey Juan Carlos, Spain
Rogerio de Lemos	University of Kent, UK
Ian Gorton	National ICT, Australia
Susanne Graf	VERIMAG, France
Mark Greenwood	University of Manchester, UK
Paul Grefen	Eindhoven University of Technology, Netherlands
Wilhelm Hasselbring	University of Oldenburg, Germany
Valérie Issarny	INRIA Rocquencourt, France
René Krikhaar	Vrije Universiteit Amsterdam and Philips Medical Systems, Netherlands
Philippe Kruchten	University of British Columbia, Canada
Nicole Levy	University of Versailles-St.-Quentin – PRiSM, France
Antonia Lopes	University of Lisbon, Portugal
Radu Mateescu	INRIA Rhône-Alpes and ENS Lyon, France
Carlo Montangero	Università di Pisa, Italy
Ron Morrison	University of St. Andrews, UK
Robert Nord	Software Engineering Institute, USA
Dewayne E. Perry	University of Texas at Austin, USA
Frantisek Plasil	Charles University Prague, Czech Republic
Ralf Reussner	University of Karlsruhe, Germany
Salah Sadou	University of South Brittany – VALORIA, France
Clemens Schäfer	University of Leipzig, Germany
Bradley Schmerl	Carnegie Mellon University, USA
Clemens Szyperski	Microsoft Research, USA

Dalila Tamzalit	University of Nantes – LINA, France
Brian Warboys	University of Manchester, UK
Eoin Woods	UBS Investment Bank, UK

Organizing Committee

Mourad Oussalah (Chair)	University of Nantes – LINA, France mourad.oussalah@univ-nantes.fr
Dalila Tamzalit	University of Nantes – LINA, France
Djamel Seriai	Ecole des Mines de Douai, Douai, France
Tahar Khammaci	University of Nantes – LINA, France
Olivier Le Goaer	University of Nantes – LINA, France
Adel Smeda	University of Nantes – LINA, France

Steering Committee

Flavio Oquendo	University of South Brittany – VALORIA, France
John Favaro	Consorzio Pisa Ricerche, Italy
Volker Gruhn	University of Leipzig, Germany
Ron Morrison	University of St. Andrews, UK
Mourad Oussalah	University of Nantes – LINA, France
Brian Warboys	University of Manchester, UK

Additional Reviewer

Apostolos Zarras	University of Ioannina, Greece

Table of Contents

Invited Talk

Primacy of Place: The Reorientation of Software Engineering Demanded by Software Architecture 1
Richard N. Taylor

Research Papers

Fault Tolerant Web Service Orchestration by Means of Diagnosis 2
Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan

Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems 17
Marco Autili, Michele Flammini, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli

Introspective Model-Driven Development 33
Thomas Büchner and Florian Matthes

Eliminating Execution Overhead of Disabled Optional Features in Connectors 50
Lubomír Bulej and Tomáš Bureš

Automated Selection of Software Components Based on Cost/Reliability Tradeoff 66
Vittorio Cortellessa, Fabrizio Marinelli, and Pasqualina Potena

On the Modular Representation of Architectural Aspects 82
Alessandro Garcia, Christina Chavez, Thais Batista, Claudio Sant'anna, Uirá Kulesza, Awais Rashid, and Carlos Lucena

Configurations by UML 98
Øystein Haugen and Birger Møller-Pedersen

Modes for Software Architectures 113
Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastian Uchitel

Integrating Software Architecture into a MDA Framework 127
Esperanza Marcos, Cesar J. Acuña, and Carlos E. Cuesta

Layered Patterns in Modelling and Transformation of Service-Based Software Architectures 144
Claus Pahl and Ronan Barrett

Towards MDD Transformations from AO Requirements into AO Architecture 159
Pablo Sánchez, José Magno, Lidia Fuentes, Ana Moreira, and João Araújo

Modeling and Analyzing Mobile Software Architectures 175
Clemens Schäfer

Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design 189
Holger Schmidt and Ina Wentzlaff

Position Papers

Identifying "Interesting" Component Assemblies for NFRs Using Imperfect Information 204
Hernán Astudillo, Javier Pereira, and Claudia López

Towards More Flexible Architecture Description Languages for Industrial Applications 212
Rabih Bashroush, Ivor Spence, Peter Kilpatrick, and John Brown

Architecture Transformation and Refinement for Model-Driven Adaptability Management: Application to QoS Provisioning in Group Communication 220
Christophe Chassot, Karim Guennoun, Khalil Drira, François Armando, Ernesto Exposito, and André Lozes

Automating the Building of Software Component Architectures 228
Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard

Component Deployment Evolution Driven by Architecture Patterns and Resource Requirements 236
Didier Hoareau and Chouki Tibermacine

Author Index 245

Primacy of Place: The Reorientation of Software Engineering Demanded by Software Architecture

Richard N. Taylor

University of California, Irvine, USA
taylor@ics.uci.edu
http://www.ics.uci.edu/~taylor/

Abstract. Software architecture is a powerful technology that has proven itself in numerous domains. It has been used, for example, to shape the contemporary World Wide Web and has provided the basis for the economic exploitation of the notion of product families. In far too many development organizations, however, consideration of software architecture is relegated to a specific time-period, or phase, of software development. This talk considers how software architecture relates to the classical conceptions of software development. What emerges is a substantial reorientation of software engineering, for the power of architecture demands a primacy of place. With architecture as a central focus the very character of key software engineering activities, such as requirements analysis and programming, are altered and the technical approaches taken during development activities are necessarily changed.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, p. 1, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Fault Tolerant Web Service Orchestration by Means of Diagnosis*

Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan

Dipartimento di Informatica - Università di Torino
Corso Svizzera 185, 10149 Torino - Italy
{liliana,furnari,goy,giovanna,marino}@di.unito.it

Abstract. Web Service orchestration frameworks support a coarse-grained kind of exception handling because they cannot identify the *causes* of the occurring exceptions as precisely as needed to solve problems at their origin.

This paper presents a framework for Web Service orchestration which employs diagnostic services to support a fine grained identification of the causes of the exceptions and the consequent execution of effective exception handlers. Our framework is particularly suitable for intelligent exception handling in Enterprise Application Integration.

1 Introduction

The importance of Enterprise Application Integration is growing due to the emerging demand for short software development time, and to the fact that several services have to be developed by composing heterogeneous applications owned by different organizations. Workflow management systems have originally been developed to coordinate the execution of tasks within a single organization. Later on, Service Oriented Architectures [16], Web Service description languages (such as WSDL [19]) and Web Service composition languages (such as WS-BPEL [14]) have been introduced in order to abstract from location, communication protocol and deployment environment issues, therefore supporting the integration of distributed, heterogeneous software in open environments.

In current workflow management and Web Service orchestration environments, the development of fault tolerant distributed processes is based on the adoption of exception handling techniques which support a graceful termination, rather than the service continuation, as the recovery actions are associated to the observable effects of the occurred problems, rather than to their *causes*. Indeed, effective recovery strategies might be adopted to let the process progress towards the service completion, or to support the human user in performing ad hoc corrective actions, if the *causes* of the exceptions were recognized.

We thus propose to support *intelligent exception management* in Web Service orchestration environments by introducing:

* This work is supported by the EU (project WS-Diamond, grant IST-516933) and by MIUR (project QuaDRAnTIS).

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 2–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

- *Diagnostic capabilities* supporting the identification of the causes of the exceptions occurring during the execution of a composite service. The analysis of the exceptions is carried out by *diagnostic Web Services* which explain the possibly incorrect execution of the orchestrated service providers by employing Model-Based Diagnosis techniques [1].
- *A novel methodology to apply exception handlers*, in order to make them sensitive to diagnostic information without modifying the standard exception management mechanisms offered by Web Service orchestration environments.

The rest of this paper is organized as follows: Section 2 outlines exception handling in workflow systems and provides some background on Model-Based diagnosis. Section 3 presents the architecture of our framework. Section 4 describes some related work and Section 5 concludes the paper.

2 Background

In Web Service orchestration, the development of fault tolerant distributed processes relies on the introduction of scopes, exception handlers and compensation handlers. When a scope fails, compensation handlers are executed to undo the completed activities. Moreover, exception handlers are performed to enable forward progress toward the termination of the process; e.g. see [14,11].

Several classifications of exceptions in categories have been made; e.g., see [9], [18] and [13]. These studies report a wide variety of events triggering the exceptions. For instance, [9] introduces different types of exceptions, among which the *expected* and the *unexpected* ones. *Expected exceptions* are associated to anomalies frequently occurring during the execution of activities, and should be managed by means of exception handlers. *Unexpected exceptions* derive from unexpected problems and typically have to be handled via human intervention, at the workflow instance level.

Various approaches have been proposed to handle the exceptions. For instance, [11] presents a technique based on *spheres of atomicity* to handle transactions. Moreover, [13] presents strategies enabling human users to participate in the recovery of a workflow instance during the service execution. Furthermore, [9] proposes a classification of activity types on the basis of factors such as the presence/absence of side-effects in compensation, and the development of a smart execution engine which may ignore the failure of *non vital* actions during the execution of a workflow.

Before concluding this section we would like to briefly introduce Model-Based Diagnosis (MBD), which we adopt in our framework to reason about exceptions. Model-Based Reasoning and, in particular, MBD, have been proposed and used within the Artificial Intelligence community for reasoning on possibly faulty physical and software systems; e.g., see [5,10]. In both cases, the system to be diagnosed is modeled in terms of components, and the goal of *diagnostic reasoning* is to determine a set of components whose incorrect behavior *explains* a given set of observations.

There are several formalizations of MBD that characterize the informal notion of explanation. In *consistency-based diagnosis* [17], which we utilize in our work, a *diagnosis* is an assignment of behavior modes to components that is consistent with observations. For static models this means that the hypotheses predict, for observable variables,

a set of possible values which includes the observed one. A *diagnostic engine* should, in general, explore the space of candidate diagnoses and perform discrimination among alternative candidates, possibly suggesting additional pieces of information to be acquired to this purpose.

3 Intelligent Exception Management Framework

We consider two types of failures: errors in data elaboration, and errors that either alter or block the execution flow. Our exception handling approach relies on:

- A *smart failure identification* aimed at identifying the cause of an observed exception, at the level of the composite service (global diagnosis). The goal is to identify the Web Service responsible for the occurred problem, the faulty activities and the other Web Services that may have been involved in the failure.
- *Diagnostic information aware exception handlers*, executed by the orchestrated Web Services. The global diagnosis is used to identify the recovery strategy to be applied by the service providers.

Consistently with Model-Based Diagnosis, the analysis of the exceptions is carried out in a component-based way[1] by introducing local diagnostic services that analyze the internal behavior of the orchestrated Web Services, and by employing a global diagnostic engine to combine the local diagnoses.

Each local diagnoser analyzes the behavior of the corresponding orchestrated Web Service WS_i by utilizing a *diagnostic model* M_i which describes the control and data flow of WS_i by specifying the possible correct and incorrect behavior. The *diagnostic model* of a Web Service is a declarative description of its business logic and it specifies, as precisely as possible, the activities carried out by the Web Service, the messages it exchanges, information about dependencies between parameters of the executed activities and the fault messages it may generate [1].

Web Services have rather diverse nature: some may execute articulated workflows; others may partially hide their internal business logic. Moreover, they may be implemented in different process languages, such as WS-BPEL [7], or the OPERA [11], XPDL [20] and BPML [3] workflow description languages. We introduced the diagnostic model as a separate representation of the business logic of a Web Service for two main purposes:

- The first one is to make local diagnosers independent of the implementation language adopted within the orchestrated Web Services.
- The second one concerns the possibility of extending the description of the business logic of a Web Service with information useful for diagnostic reasoning, without modifying the core of the Web Service; e.g., *alarm conditions* may be specified to enable the efficient isolation of the faulty activities during the Web Service execution.

[1] Notice that the orchestrated Web Services interact only by message passing. Thus, they do not share any data items which might influence each other's behavior as a side-effect.

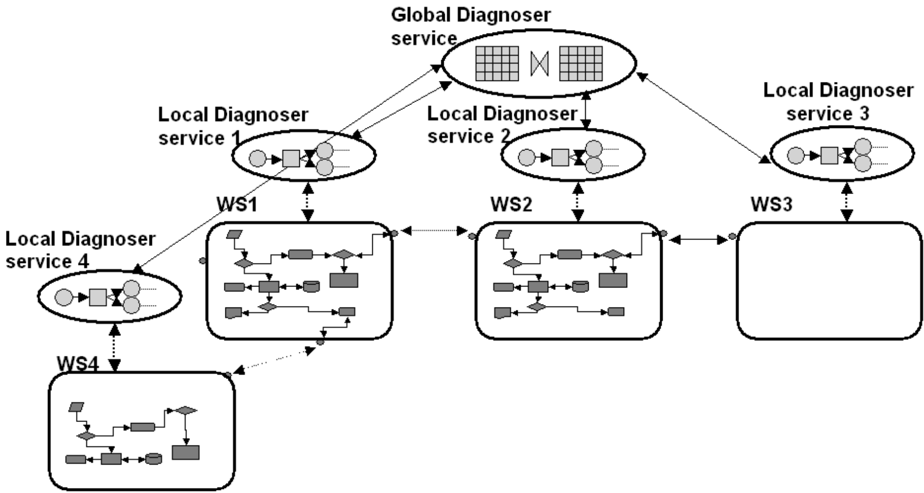


Fig. 1. Architecture of the Proposed Framework

3.1 Architecture

Figure 1 shows the proposed architecture in a composite service based on the orchestration of four Web Services. The orchestrated Web Services are depicted as rectangles with rounded corners, while the Diagnoser Web Services are represented as ovals. The dotted double arrows between Web Services and Local Diagnosers represent the asynchronous messages exchanged during both the identification of failures and the selection of the exception handlers to be executed.

- A Local Diagnoser service LD_i is associated to each orchestrated Web Service WS_i, in order to generate diagnostic hypotheses explaining the occurred exceptions from the local point of view.
  - To this purpose, LD_i utilizes the *diagnostic model* of WS_i. Moreover, LD_i interacts with WS_i by invoking a set of WSDL operations defined for diagnosis; Figure 2 shows the additional WSDL operations (described later on) that must be offered by an orchestrated Web Service in order to interact with its own Local Diagnoser.
  - The local hypotheses generated by LD_i specify various types of information, such as the correctness status of the input and output parameters of the operations performed by WS_i and the references to other orchestrated Web Services $\{WS_1, ..., WS_k\}$ which might be involved in the failure of the composite service. Errors may propagate from one Web Service to the other via message passing; thus, $\{WS_1, ..., WS_k\}$ is the set of orchestrated Web Services that have sent and/or received messages from WS_i.
- Global reasoning about the composite service is performed by a Global Diagnoser service which interacts with the Local Diagnosers of the orchestrated Web

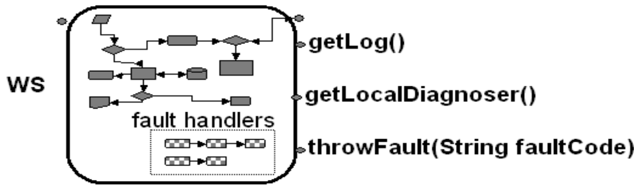


Fig. 2. Web Service Extended for Advanced Exception Management

Services. The Global Diagnoser combines the local hypotheses generated by Local Diagnosers into one or more global diagnostic hypotheses about the causes of the occurred exceptions.

Local and Global Diagnosers are themselves Web Services communicating via WSDL messages. This component-based approach has various advantages. For instance:

– Local Diagnosers may be associated to Web Services in a stable way; therefore, the diagnostic model of a Web Service WS_i has to be defined only once, at Local Diagnoser LD_i configuration time (although, as described later on, the management of diagnosis aware exception handlers requires the introduction of supplementary information, which is specific for the composite service invoking WS_i).
– Privacy preferences possibly imposed by the organizations owning the orchestrated Web Services may be satisfied. In fact, the diagnostic model of a Web Service WS_i can only be inspected by its Local Diagnoser LD_i, which does not directly interact with the Local Diagnosers of the other orchestrated Web Services. Moreover, the information provided by LD_i during the interaction with the Global Diagnoser concerns the presence of errors in the data received from, or propagated to other Web Services, omitting internal details of the Web Service implementation.
– Due to the clear separation between Local and Global Diagnosers, the latter may be developed as general services supporting diagnosis in different composite services; in fact, a Global Diagnoser only makes assumptions on the communication protocol to be adopted in the interaction with the Local Diagnosers.

3.2 Smart Failure Identification

Diagnosis is needed when exceptions occur, but can be avoided when the composite service progresses smoothly. Thus, we propose to trigger diagnosis immediately after a problem is detected.

Specifically, when an exception occurs in a Web Service WS_i, its Local Diagnoser LD_i is invoked to determine the causes. To this purpose, LD_i exploits the diagnostic model M_i and may need to analyze the messages exchanged by WS_i with the other peers (e.g., to check the parameters of the operations on which WS_i was invoked). LD_i retrieves this information by requesting from WS_i the log file of its activities, i.e., by invoking the `LogFile getLog()` WSDL operation; see Figure 2. After the generation of the local hypotheses, LD_i invokes the Global Diagnoser (`activate(Collection`

`hypotheses)` WSDL operation) to inform it about the local hypotheses it made on the causes of the exception. The Global Diagnoser, triggered by the incoming message, starts an interaction with other Local Diagnosers to identify the problem. The double arrows between Diagnosers in Figure 1 represent the synchronous messages exchanged in this phase.

The interaction between Global Diagnoser and Local Diagnosers is managed as a loop where the Global Diagnoser invokes different Local Diagnosers to make them generate their own local hypotheses, which are incrementally combined into a set of global diagnostic hypotheses about the occurred problem. Below, we summarize the actions performed in the loop.

- By analyzing the local hypotheses it has received, the Global Diagnoser identifies a list of Local Diagnosers to be invoked in order to make them generate their local hypotheses.[2] It then invokes the `Collection extend(Collection hypotheses)` WSDL operation on each Local Diagnoser in the list. This operation, given a set of hypotheses, returns a revised set of hypotheses to the caller.
- The Global Diagnoser combines the local hypotheses and generates a set $\mathcal{H}$ of global hypotheses about the causes of the occurred exceptions, consistent with the local hypotheses; in Figure 1 the local hypotheses have been depicted as tables.

The Global Diagnoser exits the loop when it cannot invoke any further Local Diagnosers. This happens either because all of them have contributed to the diagnosis, or because the remaining ones cannot provide any discriminating information, nor can they broaden the search for the causes of the exception. This means that, although several messages might be exchanged by the Local and Global Diagnosers, the diagnostic process always terminates.

The set $\mathcal{H}$ obtained by the Global Diagnoser at the end of the loop represents the global result of diagnosis about the exception occurred in the composite service. $\mathcal{H}$ can be seen as the solution of a Constraint Satisfaction Problem [8] where constraints express the relation between failures and service behavior. See [1] for details about the adopted diagnostic algorithm.

3.3 Diagnostic Information Aware Exception Handling

Local and Global Exceptions. The causes of exceptions reported in the result $\mathcal{H}$ of the global diagnosis can be employed at the level of the composite service for the selection of specific exception and compensation handlers, which might substantially differ from those that would be adopted by the orchestrated Web Services on the sole basis of the locally raised exceptions. The cardinality of $\mathcal{H}$ has to be considered:

[2] The `activate(Collection hypotheses)` message specifies the references of the orchestrated Web Services possibly involved in the failure. By invoking such Web Services, the Global Diagnoser retrieves the references of the Local Diagnosers to be contacted in order to acquire further information. The reference to the Local Diagnoser of a Web Service is obtained by invoking the `LocalDiagnoser getLocalDiagnoser()` WSDL operation; see Figure 2.

(a) If $\mathcal{H}$ is a singleton, the Global Diagnoser could find a single cause (h) explaining the occurred exceptions. The Local Diagnosers might need to inhibit the default exception handling to take h into account; in order to make the Web Services execute the appropriate handlers, Local Diagnosers should make Web Services throw *global exceptions* activating the handlers needed to repair h, instead of continuing the execution of the handlers associated to their local exceptions. Of course, if the default exception handlers are suitable to repair h, they should continue their regular execution; this corresponds to the case where the global exception coincides with the local one that was raised in the Web Service.

(b) If $\mathcal{H}$ includes more than one global diagnostic hypothesis, it should be reduced to a singleton (e.g., via human intervention). If this is not possible, default fault management behavior should be adopted. We leave this aspect apart as it concerns the diagnostic algorithm, which is out of the scope of this paper.

The execution of exception handlers depending on diagnostic information requires that, when the composite service is set up, the possible diagnostic hypotheses are mapped to global exceptions to be handled in the overall service. Specifically, for a complex service CS, the Local Diagnoser LD_i of a Web Service WS_i must store a set of mappings $map(h_x, e_{ix}, CS)$ between each possible global diagnostic hypothesis h_x and the corresponding exception e_{ix} to be raised in WS_i. These mappings complement WS_i's diagnostic model, as far as service CS is concerned.

Defining Diagnostic Information Aware Exception Handlers. We now describe how default exception handling can be overridden in order to take diagnostic information into account. If the composite service is designed by specifying the orchestration of service providers in a process language such as WS-BPEL, a different fault handler may be associated to each type of exception (*WS-BPEL fault*). Thus, we propose to modify each orchestrated Web Service WS_i as follows; see Figure 2 as a reference:

1. *Make the fault handlers of the Web Service aware of the diagnostic information.* To this purpose, each fault handler f has to be modified so that it invokes the Local Diagnoser LD_i by means of a synchronous `String getHypothesis(String localFaultCode)` message and waits for the result before executing any further actions.[3] The result is generated after the interaction between Local and Global Diagnosers and it is a String value corresponding to the global exception to be raised, depending on $\mathcal{H}$. The result received by the local fault handler f after having invoked the Local Diagnoser is characterized as follows:
   a) If f is the appropriate handler for the current case (i.e., the global exception coincides with the local one), the result is the value held by *localFaultCode*, which means that f can continue its own regular execution.[4]

[3] There is a chance that the Local Diagnoser fails itself and does not send the response message. In order to handle this case, the fault handlers might be extended by introducing a time out mechanism which enables them to resume execution after a certain amount of time. We leave this aspect to our future work.

[4] If $\mathcal{H}$ includes more than one global diagnostic hypotheses, the default handling behavior can be performed by making Local Diagnosers return the *localFaultCode* values to the fault handlers.

b) If the case has to be treated by means of another fault handler of the same Web Service, the result returned by the Local Diagnoser is set to the code of a different fault event. In that case, f has to throw the new fault and terminate the execution of the current scope. The occurrence of the new fault event in the same Web Service automatically triggers the appropriate fault handler.
c) If the Web Service is not requested to perform any fault handling procedure (i.e., another Web Service has to trigger a handler of its own), the result returned by the Local Diagnoser is a null String, and the active fault handler f has to terminate the execution.

2. *Possibly add new fault handlers.* Most original fault handlers can be employed to manage both local and global exceptions, after having been revised as specified in item 1. However, additional fault handlers might be required to handle new types of exceptions, derived from the global diagnosis.
3. *Offer a WSDL operation, `throwFault(String faultCode)`,* which a Local Diagnoser LD_i may invoke on Web Service WS_i when the execution of the composite service has failed, but no exceptions were raised in WS_i, nonetheless some recovery action implied by the global diagnosis has to be performed. When the Web Service performs the `throwFault(String faultCode)` operation, it throws the *faultCode* fault, which activates the corresponding fault handler.

All the items above, except for the last one, involve local changes to the code of the exception handlers, which may then be performed by any standard workflow or orchestration engine. In contrast, item 3 has to be implemented in different ways, depending on the characteristics of the engine. The problem is that the `throwFault(String faultCode)` might be invoked at any instant of execution of a Web Service, regardless of its business logic. It is therefore necessary that the Web Service catches the invocation of the operation as a high priority event, to be handled by possibly interrupting the regular execution flow.

For experimental purposes, we have developed our methodology for diagnostic information aware exception handling in the JBPM workflow management environment [12] and the following subsection provides some details about the implementation of the `throwFault(String faultCode)` operation. In other environments, e.g., in a BPEL one, the same methodology can be applied by exploiting the features provided by the execution engine (e.g., defining an eventHandler for an `onMessage` event whose activity will throw the related $faultCode$ fault).

Flexible Exception Management in a Workflow Management Environment. An initial prototype of the framework we propose in this paper has been developed on top of jBPM [12], a business process management system based on Petri Net model implemented in Java. jBPM is based on Graph Oriented Programming model, which complements plain imperative programming with a runtime model for executing graphs (workflows). `Actions` are pieces of Java code that implement the business logic and are executed upon events in the process. In our framework the Web Service is composed of a workflow and a client (Workflow Controller) in charge of monitoring the workflow execution. When a fault occurs, the parent hierarchy of the graph node is searched for an appropriate exception handler. When it is found, the actions of the handler are executed.

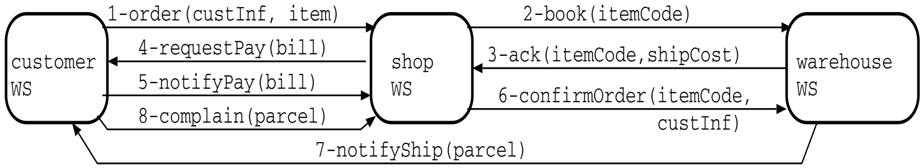


Fig. 3. Portion of a Sample Sales Scenario

Using jBPM makes the implementation of our framework straightforward, in particular referring to item 3 in Section 3.3. When the Web Service receives the `throwFault(String faultCode)` message, the Workflow Controller transfers control to the workflow by generating a special `ExternalFaultEvent`. This event causes the execution of an `Action` that can perform a repair or throw another exception, which activates the corresponding exception handler. In the jBPM case the `Action` can access the current state of the workflow execution by accessing the tree of the active tokens and the associated current graph nodes.

3.4 Example

We sketch the interaction between Diagnoser services and orchestrated Web Services in a sample scenario. Figure 3 shows a portion of the interaction diagram of an e-commerce service based on the orchestration of three Web Services: a *customer WS* enables the user to browse a product catalog and purchase goods. A *shop WS* manages orders, bills and invoicing. A *warehouse WS* manages the stock and delivers the goods to the customer.

In this scenario, the customer places an order for a product by specifying the name of the good (msg 1); the shop reserves the requested item (msg 2) from the warehouse and receives an acknowledgement message (msg 3) where the shipping cost of the parcel is specified; we assume that the shipping cost may vary depending on the address of the customer (included in the *custInf* parameter) and the size of the good that has been purchased.

Given the shipping cost, the shop sends the bill to the *customer WS* (msg 4). The customer pays the bill and notifies the shop (msg 5). At that point, the shop confirms the order (msg 6) and the warehouse sends the package to the customer and notifies the *customer WS* accordingly (msg 7).

If the customer receives a parcel including a good different from the ordered one, she can complain about the delivery problem via the *customer WS* user interface and her complaint makes the Web Service send a `complain(parcel)` message to the *shop WS* (msg 8). The occurrence of this message denotes that there was a delivery problem; therefore, we assume that it is interpreted as an exceptional case by the *shop WS*, i.e., a case in which the execution of the corresponding WSDL operation throws an exception within the Web Service execution instance.

Suppose that that the delivery of a wrong parcel may be due to the following alternative causes:

- h_1: the *shop WS* provided a wrong item code for the requested product.
- h_2: the wrong parcel is picked within the warehouse.

When an exception occurs, the repair strategy to be adopted within the overall service may differ depending on which is the faulty Web Service, i.e., on the global diagnostic hypothesis ($\mathcal{H} = \{h_1\}$ or $\mathcal{H} = \{h_2\}$) and on the corresponding global exceptions to be raised in the Web Services by their own Local Diagnosers, according to the service specific mappings held by the Local Diagnosers. In our example, we assume the following mappings:

- The Local Diagnoser of the *shop WS* maps h_1 to the $wrongItemFault$ exception and h_2 to a null exception: *map(*h_1*,* $wrongItemFault$*)*, *map(*h_2*,* $null$*)*.
- The Local Diagnoser of the *warehouse WS* maps h_1 to a null exception and h_2 to the $wrongParcelFault$ exception: *map(*h_1*,* $null$*)*, *map(*h_2*,* $wrongParcelFault$*)*.

Specifically we assume that the possible failures are handled as follows:

(a) If the *shop WS* provided a wrong item code for the requested product ($\mathcal{H} = \{h_1\}$, $wrongItemFault$ exception), the order to the warehouse and possibly the shipping cost are incorrect. Therefore, the *shop WS* should perform recovery actions that include internal activities and invocations of the other Web Services.

    In detail, the *shop WS* should execute an exception handler which prescribes to: correct the item code, reserve the correct item from the warehouse and recompute the bill; if the new bill differs from the previous one, send it to the customer asking for the difference (or refund her for the extra money). The last steps of the handler include requesting the warehouse to take the wrong parcel from the customer, deliver the new parcel and refund the warehouse for the extra delivery costs.

(b) If the *shop WS* reserved the correct item, but the wrong parcel was picked for the delivery within the warehouse ($\mathcal{H} = \{h_2\}$, $wrongParcelFault$ exception), the first part of the composite service was correctly executed and the bill was correct as well. The problem should be repaired by the *warehouse WS*, which should perform an exception handler prescribing to take the wrong parcel from the customer, deliver the new one, and cover the extra delivery costs.

In our framework, the occurrence of a `complain(parcel)` message is handled as follows (see Figures 4 and 5 for a description of the diagnosis aware fault handlers):[5]

1. Upon receiving the `complain(parcel)` message, the *shop WS* throws an internal *wrongItemFault* event to be caught by the *wrongItemFaultHandler* exception handler.

   The *wrongItemFaultHandler* invokes the Local Diagnoser of the *shop WS* (message `getHypothesis(wrongItemFault)`) to receive the global exception.
2. The Local Diagnoser of the *shop WS* retrieves the global diagnostic hypothesis $\mathcal{H}$ by interacting with the Global Diagnoser.

[5] The fault handlers are described in a java-like syntax, as the description in the process language would be too complex.

```
wrongItemFaultHanldler {

// get exception from local diagnoser
  String globException = sendReceive(shopWS, locDiagnoser,
                            getHypothesis("wrongItemFault"));
  if (globException == null)
    terminate;            // no recovery action needed

  else if (globException != "wrongItemFault")
    throw globException;  // recovery performed by other handler

  else {             // execute original exception handler code
           // order the correct item
    String newItemCode = correctCode(item);
    send(shopWS, warehouseWS, book(newItemCode));
    double shipCost = receive(shopWS,warehouseWS,ack(shipCost));
    double newBill = computeBill(newItemCode);
           // refund or additional payment
    if (newBill>bill) {
      double extraMoney = newBill - bill;
      send(shopWS, customerWS, requestPay(extraMoney));
      receive(shopWS, customerWS, notifyPay(extraMoney));
      }
    else
      send(shopWS, customerWS, refund(bill-newBill));
    send(shopWS, warehouseWS,confirmOrder(newItemCode,custInf));
    send(shopWS, warehouseWS, takeBack(itemCode, custInf));
    refund(warehouseWS, shipCost);
    }
} // end wrongItemFaultHandler
```

Fig. 4. Pseudocode of *wrongItemFaultHandler* (*shop WS*), Modified to be Sensitive to Diagnosis Information

- If $\mathcal{H} = \{h_1\}$, the Local Diagnoser of the *shop WS* returns the *wrongItem-Fault* value as the result of the `getHypothesis` message. As *wrongItem-FaultHandler* is the appropriate fault handler to be executed, the Web Service continues its execution, which involves sending another order to the *warehouse WS* and covering the extra costs. No fault events are handled in the *warehouse WS*.
- Otherwise, if $\mathcal{H} = \{h_2\}$, the Local Diagnoser of the *shop WS* returns a null value and the *wrongItemFaultHandler* terminates the execution. Moreover, the Local Diagnoser of the *warehouse WS* invokes the `throwFault (wrongParcelFault)` message on the *warehouse WS*, which throws the *wrongParcelFault*. The occurrence of such fault event starts the appropriate fault handler (take wrong parcel from customer, deliver the correct one, notify the *shop WS* about the shipping and cover extra delivery costs).

```
wrongParcelFaultHanldler {

// get exception from local diagnoser
  String globException = sendReceive(warehouseWS, locDiagnoser,
                           getHypothesis("wrongParcelFault"));
  if (globException == null)
    terminate;          // no recovery action needed



  else if (globException != "wrongParcelFault")
    throw globException;// recovery performed by other handler


  else {            // execute original exception handler code
   // re-execute the operations for the correct parcel
    String newParcel = correctParcel(itemCode);
    double shipCost = computeShipCost(item, custInf);
    takeBack(wrongParcel, custInf);
    deliver(parcel, custInf);
    send(warehouseWS, shopWS, notifyShip(newParcel));
    coverShipCost(shipCost);
    }
} // end wrongParcelFaultHandler
```

Fig. 5. Pseudocode of *wrongParcelFaultHandler* (*warehouse WS*), Modified to be Sensitive to Diagnosis Information

4 Related Work

Various proposals for the management of transactional behavior in centralized Web Service orchestration are being proposed in order to support the development of reliable composite Web Services; e.g., see WS-Transaction [6] and OASIS BTP [15]. In decentralized orchestration there is the additional problem that the entire state of the original composite Web Service is distributed across different nodes. To address this issue, Chafle and colleagues propose a framework where Local Monitoring Agents check the state of the orchestrated Web Services and interact with a Status Monitor that maintains a view on the progress of the composite service [4]. Although there is a direct correspondence with our Local and Global Diagnoser services, the authors apply traditional error detection and are therefore subject to the limitations we discussed. Moreover, the Status Monitor holds complete information about fault and compensation handlers of the orchestrated services; thus, it is only suitable for closed environments where the orchestrated services are allowed to expose complete information about themselves.

Biswas and Vidyasankar propose a finer-grained approach to fault management in [2], where *spheres of visibility, control and compensation* are introduced to establish different levels of visibility among nested Web Services and composite ones in hierarchical

Web Service composition. Although the described approach sheds light in how the most convenient recovery strategies can be selected, the paper does not clarify how the exception handlers to be executed are coordinated at the various levels of the hierarchy.

5 Discussion

We have described a framework enhancing the failure management capabilities in Web Service orchestration by employing diagnostic reasoning techniques and diagnosis aware exception handlers. Our framework is based on the introduction of a set of Local Diagnosers aimed at explaining incorrect behavior from the local viewpoint of the orchestrated Web Services and on the presence of a Global Diagnoser which, given the local diagnostic hypotheses, generates one or more global hypotheses explaining the occurred exceptions from the global point of view.

The introduction of Local and Global Diagnosers enhances the exception management capabilities of the composite service by steering the execution of exception handlers within the orchestrated Web Services on the basis of a global perspective on the occurred problem. However, it introduces at least two main kinds of overhead, concerning the composite service set up and execution time, respectively:

- At set up time, the administrator of the composite service and those of the orchestrated Web Services have to do some configuration work in order to define service specific settings and possibly to define service specific exception handlers. This aspect obviously restricts the applicability of our framework to Enterprise Application Integration, where explicit agreements between the administrator of the composite service and those of the orchestrated service providers may be defined. It should be however noticed that EAI currently represents the most important application for Web Services, if compared with Web Service invocation in open environments.
- At run time, the interaction between diagnosers, and the time needed to find a global diagnostic hypothesis, may delay the execution of the composite service. However, the diagnostic process is only activated when one or more exceptions occur (thus, in situations where the service execution is already challenged). Moreover, the interaction between the diagnosers and the orchestrated Web Services is limited to the retrieval of the log files and the final notification of the global exception to be handled; therefore, the individual Web Services are not affected by the possibly complex interaction concerning diagnosis.

Our future work concerns two main aspects: first of all, as our approach currently supports the recovery from occurred exceptions, we want to extend it in order to support the early detection of failures. To this purpose, we are analyzing the possibility of monitoring the message flow of a composite service (i.e., the progress in the execution of a conversation graph describing the message exchanges which may occur among the orchestrated Web Services) and to trigger diagnostic reasoning as soon as a deviation from the expected behavior is detected.

Second, our current contribution is related to a rather specific aspect in the general topic of fault tolerant computing. However, our work is part of the WS-DIAMOND European Project which also deals with redundancy and Web Service replacement aspects, to be taken into account in order to recover from faults in complex systems. For more information about such project, see [21].

References

1. L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. Theseider Dupré. Enhancing Web Services with diagnostic capabilities. In *Proc. of European Conference on Web Services (ECOWS-05)*, pages 182–191, Växjö, Sweden, 2005.
2. D. Biswas and K. Vidyasankar. Spheres of visibility. In *Proc. of European Conference on Web Services (ECOWS-05)*, pages 2–13, Växjö, Sweden, 2005.
3. BPMI Business Process Management Initiative. Business Process Management Language. http://www.bpmi.org, 2005.
4. G. Chafle, S. Chandra, V. Mann, and M.G. Nanda. Decentralized orchestration of composite Web Services. In *Proc. of 13th Int. World Wide Web Conference (WWW'2004)*, pages 134–143, New York, 2004.
5. L. Console and O. Dressler. Model-based diagnosis in the real world: lessons learned and challenges remaining. In *Proc. 16th IJCAI*, pages 1393–1400, 1999.
6. W. Cox, F. Cabrera, G. Copeland, T. Freund, J. Klein, T. Storey, and S. Thatte. Web Services Transaction (WS-Transaction). http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html, 2005.
7. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.
8. R. Dechter. *Constraint Processing*. Elsevier, 2003.
9. J. Eder and W. Liebhart. The workflow activity model WAMO. In *Proc. 3rd Int. Conf. on Cooperative Information Systems*, Vienna, 1995.
10. G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.
11. C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.
12. J. Koenig. JBoss jBPM white paper. http://www.jboss.com/pdf/jbpm_whitepaper.pdf, 2004.
13. H. Mourao and P. Antunes. Exception handling through a workflow. In R. Robert Meersman and Z. Tari, editors, *On the move to meaningful internet systems 2004*, pages 37–54. Springer Verlag, Heidelberg, 2004.
14. OASIS. OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel, 2005.
15. OASIS TC. OASIS Business Transaction Protocol. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction, 2005.
16. M.P. Papazoglou and D. Georgakopoulos, editors. *Service-Oriented Computing*, volume 46. Communications of the ACM, 2003.
17. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.
18. S.W. Sadiq. On capturing exceptions in workflow process models. In *Int. Conf. on Business Information Systems*, Poznam, Poland, 2000.

19. W3C. Web Services Definition Language. http://www.w3.org/TR/wsdl, 2002.
20. Workflow Management Coalition. XML process definition language (XPDL). http://www.wfmc.org/standards/XPDL.htm, 2005.
21. WS-DIAMOND. WS-DIAMOND Web Services DIagnosability MOnitoring and Diagnosis. http://wsdiamond.di.unito.it/, 2005.

Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems

Marco Autili, Michele Flammini, Paola Inverardi,
Alfredo Navarra, and Massimo Tivoli

Computer Science Department, University of L'Aquila
Via Vetoio I-67100 L'Aquila, Italy
{marco.autili,flammini,inverard,navarra,tivoli}@di.univaq.it

Abstract. Building a distributed system from third-party components introduces a set of problems, mainly related to compatibility and communication. Our existing approach to solve such problems is to build a *centralized adaptor* which restricts the system's behavior to exhibit only *deadlock-free* and *desired interactions*. However, in a distributed environment such an approach is not always suitable. In this paper we show how to automatically generate a *distributed adaptor* for a set of black-box components. First, by taking into account a specification of the interaction behavior of each component, we synthesize a behavioral model of a centralized *glue adaptor*. Second, from the synthesized adaptor model and a specification of the desired behavior, we generate a set of adaptors local to the components. They cooperatively behave as the centralized adaptor restricted with respect to the specified desired interactions.

1 Introduction

Nowadays, a growing number of software systems are built as composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. *Component Based Software Engineering* (CBSE) is a reuse-based approach which addresses the development of such systems. One of the main goals of CBSE is to compose and adapt third-party components to make up a system [1]. Building a distributed system from reusable or COTS components introduces a set of problems. Often, components may have incompatible or undesired interactions. A widely used technique to deal with these problems is to use adaptors and interpose them between the components forming the system that is being assembled.

One existing approach (implemented in the *SYNTHESIS* tool [2]) is to build a *centralized adaptor* which restricts the system's behavior to exhibit only a set of *deadlock-free* or *desired interactions*. However in a distributed environment it is not always possible or convenient to insert a centralized adaptor. For example, existing legacy distributed systems might not allow the addition of a new component (i.e., the adaptor) which coordinates the information flow in a centralized way. Moreover, the coordination of an increasing number of components can cause loss of information andbottlenecks, with corresponding

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 17–32, 2006.
© Springer-Verlag Berlin Heidelberg 2006

increase of the response time of the centralized adaptor. In contrast, building a distributed adaptor might increase the applicability of the approach in real-scale contexts.

In this paper we describe an approach for automatically generating a *distributed adaptor* for a set of black-box components. Given (i) a specification of the *interaction behavior* of each component with its environment and (ii) a specification of the *desired behavior* that the system to be composed must exhibit, it generates *component local adaptors* (one for each component). These local adaptors suitably communicate in order to avoid possible deadlocks and enforce the specified desired interactions. They constitute the distributed adaptor for the given set of black-box components.

Starting from the specification of the components' interaction behavior, our approach synthesizes a behavioral model (i.e., a Labeled Transition System (LTS)) of a centralized *glue adaptor*. This is done by performing a part of the synthesis algorithm described in [2] (and references therein). At this stage, the adaptor is built only for modeling all the possible component interactions. It acts as a simple router and each request/notification it receives is strictly delegated to the right component. By taking into account the specification of the desired behavior that the composed system must exhibit, our approach explores the centralized glue adaptor model in order to find those states leading to deadlocks or to interactions different from the desired ones. This process is used to automatically derive the set of local adaptors that constitute the *correct*[1] and *distributed* version of the centralized adaptor. It is worth mentioning that the construction of the centralized glue adaptor is required to deal with deadlock in a fully-automatic way. Otherwise we should make the stronger assumption that the specification of the desired behaviors itself ensures also deadlock-freeness. The approach presented in this paper has various advantages with respect to the one described in [2] concerning the synthesis of centralized adaptors. The most relevant ones are: (a) no centralized point of information flow exists; (b) the degree of parallelism of the system without the adaptor is now maintained. Conversely, the approach in [2] does not permit parallelism due to the adaptor centralization; (c) all the domain-specific deployment constraints imposed on the adaptor can be removed. In [2] we applied the synthesis of centralized adaptors to COM/DCOM applications. In this domain, the centralized adaptor and the server components had to be deployed on the same machine. On the contrary, the approach described in this paper allows one to deploy each component (together with its local adaptor) on different machines.

The remainder of the paper is structured as follows: Section 2 describes the application domain. In Section 3 the synthesis of decentralized adaptors is firstly described and then formalized by also proving its correctness. Section 4 describes our approach at work by means of a running example. Section 5 discusses related work, and finally, Section 6 concludes and discusses future work.

[1] With respect to deadlock-freeness and the specified desired behavior.

2 The Context

In our context, a distributed system is a network of interacting black-box components $\{C_1, \ldots, C_n\}$ that can be simultaneously executed. Components communicate each other by message passing according to synchronous communication protocols. This is not a limitation because it is well known that with the introduction of a buffer component we can simulate an asynchronous system by a synchronous one [3]. We distinguish between *standard communication* and *additional communication*. The first denotes the messages that components can exchange. The latter denotes the messages that the local adaptors exchange in order to coordinate each other. Due to synchronous communication, a deadlocking interaction might occur whenever components contend the same request. Furthermore, by letting components interact in an uncontrolled way, they might perform undesired interactions. To overcome this problem we promote the use of additional components (called local adaptors). Each local adaptor is a wrapper that performs the component' standard communication and mediates it by exchanging synchronizing information (i.e., additional communication), when needed. Synchronizing information allow components to harmonize their interaction on requests and notifications. Each component is directly connected to its local adaptor through a synchronous channel; each local adaptor is connected to the other ones, through asynchronous channels, in a peer-to-peer fashion (see for instance the right-hand side of Figure 1). For the sake of clarity, we assume the components are single-threaded and hence all the requests and notifications can be totally ordered to constitute a set of sequences (i.e., a set of traces). Note that this is not a restriction since a multi-threaded component can always be modeled as a set of single-threaded (sub)components simultaneously executed. Interaction among components is modeled as a set of *linearizations* obtained by means of interleaving [4]. It is worth noting that, in such a concurrent and distributed context, we cannot assume either a single physical clock or a set of perfectly synchronized ones in order to determine whether an event a occurs before an event b or vice versa. We then need to define a relationship among the system events by abstracting both on the absolute speed of each processor and on the absolute time. In this way we ignore any absolute time scale and we use the well known *happened-before relation* and *time-stamps method* (see [5] for a detailed discussion).

3 Method Description and Formalization

In this section we first describe our method to deal with the adaptation problem in a component-based setting. Then, we gradually formalize it by means of a detailed discussion and pseudo-code description of the setup and local adaptors interaction procedures. This section also proves the correctness of our approach and concludes with a brief discussion about the additional communication overhead.

3.1 Method Description

Our method (see Figure 1) assumes as input: (i) a behavioral specification of the system formed by interacting components. It is given as a set $\{AC_1, \ldots, AC_n\}$ of LTS (one for each component C_i). The behavior of the system is modeled by composing in parallel all the LTS and by forcing synchronization on common actions; (ii) the specification of the desired behavior that the system must exhibit. It is given in terms of a LTS, from now on denoted by P_{LTS}.

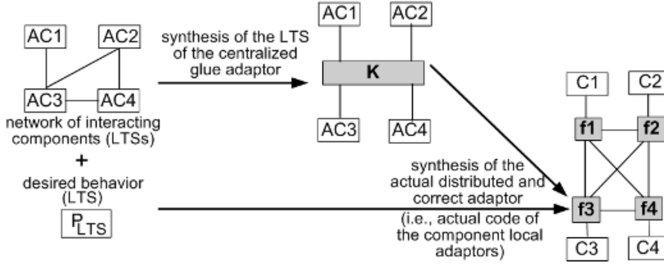


Fig. 1. 2-step method

These two inputs are then processed in two main steps. (1) By taking into account all component LTSs, we automatically derive the LTS K that models the behavior of a centralized glue adaptor. K, at this stage, models all the possible component interactions and it does not apply any adaptation policy. In other words, K performs standard communication simply routing components requests and notifications. In this way, it represents all possible linearizations by using an interleaving semantics. K is derived by performing the *graph unification* algorithm described in [2]. It is worth mentioning that each state of K (i.e., a global state) is a tuple $< S_1, \ldots, S_n >$ where each S_i is a state of AC_i (see for instance Figure 2). Hereafter, when the current state of a component appears in a tuple representing a global state we simply say that the component is in that global state.[2] This first step is taken from the existing approach [2] for the synthesis of centralized adaptors. As already mentioned in Section 1, whenever P_{LTS} ensures itself deadlock-freeness, such a step is not required. For the sake of presentation we will always assume that K exists. The novel contribution of this paper is represented by the second step. (2) If K has been generated, our method explores it looking for those states representing the *last chance* before entering into an execution path that leads to deadlock. The restriction with respect to the specified desired behavior is realized by visiting P_{LTS}. The aim is to split and distribute P_{LTS} in such a way that each local adaptor knows which actions the wrapped component is allowed to execute. The sets of last chance states and *allowed actions* are stored and, subsequently, used by the local adaptors as basis for correctly exchanging synchronizing information. In other words, the local

[2] In general, a component might be in more than one global state.

adaptors interact with each other (by means of both standard and additional communication) to perform the correct behavior of K with respect to deadlock-freeness and P_{LTS}. Decentralizing K, the local adaptors preserve parallelism of the components forming the system. In the following subsection we formalize the second step of our method by also providing its correctness.

3.2 Second Step Formalization

As described before, the second step gets in input: (i) the set $\{AC_1, \ldots, AC_n\}$, (ii) K and (iii) P_{LTS}. In order to detect deadlocks, our approach explores K and looks for sinks. A deadlock state (see Figure 2) is in fact a sink of K. We call *Forbidden States* (FS) the set of deadlock states[3] and all the ones within *forbidden paths* necessarily leading to them. A forbidden path in K is a path that starts at a node which has no transitions that can avoid a forbidden state and thus necessarily ends in a sink (see for instance Figure 2). The states in FS can be avoided by identifying a specific subset of K's states that are critical with respect to FS (see for instance S in Figure 2). In this way we can avoid to store the whole graph at runtime as we just need to store the critical states. More precisely, in order to avoid a state in FS, we are only interested in those nodes representing the last chance before entering into a forbidden state.

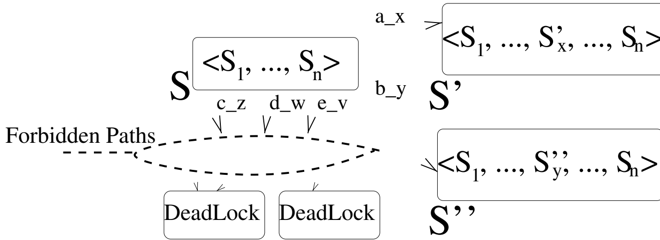


Fig. 2. A last chance node S of K

The last chance nodes have some outgoing edges leading to a forbidden state, the *dead* edges, and other ones, the *safe* edges (see for instance the edges labeled with *a_x* and *b_y* in Figure 2). According to the labels of the dead edges we store in the local adaptors associated to the corresponding components the last chance node, and the *critical action* that each component should not perform in order to avoid a state in FS (in Figure 2, the action c is critical for the component z). From the implementation point of view, each local adaptor F_{C_i} uses a table $F^{LC}_{C_i}$ (*Last Chance table* of F_{C_i}) of pairs <*last chance state of K, critical action of AC_i*>. Thus, once all the graph has been visited, each local adaptor knows the critical actions of the corresponding component. Before a component can perform a critical action its local adaptor has to ask permissions to the other components

[3] Abusing notation, sometimes we refer to the states as nodes.

(see procedure *KVisit*). The following procedure computes and distributes the last chance node tables among the local adaptors. Given in input the centralized glue adaptor K of n components, the procedure makes use of the following variables: $F_{C_i}^{LC}$ is the table of last chance nodes associated to the component C_i; $Flag_Forbidden_S$ is a flag to check whether the current node S eventually leads to deadlock or not; $Dead_Son_S$ counts the number of sons of the current node S that eventually lead to forbidden states of K; $Safe_Son_S$ counts the number of sons of the current node S that may lead to allowed states of K.

procedure KVisit(state of K: S;)
1: **for** each $i := 1$ **to** n **do**
2: $F_{C_i}^{LC} := \emptyset$;
3: **end for**
4: $Flag_Forbidden_S :=$ False;
5: $Dead_Son_S := 0$;
6: $Safe_Son_S := 0$;
7: mark S as Visited;
8: **for** each son S' of S **do**
9: **if** the edge (S, S') is not visited **then**
10: mark the edge (S, S') as Visited;
11: **if** S' is not visited **then**
12: $KVisit(S')$;
13: **end if**
14: **if** $Flag_Forbidden_{S'}$ **then**
15: $Dead_Son_S$++;
16: **else**
17: $Safe_Son_S$++;
18: **end if**
19: **end if**
20: **end for**
21: **if** $Safe_Son_S == 0$ **then**
22: $Flag_Forbidden_S :=$ True;
23: **end if**
24: **if** $Safe_Son_S > 0$ && $Dead_Son_S > 0$ **then**
25: for every dead edge, let α_x be the associated action, $F_{C_x}^{LC} = F_{C_x}^{LC} \bigcup < S, \alpha >$;
26: **end if**

Before starting a critical action (that might lead to a state in FS), a local adaptor has to verify (by performing additional communication) if the global state represents a last chance state with respect to that action. Since at runtime we do not store K, this verification is made by enquiring the other local adaptors about the states of the corresponding components, hence deriving the appropriate consequences. If a component is not in the enquired last chance state, its associated local adaptor immediately replies ensuring that the component will not reach such a state. In some way it is *self-blocked* with respect to the enquired state. If the component is already in the enquired last chance state or it is interested in reaching it, its local adaptor defers the answer and hence, it attempts

to block the enquiring local adaptor. The only case in which an enquiring local adaptor has to ask the permission to all the others is when the global state is exactly a last chance one. Once the enquiring local adaptor receives an answer it allows its corresponding component to proceed with its standard communication by delegating the critical action. After that, it sends a message to unblock all the other local adaptors previously enquired (additional communication). The unblock message is needed because once a local adaptor allows an enquiring one to perform a critical action, it ensures also that it will not reach the last chance state before receiving an unblock message with respect to such a state (see code lines 7 and 14 of Procedure *Ack* below). In practice it is self-blocked just with respect to the enquired state.

Concerning P_{LTS}, we visit and distribute it among the local adaptors (see Procedure *PVisit* reported below). Such a distribution is made by means of another table $F^{UA}_{C_i}$ for each local adaptor F_{C_i} (called *Updating and Allowed actions table* of F_{C_i}) of tuples <*state of* P_{LTS}*, allowed action of* AC_i*, state of* P_{LTS}*, set of components, set of components* >. The first three elements of each tuple represent an edge of P_{LTS}. The fourth (fifth) is the set of *active components*, i.e., the ones that can perform some action "matching" with a transition outgoing from the state of P_{LTS} specified by the first (third) element of each tuple. By means of *PVisit* each local adaptor knows its allowed actions that can change the state of P_{LTS}. Moreover, a local adaptor knows also which are the active components that can move and which must be blocked according to the current state of P_{LTS}. Let us assume that a component C_i is going to perform an action contained in the table $F^{UA}_{C_i}$. If it can proceed according to the current state of P_{LTS}, then all the other active components are blocked by sending a blocking message to the corresponding local adaptors. Once C_i has performed the action, all the components that can move in the new state of P_{LTS} are unblocked. Note that if an action of an active component does not change the state of P_{LTS}, it can be performed without exchanging messages among the system components, hence maintaining pure parallelism (this is realized by Procedure *Ask*, code line 34). The setup of the Last Chance and the Updating and Allowed action tables is realized by means of two procedures *KVisit* (see above) and *PVisit* (see below). They are depth-first visits of K and P_{LTS}, respectively. These procedures are executed at design-time in order to setup the corresponding tables. After their execution, K and P_{LTS} can be discarded. Procedures *Ask* and *Ack*, instead, implement the local adaptors interactions at runtime. Referring to the table of updating allowed actions, let $Lookahead(state\ of\ P_{LTS} : p)$ be a procedure that given a state p of the P_{LTS} automaton, returns the set of components that are allowed to perform an action in the state p. The following procedure distributes P_{LTS} among the local adaptors. Given in input P_{LTS} referred to n components, the procedure makes use of the following variables: $Active_Components$ is the set of components that are allowed to make a move in the current state p of P_{LTS}; $Next_Components$ is the set of components that must be allowed to move once the current state of P_{LTS} has changed; $F^{UA}_{C_i}$ is the table of updating and allowed actions of the component C_i.

procedure PVisit(state of P_{LTS}: p;)
1: **for** each $i := 1$ **to** n **do**
2: $\quad F^{UA}_{C_i} := \emptyset$;
3: **end for**
4: $Active_Components := Lookahead(p)$;
5: $Next_Components := \emptyset$;
6: mark p as Visited;
7: **for** each son p' of p **do**
8: $\quad$ **if** the edge (p, p') is not visited **then**
9: $\quad\quad$ mark the edge (p, p') as Visited;
10: $\quad\quad Next_Components := Lookahead(p')$;
11: $\quad\quad$ **for** each $C_i \in Active_Components$ allowed to perform an action α by the label of the edge (p, p') **do**
12: $\quad\quad\quad F^{UA}_{C_i} := F^{UA}_{C_i} \bigcup < p,\ \alpha,\ p',\ Active_Components,\ Next_Components >$;
13: $\quad\quad\quad$ **if** p' is not visited **then**
14: $\quad\quad\quad\quad PVisit(p')$;
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end for**
17: $\quad$ **end if**
18: **end for**

Once this procedure is performed, each local adaptor knows in which state of P_{LTS} it can allow the corresponding component to perform a specific action. Moreover, once the component performs such an action, it knows also which are the components that must be blocked and which ones must be unblocked in order to respect the behavior specified by P_{LTS}.

In the following we describe how a local adaptor uses the tables to correctly interact with each other (i) in a deadlock-freeness and (ii) as specified by P_{LTS}. On the exchanged messages, when needed, we use the standard time-stamps method in order to avoid problems of synchronization. In this way an ordering among dependent messages is established and starvation problems are also addressed. Note that also a priority ordering among components is *a priori* fixed. This solves ordering problems concerning messages with the same time-stamps. A local adaptor, whose current time-stamp is TS, whenever receives a message with associated a time-stamp ts, it makes use of the following simple procedure in order to update TS.

procedure UpTS(timestamp: ts;)
1: **if** $TS < ts$ **then**
2: $\quad TS := ts + 1$;
3: **end if**

Let C_x be an active component that is going to perform action α (i.e., in AC_x there is a state transition labeled with α and α does not collide with respect to P_{LTS}). The associated local adaptor F_{C_x} checks if α is either (i) a critical action (i.e., α appears in $F^{LC}_{C_x}$) or (ii) an updating and allowed action (i.e., α appears in $F^{UA}_{C_x}$). If it is not, F_{C_x} delegates α with associated the current time-stamp TS

increased by 1 to synchronize itself with the rest of the system. If (i) then F_{C_x} enters in the following procedure in order to ask for the permission to delegate α. This is done by checking if for any pair $< S, \alpha > \in F^{LC}_{C_x}$ there is at least one local adaptor F_{C_y} whose corresponding component C_y is not in S. If (ii) then F_{C_x} enters in the following procedure in order to try to block all the active components and after having performed α, it unblocks the components that can be activated with respect to the new state reached over P_{LTS}.

procedure Ask(action: α;)

1: Let C_x be the current component that would perform action α and let S_{C_x} be its current state and p be the current state of P_{LTS};
Let $< t_i >^{UA}_x$ be the i-th tuple contained in the table $F^{UA}_{C_x}$ and $< t_i >^{UA}_x [j]$ be its j-th element;
2: $flag_forbidden := 0$;
3: **if** $\exists\, i \mid < t_i >^{UA}_x [1] == p \;\&\&\; < t_i >^{UA}_x [2] == \alpha$ **then**
4: **if** α appears in some pair of $F^{LC}_{C_x}$ **then**
5: **for** every entry $< S, \alpha > \in F^{LC}_{C_x}$ **do**
6: $i := 1$;
7: $TS++$;
8: **while** no "ACK, α,ts" received && $i \leq n$ **do**
9: Let $S \equiv < S_{C_1}, \ldots, S_{C_n} >$; F_{C_x} asks to local adaptor F_{C_i} if it is in or approaching the state S_{C_i} with associated TS;
10: $i++$;
11: **end while**
12: **if** $i > n$ **then**
13: WAIT for an "ACK, α, ts" message
14: **end if**
15: $UpTS(ts)$;
16: **if** $i > n$ **then**
17: $i := n$;
18: **end if**
19: **for** $j := 1$ **to** i **do**
20: send "UNBLOCK, α, TS" to F_{C_i};
21: **end for**
22: **end for**
23: **end if**
24: $TS++$;
25: **if** $< t_i >^{UA}_x [1] != < t_i >^{UA}_x [3]$ **then**
26: **for** each component $C_j \in < t_i >^{UA}_x [4]$ **do**
27: send "BLOCK, TS" to F_{C_j};
28: **end for**
29: perform action α;
30: **for** each component $C_j \in < t_i >^{UA}_x [5]$ **do**
31: send "UNBLOCK, $< t_i >^{UA}_x [3]$, TS" to F_{C_j};
32: **end for**
33: **else**
34: perform action α;
35: **end if**
36: **end if**

Note that, by code line 13, the present local adaptor is self-blocked till some local adaptor gives the permission to proceed, i.e. an "ACK". The *"UNBLOCK"* messages of code line 20 say to all the local adaptors that were blocked with respect to the enquired forbidden states, to proceed. The *"UNBLOCK"* messages of code line 31 are instead to unblock components due to the change of state of P_{LTS} occurred after having performed action α. On the other hand, when a local adaptor receives a request for a permission, after having given such a permission, it is implicitly self-blocked in relation to the set of states it was enquired for. The following procedure describes the *"ACK"* messages exchanging method.

procedure Ack(last chance state: S; action: α; timestamp: $ts1$;)

1: Let F_{C_y} be the local adaptor (performing this Ack) that was enquired with respect to the state S and the action α that C_x would perform; let S'_{C_y} be the current state of F_{C_y} and S''_{C_y} be the state that F_{C_y} would reach with the next hop.
2: $UpTS(ts1)$;
3: **if** $S'_{C_y} \neq S$ && F_{C_y} didn't ask the permission to get in S **then**
4:     send "ACK, α, TS" to F_{C_x} that allows C_x to perform the action α;
5:     **if** $S''_{C_y} == S$ **then**
6:         WAIT for "UNBLOCK, α, $ts2$" from F_{C_x};
7:     **end if**
8:     S''_{C_y} := next desired state of F_{C_y};
9: **else**
10:     once $S'_{C_y} \neq S$ send "ACK, α, TS" to F_{C_x} that allows C_x to perform the action α;
11:     **if** no "UNBLOCK, α, $ts2$" from F_{C_x} has been received **then**
12:         WAIT for it;
13:     **end if**
14:     $UpTS(ts2)$;
15: **end if**

The *"WAIT"* instructions of code lines 6 and 12 block the current local adaptor in order to not allow the corresponding component to enter in a forbidden state. Note that, while the *"UNBLOCK"* message has a one-to-one correspondence, that is, for each message there is a receiver waiting for it, the *"ACK"* message can be sometimes useless. In fact a local adaptor needs just one *"ACK"* message in order to allow the corresponding component to proceed with the enquired critical action. All the other possible *"ACK"* messages are ignored.

3.3 Correctness

We now provide the correctness of our method by proving that assuming K and P_{LTS}, the method synthesizes local adaptors that (i) allow the composed system to be free from deadlocks and (ii) allow P_{LTS} to be exhibited.

We prove (i) by focussing on the last chance nodes. Note that, since the synthesis of K is correct as proved in [2], we can assume that the last chance nodes are correctly discovered by means of the procedure $KVisit$ that performs a standard depth-first visit. Thus, our proof can be reduced to show that the

local adaptors disallow the system to reach a forbidden path. Note that, by construction, such a path can be undertaken only through a last chance node by performing an action that labels one of its outgoing dead edges. Let us assume by contradiction that the component z can perform the critical action c from the last chance state S, and that S has an outgoing dead edge labeled by c_z (see for instance Figure 2). Since, as already noticed, the last chance nodes are correctly discovered, when procedure *KVisit* is visiting S, it stores in F_z^{LC} the tuple $< S, c >$. At runtime, whenever the component z would perform action c, F_z checks if c is a critical action by means of code line 4 of its *Ask* procedure. It then starts to ask the permission (at least an "ACK" message) to all the other components by means of the "while" cycle of code line 8 of the same procedure. Each enquired local adaptor F_{C_i}, by the *Ack* procedure, checks if the current state of the corresponding component C_i is in S. If it is, it does not reply to z till it does not change status (code line 10 of the *Ack* procedure). In doing so, until the system state remains S, no local adaptor will reply to F_z. Since F_z is blocked on code line 13 of the *Ask* procedure till no "ACK" message is received, a contradiction follows by observing that action c can be performed by z at code line 29 of the same procedure.

To prove (ii), let us assume by contradiction that the component x performs the action a when this is not allowed by P_{LTS}, that is, the current state S_P of P_{LTS} has no outgoing edge labeled by a_x. First of all, in order for a component to be active, either its local adaptor has received an "UNBLOCK" message from some other local adaptor (by means of code line 31 of the *Ask* procedure) or the system is just started and F_x^{UA} has some entry with $S0$ (the initial state of K) as first element. In both cases each time a component is active, its local adaptor knows exactly which is the current P_{LTS} state. By construction, x can perform action a if there exists an entry in F_x^{UA} whose first element matches with the current state of P_{LTS} and whose second element matches with a (see code line 3 of the *Ask* procedure). The contradiction follows by observing that such an entry was obtained by visiting P_{LTS} hence, by construction, there must exist an outgoing edge whose label matches with a_x from the node labeled by S_P.

4 Running Example

In this section we show our approach at work by means of a running example. This example concerns the semi-automatic assembly of a distributed client-server system made of four components, two servers (denoted by *C1* and *C2*) and two clients (denoted by *C3* and *C4*). The behavioral specification of *C1*, *C2*, *C3* and *C4* (shown in Figure 3 in form of LTSs) has been borrowed from an industrial case study described in [6]. *C1* (resp., *C2*) provides two methods *p* and *FreeP* (resp., *p1* and *FreeP1*). Moreover, *C2* provides also a method *Connect*. By referring to the method described in Figure 1, by taking into account the LTSs of *C1*, *C2*, *C3* and *C4*, we automatically synthesize a model of the centralized glue adaptor K. This is done by using SYNTHESIS and performing the approach

described in [2]. Finally, by taking into account the LTS specification of the desired behavior that the composed system must exhibit, we mechanically distribute the correct behavior of K in a set of local adaptors *f1*, *f2*, *f3* and *f4*.

4.1 Our Approach at Work

Figure 3 shows the LTSs of *C1*, *C2*, *C3* and *C4*. Within the LTS of a component, a message $?m$ ($!m$) denotes a received (sent) request or notification labeled with m. The state with an incoming arrow denotes the initial state. For instance, *C1* (*C2*), from its initial state, receives a request of p (*p1*) followed by a request of *FreeP* (*FreeP1*).

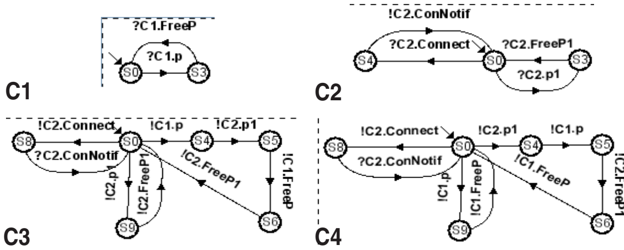


Fig. 3. Components' LTSs

Moreover, *C2* from the initial state can receive a request of *Connect* and, subsequently, replies to it by means of the notification *ConNotif*. The interaction behavior of the clients $C3$ and $C4$ can be easily understood by simply looking at Figure 3.

Figure 4 shows part of the LTS of K. Within K, a message $?m_j$ ($!m_j$) denotes a request or notification labeled with m and received (sent) from (to) Cj. The state $S0$ with an incoming arrow denotes the initial state. K contains filled nodes, which denote deadlocks. Deadlocks might occur, e.g., because of a "race condition" among $C3$ and $C4$. In fact, one client (e.g., *C3*) performs a request of *p1* (see the sequence of transitions from the state $S0$ to $S2$ in K) and waits for performing the request p while the other client (i.e., *C4*) performs p (see the sequence of transitions from the state $S2$ to $S16$ in K) and waits *p1*. In this scenario $C3$ and $C4$ are in the state $S4$ of their LTSs (see Figure 3). Since none of the clients performs the corresponding *FreeP* method before having performed both p and *p1*, a deadlock occurs (see the filled state $S16$ in K). Note that, following the sequence of transitions from the state $S0$ to $S12$ leads to the symmetric scenario. By referring to Figure 4, the states $S15$ and $S29$ are filled since they can only lead to deadlocks. $S2$ and $S12$ are last chance states. The paths from $S2$ to $S16$ and from $S12$ to $S16$ are forbidden paths and $FS = \{S15, S16, S29\}$. Following we show the tables of last chance nodes used by each local adaptor as generated by procedure *KVisit*. For both $S2$ and $S12$ there is just one critical action that leads to a deadlock. This is translated by procedure

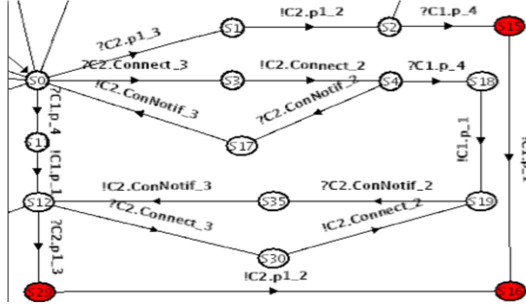


Fig. 4. Part of the LTS of the centralized glue adaptor K. The filled nodes belong to deadlock paths.

$KVisit$ in storing the entries $< S12 \equiv < S3_{C1}, S0_{C2}, S0_{C3}, S9_{C4} >, !C2.p1 >$ in F^{LC}_{C3} and $< S2 \equiv < S0_{C1}, S3_{C2}, S9_{C3}, S0_{C4} >, !C1.p >$ in F^{LC}_{C4}, respectively. In this way, each time the component $C3$ ($C4$) performs the action $!C2.p1$ ($!C1.p$), the corresponding local adaptor has to check that the global status is not $S12$ ($S2$). After performing $KVisit$ to derive the last chance nodes table for each local adaptor, SYNTHESIS performs $PVisit$ to derive the updating and allowed actions table for each local adaptor. This is done by taking into account the LTS specification P_{LTS} (see Figure 5). We recall that these tables are needed to distribute P_{LTS} among the local adaptors.

Fig. 5. The system's desired behavior specified by P_{LTS}

In our context, P_{LTS} describes (at an high-level) a desired behavior for the composed system. Each node is a state of the system. The node with the incoming arrow is the initial state. The syntax and semantics of the transition labels is the same of the LTS of K except for two kinds of action: i) a universal action (i.e., $?true_$) which represents any possible action, and ii) a negative action (e.g., $! - C2.Connect_4$ in Figure 5) which represents any possible action different from the negative action itself. P_{LTS} specifies that it is mandatory for $C3$ to perform a *Connect* before performing *p1* (see the self-transition on the state $S1$ and the transition from $S1$ to $S2$ showed in Figure 5). The self-transition on $S1$ is the logical AND of the actions in the action list delimited by '{' and '}'. The semantics of this self transition is that the current state of P_{LTS} (i.e., $S1$) remains unchanged until an action different from $!C2.Connect_3$, $!C2.FreeP1_3$ and $!C2.p1_3$ is performed. When $C3$ performs $!C2.Connect_3$ the current state of P_{LTS} becomes $S2$. Then, while being in the state $S2$ all the components but

$C4$ simultaneously execute unconstrained (see the negative self-transition on the state $S2$ in Figure 5). Finally, *FreeP1* will be performed by $C3$ to allow another client to perform *p1* (see the transition from $S2$ to $S1$ showed in Figure 5). Following we show the tables of updating and allowed actions used by each local adaptor as generated by the procedure *PVisit*. Denoting by "$*$" any possible value of a specified scope, P_{LTS} is translated by procedure *PVisit* in storing the entries $< *, *, *, *, * >$ in F^{UA}_{C1}, F^{UA}_{C2}; while $< S0, !C2.Connect, S1, *, * >$, $< S1, !C1.p, S1, *, * >$, $< S1, !C2.p1, S1, *, * >$, $< S1, !C1.FreeP, S1, *, * >$, $< S1, !C2.Connect, S1, *, * >$, $< S1, !C2.FreeP1, S0, *, * >$ in F^{UA}_{C3} and $< S0, *, S0, *, * >$, $< S1, !C1.p, S1, *, * >$, $< S1, !C2.p1, S1, *, * >$, $< S1, !C1.FreeP, S1, *, * >$, $< S1, !C2.FreeP1, S1, *, * >$ in F^{UA}_{C4}. Note that, when during the runtime, the state of P_{LTS} changes from $S0$ to $S1$ by means of the action $!C2.Connect$ performed by $C3$, F_{C3} informs F_{C4} of the new state of P_{LTS} by means of the "UNBLOCK" message of code line 31 of its *Ask* procedure. Consequently F_{C4} knows that in such a state $C4$ cannot perform $!C2.Connect$ since the entries $< S1, !C2.Connect, *, *, * >$ are not present in F^{UA}_{C4}. Once the LC and UA tables are filled, the interactions among local adaptors can start by means of procedures *Ask* and *Ack*. In order to better understand such an interaction, let us consider the sequence of messages that according to the glue coordinator of Figure 4 leads the global state from $S0$ to $S15$. Note that a forbidden path starts from $S15$. The first message is sent by F_{C3} to F_{C2} in order to ask the resource $p1$. This is allowed by the entry $< *, *, *, *, * >$ contained in F^{UA}_{C2}. It means, in fact, that $C2$ can perform any action from any global state according to P_{LTS}. When from $S1$, F_{C4} would perform $?C3.p_4$, according to the entry $< S2 \equiv < S0_{C1}, S3_{C2}, S9_{C3}, S0_{C4} >, !C1.p >$ contained in F^{LC}_{C4}, it has to check if the current global state is $S2$ in order to not incur in the forbidden path that starts from $S15$. According to procedure *Ask*, it starts to ask the permission to all the other local adaptors (see code lines 8 of procedure *Ask*). Since the current state is exactly $S2$ it will not receive any answer from the other local adaptors (this is accomplished by code line 3 of procedure *Ack* since if the enquired local adaptor is in the enquired state, the if condition is not satisfied and the ACK message cannot be sent). Such a situation changes as soon as some component changes its status hence unblocking F_{C4} (see code line 12 of procedure *Ack*). Note that since such interaction concern just an action of $C4$, by construction, this allow all the other local adaptors to continue their interaction according to P_{LTS} hence maintaining the eventual parallelism.

5 Related Work

The approach presented in this paper is related to a number of other approaches that have been considered by researchers. For space reasons, we discuss only the ones closest to our approach.

In [7] a game theoretic approach is used for checking whether incompatible component interfaces can be made compatible by inserting a converter between

them. This approach is able to automatically synthesize the converter. Contrarily to what we have presented in this paper, the synthesized converter is a centralized adaptor.

Our research is also related to [8] in the area of protocol adaptor synthesis. The main idea is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow to automatically derive a distributed implementation of the adaptor. Note that our approach can be easily extended to address syntactic incompatibilities between component interfaces. We refer to [2] for details concerning such an extension.

In another work by some of the authors [6], it is showed how to generate a distributed adaptor by exploiting an approach to the definition of distributed Intrusion Detection Systems (IDS). Analogously to the approach described in this paper, the distributed adaptor is derived by *splitting* a pre-synthesized centralized one in a set of local adaptors (each of them local to each component). The work in [6] represents a first attempt for distributing centralized adaptors and it has two main disadvantages with respect to the approach described here: (a) the method requires a more complex (in time and space) process for pre-synthesizing the centralized adaptor. In fact, it does not simply model all the possible component interactions (like our centralized glue adaptor), but it has to model the component' interactions that are deadlock-free and that satisfies the specified desired behavior (P_{LTS}). In that approach, in fact, the glue adaptor is generated and, afterwards, a suitable synchronous product with P_{LTS} is performed. This longer process with respect to the current approach might also lead to a final bigger centralized adaptor. (b) The adopted solution realize distribution but not parallelism. The distributed local adaptors realize, in fact, the strict distribution of the obtained centralized adaptor by means of the pre-synthesizing step. This means that, since the centralized coordinator cannot parallelize its contained traces, the interactions of the local adaptors maintain this behavior.

In [9], the authors show how to monitor safety properties locally specified (to each component). They observe the system behavior simply raising a *warning message* when a violation of the specified property is detected. Our approach goes beyond simply detecting properties by also allowing their enforcement. In [9] the best thing that they can do is to reason about the global state that each component *is aware of*. Note that, such a global state might not be the actual current one and, hence, the property could be considered guaranteed in an *"expired"* state. Furthermore, they cannot automatically detect deadlocks.

6 Conclusion and Future Work

In this paper we have presented an approach to automatically assemble concurrent and distributed component-based systems by synthesizing distributed adaptors. Our method extends our previous work described in [2] that permitted to

automatically synthesize centralized adaptors for component-based systems. The method described in this paper allows us to derive a distributed implementation of the centralized adaptor and, hence, it enhances *scalability*, *fault-tolerance*, *efficiency*, *parallelism* and *deployment*. We successfully validated the approach on a running example. We have also implemented it as an extension of our SYNTHESIS tool [2]. The state explosion phenomenon suffered by the centralized glue adaptor K still remains an open problem. K is required to detect the last chance nodes that are needed to automatically avoid deadlocks. Indeed when the deadlocks can be solved in some other ways (e.g., using timeouts) or P_{LTS} ensures their avoidance, generating K is not needed. Local adaptors may add some overhead in terms of messages exchanged. In practical cases, where usually many parallel computations are allowed, the overhead is negligible since additional communications are much less then standard ones. As future work, whenever K is required, an interesting research direction is to investigate the possibility of directly synthesizing the implementation of the distributed adaptor without producing the model of the centralized one. Further validation by means of a real-scale case study would be interesting.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (2004)
2. M.Tivoli, M.Autili: Synthesis: a tool for synthesizing “correct” and protocol-enhanced adaptors. RSTI L'Objet journal **12** (2006) 77–103
3. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
4. Ben-Ari, M.: Principles of concurrent and distributed programming. Prentice Hall (1990)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
6. P.Inverardi, L.Mostarda, M.Tivoli, M.Autili: Synthesis of correct and distributed adaptors for component-based systems: an automatic approach. In: Proc. of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)- Long Beach, CA, USA. (2005)
7. Passerone, R., de Alfaro, L., Heinzinger, T., Sangiovanni-Vincentelli, A.L.: Convertibility verification and converter synthesis: Two faces of the same coin. In: Proc. of International Conference on Computer Aided Design (ICCAD) - San Jose, CA, USA. (2002)
8. Yellin, D., Strom, R.: Protocol specifications and component adaptors. ACM Trans. on Programming Languages and Systems **19**(2) (1997) 292–333
9. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proc. of International Conference on Software Engineering (ICSE) - Edinburgh - UK. (2004).

Introspective Model-Driven Development

Thomas Büchner and Florian Matthes

Chair of Software Engineering for Business Information Systems
Technische Universität München
Boltzmannstraße 3, 85748 Garching b. München
{buechner,matthes}@in.tum.de

Abstract. In this paper, we propose a new approach to model-driven development, which we call introspective model-driven development (IMDD). This approach relies heavily on some well-understood underlying abstractions, in order to bridge the abstraction gap between the requirements and the actual executable system. These abstractions are object-oriented programming languages and frameworks as a means of architectural abstraction. The main idea of IMDD is to annotate the extension points of a framework explicitly, which enables the automatic introspection of the defined metamodel. In a second step, a model of the customizations can be obtained by model introspection. There are two kinds of introspective frameworks – introspective blackbox and introspective whitebox frameworks. We developed an extension of the Eclipse IDE, which supports introspective model-driven development. Furthermore, we discuss the characteristics of the proposed approach, compared to established generative approaches.

1 Introduction

Dealing with the growing complexity of modern information systems is one of the challenges in computer science. One way to cope with this issue is the use of abstractions. There are some well-understood levels of abstraction as shown in figure 1.

The basic abstraction which hides some details of the underlying executable system is an object-oriented programming language (e.g. Java, C#). These languages are so-called *General Purpose Languages* (GPLs), which means that they are used to solve a broad spectrum of problems.

On top of object-oriented programming languages there are frameworks as a category of architectural abstraction. A framework embodies an abstract design for solutions to a family of related problems [1]. In order to solve a concrete problem, a framework has to be customized. The concrete task of customizing a framework involves the manipulation of low-level constructs like XML-files or code of the base programming language. The relationship between these constructs and the conceptual decisions in the problem space is not stated explicitly, and the intellectual distance between the adaptation constructs and the problem domain is pretty large.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 33–49, 2006.
© Springer-Verlag Berlin Heidelberg 2006

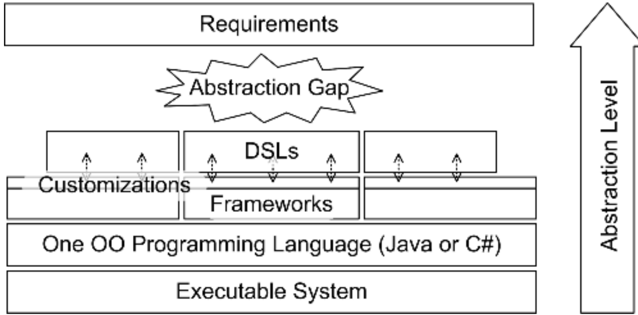


Fig. 1. Bridging the Abstraction Gap

One proposed solution to raise the level of abstraction of the framework customization process, is the use of a domain-specific language (DSL), which represents the extension points of a framewok in a usually declarative way. This approach is called model-driven development and implies an explicit connection between the high-level constructs of a DSL and the corresponding customization artifacts [3]. Technically speaking, there exists a transformation between the model and the customization artifacts. This distinguishes MDD from so-called *model-based* processes, in which models are merely used to illustrate certain aspects of a system in an understandable way, but the models created are not tied directly to the executable system. In this case, the models often do not "tell the truth" about the current system, and the creation of models is often seen as an overhead to the actual development process. So only with an MDD approach it is possible to obtain all benefits of using models to build information systems.

The most important point in using an MDD process is the explicit connection between the models and the actual customization artifacts. This leads to the question, in which direction the transformation is being applied. If the direction points from the model to the customization artifacts, this is called a *forward engineering* process [4]. Processes which use a transformation in the other direction are called *reverse engineering* processes. In this paper, we call these processes *top-down* and *bottom-up*.

A particular challenge for MDD processes results from the nature of the artifacts involved. In most cases, neither of them is sufficient to specify a complete system. Both, the model and the customization artifacts should be editable, and changes should lead to an immediate synchronization of the affected artifact. This is called *roundtrip engineering*. Realizing roundtrip engineering with a top-down process is a challenging task [5]. A promising approach to this problem is that of roundtrip visualizations [6].

All proposed implementations [7], [8] of model-driven development favor a top-down approach, in which they generate customization artifacts from models. To emphasize this, we call this approach *generative model-driven development*.

We propose in this paper a bottom-up approach, in which the high-level models are a rather transient result of an introspection process. We call this approach *introspective model-driven development.*

The article is structured as follows: In chapter 2 we give a short overview on generative model-driven development. In chapter 3, we introduce introspective model-driven development, which will be refined in chapter 4 and 5. In chapter 6, we will conclude with a comparison of the proposed approach with the prevailing approach to model-driven development.

2 Generative Model-Driven Development

An overview of generative model-driven development is illustrated schematically in figure 2. Similar to the life cycle of a framework the process is divided in a core development phase and an application development phase, with different roles of developers involved. The first result of the core development phase is the framework with its extension points. The creation of the core framework will be done by framework developers. In order to provide a more abstract view on the extension points of the framework, a language developer extracts the metamodel of the framework and creates a domain-specific language which reflects this metamodel. The extracted metamodel only reflects these parts of the framework, which will be customized in a declarative way. The metamodel will usually be specified using an existing meta-metamodel, as e.g., EMF [9] or MOF [10]. Given the metamodel, a *transformation developer* will create transformation rules which enable the transformation of models to concrete customization constructs. This will usually be done using a specific template language.

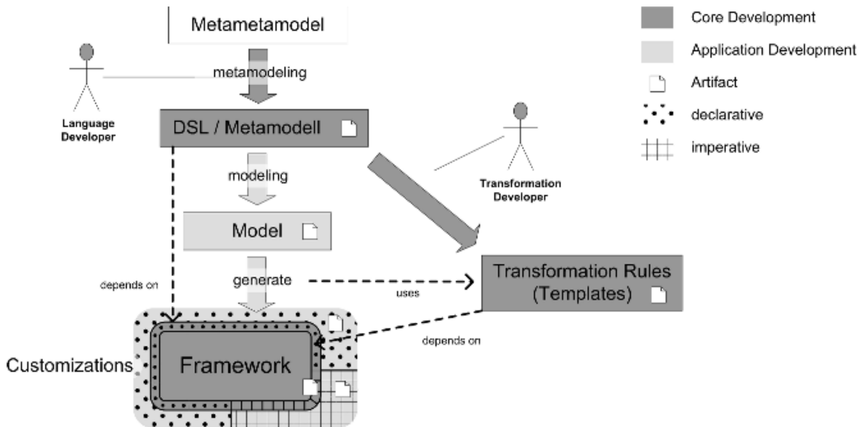


Fig. 2. Generative Model-Driven Development

In the application development phase, the framework user uses the metamodel and creates a model which solves a concrete problem. The creation of

concrete customization artifacts will be done by a generator based on the provided transformation rules. This only applies to these customizations which can be done declaratively. The imperative adaptations have to be done manually by the framework user.

3 Introspective Model-Driven Development

In this paper, we propose a bottom-up approach to realize model-driven development. We call this new approach *introspective model-driven development* (IMDD).

The main idea of IMDD is the construction of frameworks that can be analyzed in order to obtain the metamodel for customizations they define. The process in which the metamodel is retrieved is called *introspection*. The term introspection stems from the latin verb *introspicere*: to look within. Special emphasis should be put on the distinction between introspection and *reflection* in this context. We use both terms as they have been defined by the OMG [11]:

Table 1. Term Definitions

introspection	A style of programming in which a program is able to examine parts of its own definition. Contrast: reflection
reflection	A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
reflective	Describes something that uses or supports reflection.

According to the definition of reflective, *introspective* describes something that supports introspection. An introspective framework supports introspection in that its metamodel can be examined.

The whole process of introspective model-driven development is schematically shown in figure 3. The process is divided into the well known core development phase and application development phase. The first result of the core development phase is an introspective framework. An introspective framework supports introspection by highlighting all declaratively customizable extension points through annotations [12]. This enables the extraction of the metamodel by *metamodel introspection*. It is important to understand, that the metamodel is not an artifact to be created from the framework developer, but rather can be retrieved at any point in time from the framework.

The central artifact of the application development phase are the customizations to be made by the framework user. In IMDD it is possible to analyze these artifacts and to obtain a model representation of them. This is called *model introspection*. The model is an instance of the retrieved metamodel and can be

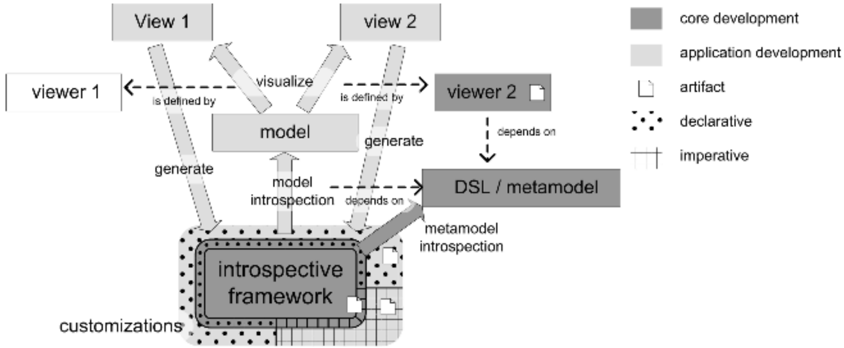


Fig. 3. Introspective Model-Driven Software Development

visualized by different viewers (i.e. visualization tools). There exist out-of-the-box viewers which can visualize an introspective model in a generic way. In some cases it is desirable to develop special viewers which visualize the model in a specific way. This will be done by framework developers in the core development phase. The manipulation of the model can be either done by using the views or by manipulating the customization artifacts directly. In both cases an updated customization artifact leads to an updated model and subsequently to an updated view. As a result of this, the model and the views are always synchronized with the actual implementation and can never "lie".

The main idea of introspective model-driven development is the direct extraction of the model and the metamodel from the framework artifacts which define them. There are two categories of frameworks which differ in the way adaptation takes place. *Blackbox frameworks* can be customized by changing association relationships flexibly. There are as many implementations as necessary to address all imaginable problems available as part of the framework core. The framework user just chooses the appropriate classes and configures their properties and the associations between them. In contrast, customization of *whitebox frameworks* takes place by creating subclasses of existing classes of the framework core. In this case the framework user has to provide concrete implementations.

Accordingly, the way introspective model-driven development is done is different for these kinds of frameworks. In the next chapter we will discuss introspective model-driven development for blackbox frameworks. In chapter 5, IMDD for whitebox frameworks will be introduced.

In order to enable introspective model-driven development we created a framework which supports blackbox introspection as well as whitebox introspection. This framework is called *Introspective Modeling Framework* – IMF. IMF provides its functionality by extending the post-IntelliJ-IDE Eclipse. Technically speaking, IMF consists of three Eclipse plugins.

Example. The process of developing an introspective framework and customizing it is illustrated using a simple example framework. We use a "textbook"

scenario described by Martin Fowler, in which we have a system that reads files and needs to create objects based on these files [2]. Each line can map to a different class, the class is indicated by a four-character code at the beginning of the line. The rest of the line contains the data for the object to be created. The following two lines result in the creation of two objects of type `ServiceCall` and `Usage` with attribute values as shown in an object diagram in figure 4:

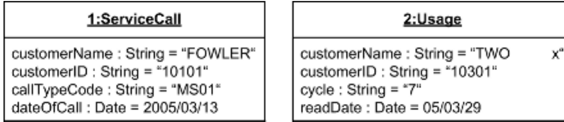


Fig. 4. Initialized Objects

```
#123456789012345678901234567890123456789012345678901234567890
SVCLFOWLER 10101MS0120050313.........................
USGE10301TWO x50214..7050329........................
```

The process of reading a file and instantiating objects accordingly should be adaptable in a high-level model-driven way. A conceptual metamodel which models the problem as an object-oriented design is shown in figure 5.

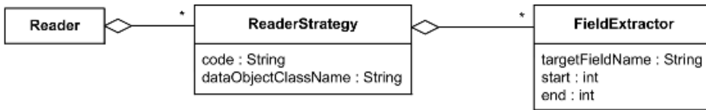


Fig. 5. Conceptual Metamodel

In the following we will show how to create an introspective blackbox and whitebox framework which realize a solution to this problem.

4 Blackbox Introspection

The framework core of a blackbox framework provides ready-to-use implementations of functionality, which only has to be customized to solve a family of related problems. The extension points of a blackbox framework are places which enable the adaptation of either elementary properties or associations between objects. In an introspective blackbox framework these extension points are tagged explicitly. That enables tool support for the customization process, which involves the selection and configuration of classes to be instantiated and the creation of associations between the objects constructed.

The idea of introspective blackbox frameworks is similar to that of *dependency injection* [13]. This means, that the instantiation of the framework classes is done by a dedicated component, which can be configured declaratively. The classes to be instantiated are rather passive in this process, they get their required dependencies “injected”. Introspective blackbox frameworks take this idea one step further by declaring all resources to be injected explicitly. This enables tool support.

4.1 Core Development

The core of a blackbox framework consists of classes which can have configurable elementary properties and associations with other classes, which are also configurable. These configurable elements define the metamodel of the framework, and a concrete configuration is a model which has to conform to the metamodel.

The key point of *introspective* blackbox frameworks is that these configurable elements are tagged explicitly using annotations as being configurable. This enables the automatic introspection of the metamodel and as a result of this it is possible to support the modeling step.

There are two types of annotations, which enable the identification of configurable properties and associations. Configurable properties are tagged using the annotation type `Property`. In order to create the configurable property `code` of the class `ReaderStrategy` in our example, it is necessary to tag the definition of the attribute as shown:

```
@Property(description="these four letters indicate this strategy")
String code;
```

Configurable relationships are created using the annotation type `Association`. Creating the association between the classes `Reader` and `ReaderStrategy` is done with following piece of code:

```
@Association List<ReaderStrategy> readerStrategies;
```

This leads to the meta-metamodel of blackbox introspection as shown in figure 6. A configuration consists of many configurable classes which can have many customizable properties and associations. An association connects configurable classes with each other. The icons besides the classes `ConfigurableClass`, `Property` and `Association` can be used to annotate introspective elements in class diagrams.

An implementation of the example problem as an introspective blackbox framework looks like shown in figure 7. The introspective elements are annotated using the icons mentioned above.

4.2 Application Development

So far we have looked at how to create the introspective framework core. This task is done by the framework developer and consists in writing a “plain old

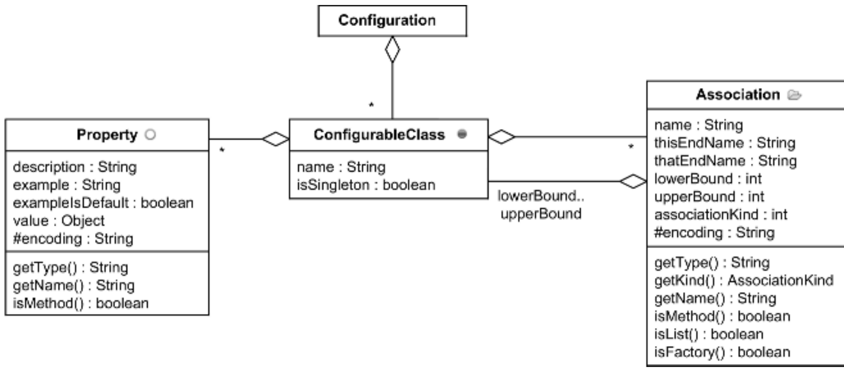


Fig. 6. The Meta-Metamodel of Blackbox Introspection

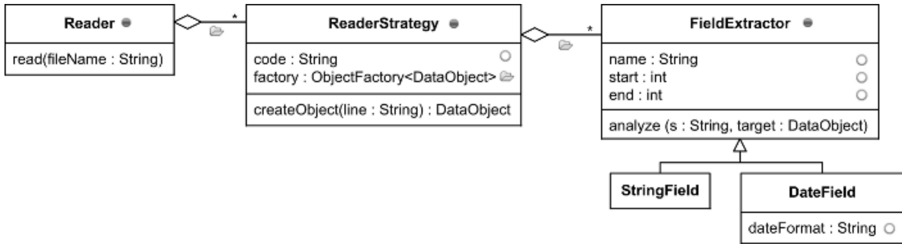


Fig. 7. Implementation of the Example Problem as an Introspective Blackbox Framework

framework" with some additional annotations to tag the extension points. As a result it is possible to retrieve the metamodel of the framework by doing introspection on the framework core.

In the second phase of the life cycle, the framework user customizes the framework to solve a concrete problem. The customization of an introspective blackbox framework is done using the *IMF Blackbox Modeler* tool. Technically speaking is this a plugin for the Eclipse IDE which analyzes the metamodel of the framework core. Based on this metamodel the Blackbox Modeler provides a view which enables the creation of a model which is an instance of the metamodel. A screenshot of the modeler, in which the example problem is modeled, is shown in figure 8. From the modeler view, which shows the model, it is always possible to navigate to the corresponding metamodel element, which is also shown in figure 8.

5 Whitebox Introspection

As already mentioned, the customization of whitebox frameworks is done by providing implementations of abstract classes of the framework core. More specifically, the framework user specifies the desired behavior by implementing methods.

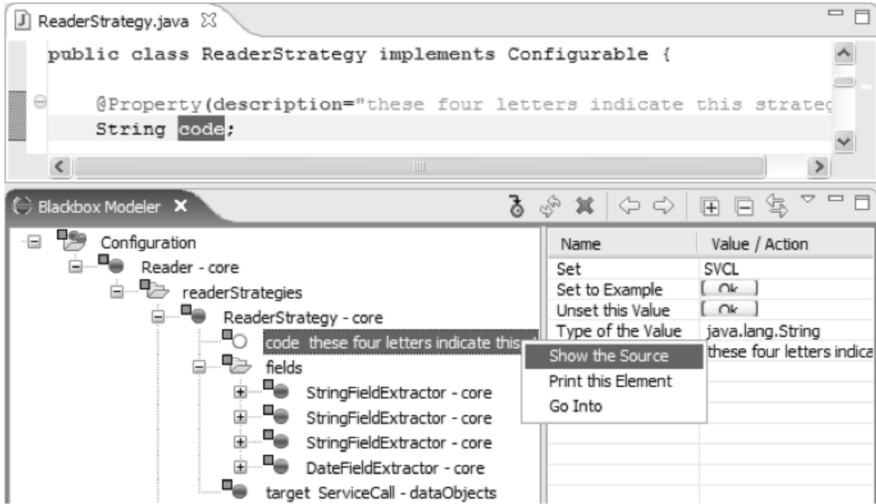


Fig. 8. Modeling Perspective for Blackbox Introspection in the Blackbox Modeler

These methods are called *hook methods* and represent the extension points of the framework [15]. Regarding introspective whitebox frameworks there are two kinds of hook methods – introspective and non-introspective hook methods. Customization of introspective hook methods can be done using a declarative programming style, while implementing non-introspective hook methods requires imperative constructs. The main idea of whitebox introspection is to annotate introspective hook methods in the framework core and to analyze the declarative customization artifacts. The analysis of the structure of the introspective methods results in the metamodel of the framework, and the analysis of the customizations leads to a model of the provided adaptations.

To build a whitebox framework, which addresses our example problem, we create an abstract class `ReaderStrategy`. This abstract class specifies, that subclasses have to provide a concrete value of the `code` property:

```
public abstract class ReaderStrategy {
    @Introspective public abstract String getCode();
    ...
```

The annotation type `Introspective` indicates, that this method is an introspective method, which means that it has to be implemented in a declarative way. In fact, this is the simplest kind of an introspective method, the so-called *value-method*. A value-method has no parameters and returns either a primitive value or an object of type `String` or `Class`.

In order to specify the programming model formally, which can be used to implement the method we use a context-free grammar. This grammar is used to restrict the expressive power of the underlying programming language to

a declarative programming model. We define our grammar based on the non-terminals used by the Eclipse project JDT [16]. Non-terminals are shown in italic type, terminal symbols are shown in fixed width font. Non-terminals introduced by us are printed in ***bold italic*** face.

The non-terminal which defines the programming model for customizing value-methods looks like the following:

ValueMethod_M1 :
  { Modifier } ***ValueMethodType*** SimpleName `( )` {
    `return` ***AbstractValue*** ; }

ValueMethodType :
  `String` | `Class` | `boolean` | `byte` | `short` | `char` |
  `int` | `long` | `float` | `double`

The return type of a value-method is therefore restricted to be of either primitive type or one of `String` or `Class`. The return statement is defined by the non-terminal ***AbstractValue***:

AbstractValue :
  Value
  $Name_{ValueVariableName}$

This can be either a value of one of the following types, or a variable name:

Value :
  BooleanLiteral | *CharacterLiteral* | *NumberLiteral* |
  StringLiteral | *TypeLiteral* | *NullLiteral*

We call these two ways to return the result *by-value* and *by-constant*. The variable name has to be bound to a field declaration which defines a variable which is declared as being final:

FinalValueFieldDeclaration :
  [Javadoc] ***FinalModifiers***
    ValueMethodType $SimpleName_{ValueVariableName}$ `=` ***Value*** ;

The non-terminal ***FinalModifiers*** specifies a set of modifiers which contains the final modifier. A valid implementation of the introduced method **getCode** looks like the following:

```
@Override public String getCode() {
    return "SVCL";
}
```

An alternative solution in which case the result is returned by-constant is like the following:

```
final CODE = "SVCL";

@Override public String getCode() {
    return CODE;
}
```

Because of the declarative programming model, it is possible to analyze the customization artifacts. This analysis is called *model introspection* and is supported by the IMF-Whitebox Modeler tool. A screenshot, which shows the tool, is illustrated in figure 9. In this view, it is possible to manipulate the value of the property, which results in a manipulation of the code and a subsequent redrawing of the model. It is also possible to navigate to the construct which defines the metamodel for the current model element.

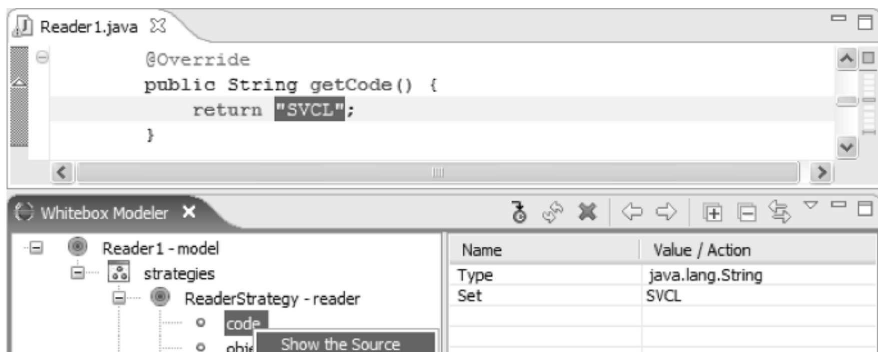


Fig. 9. An Introspective View of a Value-Method in the Whitebox Modeler

The value-method described so far enables us to model elementary properties. In order to build a framework which addresses the example problem we also have to model associations. This can be done using another kind of introspective method, the so-called *objects-method*. An objects-method has no parameters, but returns either one or many objects of a specific type. Unlike for the value-method, there are multiple introspective programming models, which can be used to implement an objects-method. The simplest programming model returns just a newly created object, which is similar to the programming model of the value-method. An in-depth treatment of all identified programming models will be provided in [14]. We introduce here the fields-by-type programming model, which allows the definition of a set of objects by declaring variables. The specification of an objects-method using the fields-by-type programming model in the framework core looks like the following:

```
public abstract class Reader {

    private List<ReaderStrategy> strategies;

    @Introspective public List<ReaderStrategy> getStrategies() {
        if(strategies == null) {
            strategies = FieldFinder.getFields
                (this,ReaderStrategy.class);
        }
        return strategies;
    }
    ...
```

This method returns all final fields which specify an object of type `ReaderStrategy`. A specification of a concrete strategy looks like this:

```
public class Reader1 extends Reader {
    final ReaderStrategy STRATEGY_1 = new ReaderStrategy() {
    ...
```

As a result of this we have introduced a meta-metamodel of whitebox introspection, which is shown in figure 10. The meta-metamodel we show here is a simplified version of the one introduced in [14]. A model consists of introspective classes. An introspective class has introspective methods, which can be either value-methods or objects-methods. Using this meta-metamodel it is possible to build introspective whitebox frameworks.

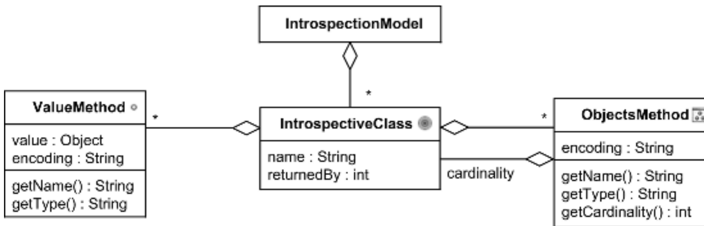


Fig. 10. The Simplified Meta-Metamodel of Whitebox Introspection

In figure 11 the four meta-layers and their equivalents in the case of whitebox introspection are shown. The metamodel at M2 is defined by the framework core by using introspective methods. The model at the meta-layer M1 is defined by the framework user. The model is represented as declarative code of the host language. Modeling can be done either by writing code manually or by using the Whitebox Modeler to do so at a rather high level of abstraction. The programming model, which is used to express the model, depends on the actual

introspective method. The Whitebox Modeler also verifies the correct use of the programming model.

We now study how to solve the example problem with an introspective whitebox framework. Our example can be customized completely declaratively, so it can be solved using a blackbox framework. To demonstrate one of the advantages of whitebox introspection we vary the example scenario a little bit. Let's assume, the created objects should be used to do some rather complex business logic directly after their creation. This business logic should be done using a specific API, which enables the manipulation of some data store. The most convenient way to express such kind of business logic is by writing some imperative code, which encodes the desired behavior. This means, that there are declaratively customizable parts of the framework as well as imperatively customizable part. One benefit of our approach lies in the uniform treatment of both introspective and non-introspective hook methods. The content of the introspective methods will be analyzed, whereas the non-introspective are not analyzed by the modeler.

A class diagram of an introspective whitebox framework, which addresses the modified example problem, is shown in figure 12. The method `processObject` of the class `ReaderStrategy` is a non-introspective hook method which gets the created object as a parameter and does the business logic. The framework user can use the full power of the base language to specify the business logic here.

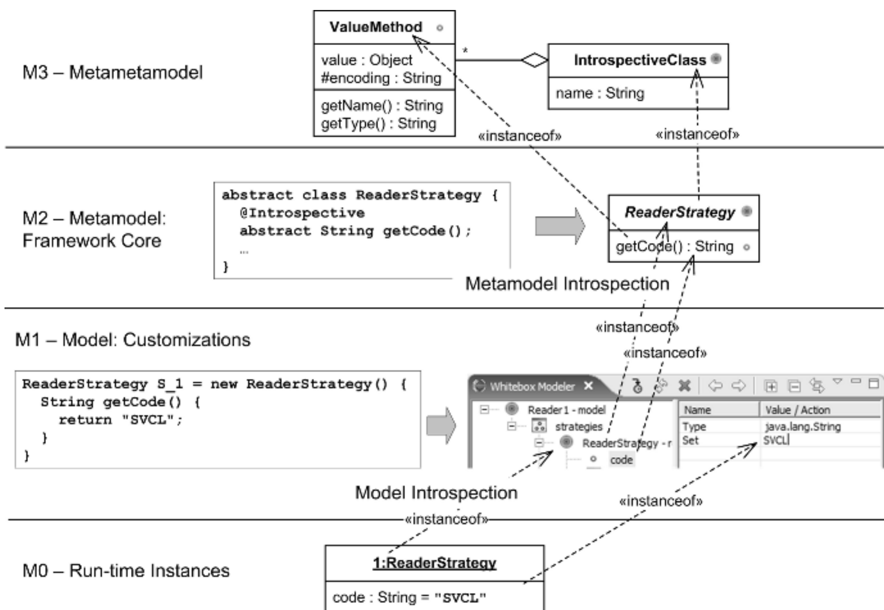


Fig. 11. Four Meta-Layers of Whitebox Introspection

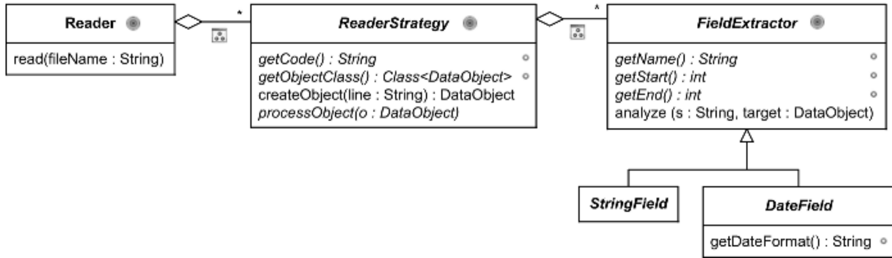


Fig. 12. Implementation of the Example Problem as an Introspective Whitebox Framework

5.1 Case Study

In [14] we present our experience in building two whitebox frameworks as part of a commercial knowledge management system. The first one is a web-visualization framework. The main abstraction of this framework are so-called *handlers*. A handler reacts on requests by reading parameters, doing some business logic and rendering response pages in the end. Except for the business logic, all aspects of the handlers are realized introspectively and can be analyzed and modeled. We have built a derivative of the Whitebox Modeler, which is tailored specifically to this framework. In order to render the dynamic response pages, the framework uses HTML-templates. By means of the introspective model, it is possible to check the consistency of the templates with the code which instruments them [17]. The whole knowledge management system consists of approximately 500 handlers.

6 Discussion and Concluding Remarks

We believe, that modeling as a means of building and understanding systems at a rather high level of abstraction should play a more important role in software engineering. Furthermore, we think, that model-driven approaches offer a lot of benefits over merely model-based approaches. The prevailing approach to realize model-driven development is the generation of artifacts which customize frameworks, as shown in figure 1. In this paper, we propose an alternative approach to realize MDD, which we call introspective model-driven development (see figure 13). In the following we will discuss the implications of using introspective vs. generative model-driven development.

IMDD relies on some infrastructure, which has to be in place. The base language used has to be a statically typed object-oriented programming language. In our case, we chose Java as the base language. The second prerequisite is the existence of a "post-IntelliJ-IDE", on top of which a tool to support IMDD can be created. We chose the Eclipse IDE to build IMF, which is a framework that supports IMDD. According to the two types of introspective frameworks - introspective blackbox and introspective whitebox frameworks, IMF provides generic

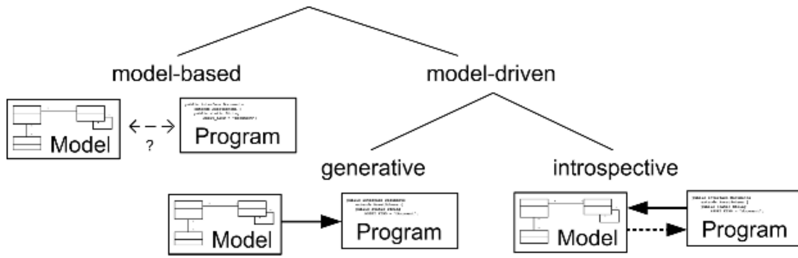


Fig. 13. Modeling Approaches

modelers for both of them. Therefore, using IMF out-of-the-box it is possible to do introspective model-driven development immediately. In some cases, it is useful to create tailored modelers, which will be supported by IMF.

6.1 Advantages of IMDD

IMDD is a *single-source* approach, which means that the metamodel and the model are respectively represented by exactly one artifact. This is not the case for generative approaches, in which information about the metamodel is encoded implicitly in the framework core, and in the explicit metamodel of the DSL. The same is true for the models. They are represented as artifacts of the modeling process, as well as customizations, which will be generated. To specify the transformation process, there are additional artifacts, which rely on the conceptual metamodel, and the way the concrete customization artifacts look like. All this leads to a lot of redundancies and a lot of artifacts, which have to be consistent.

In introspective model-driven development the metamodel is represented by the annotated framework core, and the model is represented directly by the customization artifacts. In both cases, these artifacts are used to specify the executable system, as well as to provide modeling information. This means, that in IMDD “code is model” [18]. Code means here also declarative customization artifacts, which configure an introspective blackbox framework. The model is a transient view on the underlying code. The most striking advantages of IMDD follow from this fact.

At first, this enables roundtrip visualizations, which are hard to achieve for generative approaches [5]. As another immediate implication of this, the model “never lies”. This means, that the model reflects properties of the system precisely all the time.

Because the modeling information in IMDD is represented by code, refactoring the metamodel of the framework [19] can be done easily using a post-IntelliJ-IDE. In the case of an introspective whitebox framework, also the model will be refactored accordingly. Broadly speaking, keeping the involved artifacts consistent is quite easy in IMDD. In a generative approach, evolving the framework core means evolving the metamodel, the transformation rules and the models manually in parallel.

Another advantage of IMDD is the possibility to achieve symbolic integration [2] between declarative models and imperative artifacts. This makes it easy to mix both programming styles, and get the benefits of modeling the declarative aspects on a high level of abstraction. Using a generative approach, it is quite complicated to integrate both paradigms, by e.g. editing generated artifacts, using protected source code areas.

Furthermore, we consider the introspective approach as being *lightweight*. This means, that no additional meta-metamodel is needed to do IMDD, and that the overall process is much simpler. As a meta-metamodel, we use some of the capabilities of the object-oriented base language. There are no additional languages to be learned by the developer. Generative approaches are more heavyweight, because they involve an additional meta-metamodel, and a language to do the transformation. As another aspect of using the base language to do metamodeling, fundamental consistency constraints on the metamodel will be checked by the compiler of the base language. In the case of introspective whitebox frameworks, the compiler also checks some aspects of the well-formedness of the model using rich typing, binding and scoping rules of the base language.

The code-centricity of IMDD matches well with the development approaches used in practice.

6.2 Disadvantages of IMDD

IMDD relies on the explicit annotation of the extension points in the framework core, so it requires the construction of introspective frameworks. It is not possible to do IMDD with classical frameworks, which do not support this development approach. As a consequence of this, doing introspective development with the existing frameworks is not possible. They have to be modified, in order to be introspective.

References

1. Ralph E. Johnson and Brian Foote, *Designing reusable classes.* Journal of Object-oriented Programming, vol. 1(2), pp. 22-35, 1988.
2. Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?.* http://www.martinfowler.com/articles/languageWorkbench.html
3. Markus Völter and Thomas Stahl, *Model-Driven Software Development.* John Wiley & Sons, 2006.
4. Elliot J. Chikofsky and James H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, vol. 7, 1990.
5. Shane Sendall and Jochen Küster, *Taming Model Round-Trip Engineering.* Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada, 2004.
6. Stuart M. Charters, Nigel Thomas, and Malcolm Munro, *The end of the line for Software Visualization?.* VISSOFT 2003: 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, Amsterdam, September 2003.

7. David S. Frankel, *Model Driven Architecture – Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.
8. Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, *Software Factories.* Wiley Publishing, Inc., 2004.
9. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose, *Eclipse Modelling Framework.* Addison-Wesley Professional, 2003.
10. OMG – Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification.* http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-15.pdf
11. OMG – Object Management Group, *Common Warehouse Metamodel (CWM), v1.1 – Glossary.* http://www.omg.org/docs/formal/03-03-44.pdf
12. Joshua Bloch, *JSR 175: A Metadata Facility for the Java Programming Language.* http://www.jcp.org/en/jsr/detail?id=175
13. Martin Fowler, *Inversion of Control Containers and the Dependency Injection Pattern.* http://www.martinfowler.com/articles/injection.html
14. Thomas Büchner, *Introspektive modellgetriebene Softwareentwicklung.* Technische Universität München, München, Dissertation (in Vorbereitung).
15. Wolfgang Pree, *Essential Framework Design Patterns.* Object Magazine, vol. 7, pp. 34-37, 1997.
16. Eclipse Foundation, *Eclipse Java Development Tools (JDT) Subproject.* http://www.eclipse.org/jdt/
17. Stefan Käck, *Introspektive Techniken zur Sicherung der Konsistenz zwischen Webpräsentationsvorlagen und Anwendungsdiensten.* Diplomarbeit, Technische Universität München, 2005.
18. Harry Pierson, *Code is Model.* https://blogs.msdn.com/devhawk/archive/2005/-10/05/477529.aspx
19. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

Eliminating Execution Overhead of Disabled Optional Features in Connectors

Lubomír Bulej[1,2] and Tomáš Bureš[1,2]

[1] Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Prague, Czech Republic
Phone: +420-221914267; Fax: +420-221914323
{bulej,bures}@nenya.ms.mff.cuni.cz

[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague, Czech Republic
Phone: +420-266053831

Abstract. Connectors are used to realize component interactions in component systems. Apart from their primary function, which is mediating the communication, their implementation can also support additional features that, while unrelated to the primary function, may benefit from their placement in connectors. Such features are often optional in the sense that they can be activated and deactivated at run-time. The problem is that even if they are disabled, their very presence in the connector incurs certain overhead. In this paper, we describe an approach to eliminate this overhead by reconfiguration of the connector implementation. Besides connectors, the approach is applicable to similar technologies such as reflective middleware and other architecture-based component systems and frameworks.

Keywords: Component systems, software connectors, runtime reconfiguration.

1 Introduction

In component systems with support for distribution, the design-time connections among components usually represent a more powerful concept than just a plain reference as known from programming languages. Such connection, a hyper-edge in general, connects components that participate in some kind of interaction. The endpoints of a connection correspond to the roles the connected components assume in the interaction.

At runtime, besides entities implementing the components, additional entities are required to realize the interaction modeled by such connections. The exact composition of entities required to implement a given interaction depend on the communication style governing the interaction. In case of e.g. procedure call, these entities could be a stub and a skeleton, or a plain reference, depending on the location of the participating components.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 50–65, 2006.
© Springer-Verlag Berlin Heidelberg 2006

To model the component interactions, many component systems have introduced connectors as first-class design entities that model the interactions among components. At the design level, connectors represent a high-level specification of the interactions they model, aggregating requirements imposed on the interaction. The specification is then used for transformation of a design-time connector into runtime connector, which implements the interaction and has to satisfy the requirements from the specification.

Some of the requirements may not be directly related to the primary function of a connector, which is to enable communication among components. The properties a runtime connector needs to have to satisfy such requirements are called *non-functional properties*. A connector implementation gains such properties by implementing additional functionality unrelated to its primary function, such as logging, performance and behavior monitoring, security[1], etc.

Some of the features implemented by a connector to satisfy connection requirements may be *optional*, which means that they do not need to be active at all times. The implementation of a connector may support selective activation and deactivation at runtime, especially if a feature incurs considerable *dynamic overhead* when active.

The problem is that even when an optional feature is disabled, its presence in a connector may incur certain *static overhead*. The overhead is caused by the mechanism used to integrate an optional feature with other code. Compared to its dynamic overhead, static overhead of an optional feature tends to be small or even negligible, depending on the integration mechanism. Nevertheless, the existence of the static overhead and the lack of data quantifying its impact is often a reason for not including useful features such as logging or monitoring in production-level applications. Such features can provide an application administrator with tools for diagnosing problems in a running application.

As discussed in [1], due to complexity of contemporary software, problems and misbehavior occurring in production environment, which typically does not provide sufficient diagnostic tools, are hard to reproduce in development environment where the tools are available. For this reason, even production-level applications should always provide features that can be activated at runtime and that can assist in diagnosing hardly reproducible problems. Moreover, as long as they are not used, those features should have no impact on the execution of the application.

In this paper, we present an approach to eliminate static execution overhead of disabled optional features in the context of one particular model [4] of architecture-based connectors. The model is being developed within our research group with focus on automatic generation of connector implementation from the high-level specification. The issues concerning efficiency of generated connectors have prompted the research presented in this paper.

[1] Security properties such as authentication or encryption cannot be considered entirely unrelated, but are still considered non-functional, because they are not essential.

Even though we present it on a specific connector model, the scope in which the approach can be applied is much broader. The principal requirements are construction through composition and the ability to track dependencies in a composite at runtime. These requirements are satisfied even in very simple component environments, therefore the presented approach is directly applicable e.g. to component-based middleware (e.g. the reflective middleware by Blair et al. [2], Jonathan [3], etc.), which in fact plays the role of connectors.

1.1 Goals of the Paper

The main goal is to eliminate the static execution overhead associated with optional connector features that have been disabled. This will allow including optional features such as logging, or monitoring even in production-level applications without impacting performance while the features are disabled.

To solve the problem in the context of architecture-based connectors, we need an algorithm that allows us to propagate changes in the runtime structure of a connector implementation through the connector architecture. This in turn cannot be done without imposing certain requirements on the connector runtime.

The goals are therefore to devise an algorithm for propagating changes in a connector architecture, to formulate the requirements that a connector runtime must satisfy for the algorithm to work, and to prove that the algorithm always terminates in a finite number of steps.

1.2 Structure of the Text

The rest of the paper is organized as follows: Section 2 gives an overview of the connector model used as a testbed for our approach, Section 3 provides discussion of the overhead associated with optional features with respect to the connector model, Section 4 outlines the solution and Section 5 describes in detail the proposed changes in connector runtime and the reconfiguration process used to eliminate the static execution overhead of disabled optional features, Section 6 provides an overview of related work, and Section 7 concludes the paper.

2 Connector Model

Throughout the paper, we use a connector model described in our earlier work [4] as a test bed. The model has a number distinguishing features, which are not commonly present in other connector models. Connectors are modeled by connector architectures describing composition of entities with limited, but well-defined functionality. The key feature with respect to the approach presented in this paper is that the design architecture of a connector is reflected in runtime architecture of the connector implementation and can be traversed and manipulated at runtime.

2.1 Connectors from the Outside

Conceptually, a connector is an entity exposing a number of attachment points for components participating in an interaction. Technically, due to its inherently distributed nature, a connector comprises a number of *connector units*. Each unit represents a part of a connector that can exist independently in a *deployment dock*, which hosts instances of components and connectors. A connector unit communicates (strictly) locally with components attached to it and remotely (using middleware) with other connector units.

An example in Figure 1 shows client components A, B, and C connected to the Server component using connectors. In case of components B and C the respective connectors cross the distribution boundary between address spaces.

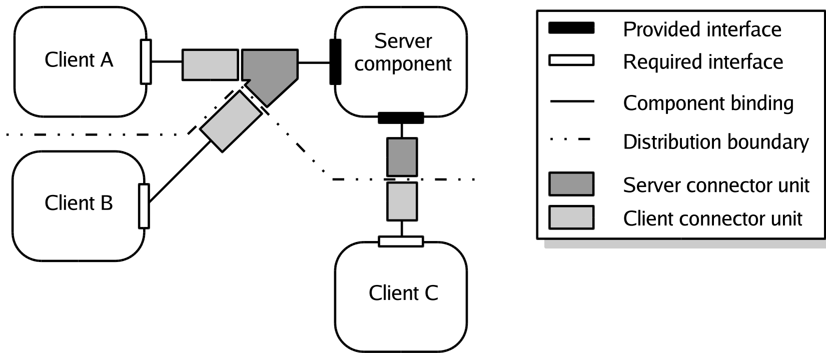


Fig. 1. Using connectors to mediate communication among components

2.2 Connectors from the Inside

The construction of connectors is based on hierarchical composition of well-defined entities with limited functionality. A connector is made of connector units, and connector units are made of *connector elements*. The composition of connector elements is described by *connector architecture*.

Since the connector elements can be nested, the entire architecture forms a hierarchy with the connector as a whole represented by its root. The internal nodes represent composite elements which can contain other elements, and the leaves represent primitive elements which encapsulate implementation code. Connector units at the second level of the hierarchy are also connector elements, except with certain restrictions on bindings among other elements.

While the connector architecture models composition of *connector element types* [4], *connector configuration* provides a white-box view of a connector, which is obtained from the architecture by assigning concrete implementation to the element types present in the architecture. A connector configuration therefore fully determines the connector implementation and its properties.

An example of a connector configuration is given in Figure 2, which shows a connector realizing a remote procedure call. The connector consists of one server unit and one client unit. The client unit consists of an *adaptor* (an element realizing simple adaptations in order to overcome minor incompatibility problems) and a *stub*. The server unit comprises a *logger* (element responsible for call tracing), a *synchronizer* (element realizing a specific threading model), and a *skeleton collection* (element which groups together multiple skeleton implementation using different middleware, thus enabling access to the server unit via different protocols).

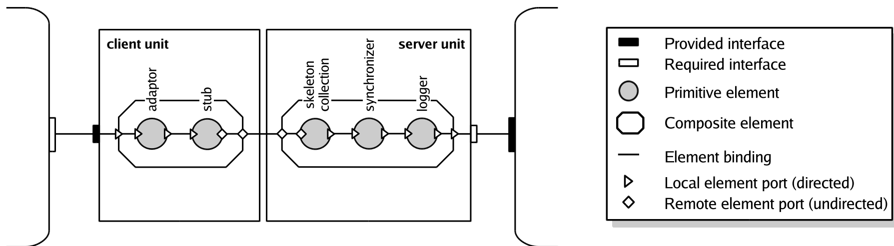


Fig. 2. Connector at runtime

A connector element communicates with other elements or with a component interface only through designated ports. There are three basic types of element ports in the model: (a) *provided ports*, (b) *required ports*, and (c) *remote ports*. Based on the types of ports connected together we distinguish between local (required-to-provided or provided-to-required ports) and remote (between multiple remote ports) bindings.

Local bindings are realized via local calls, which limits their use to a single address space. Ports intended for local bindings thus serve for (a) element-to-element communication within a connector unit and (b) element-to-component communication with the component attached to a respective connector unit.

Remote bindings represent a complex communication typically realized by middleware (e.g., RMI, JMS, RTP, etc.). We do not attempt to model this communication or capture the direction of the data flow in our connector model – instead we view these bindings as undirected. To support other communication schemes than just point-to-point (e.g., broadcast), we model a remote binding as a hyper-edge which connects multiple remote ports.

The exact implementation of a remote binding depends on the participating elements. Their responsibility is to provide remote references or use remote references for establishing a connection. From the point of view of the connector model, a remote binding only groups together ports of elements sharing the same set of remote references.

Due to inherently distributed nature of a connector, there are restrictions on the occurrence of local and remote ports in the architecture and the bindings among them. At the top level of the architecture, a connector can only expose

local ports. Remote bindings can only occur at the second level of the architecture, i.e. among connector elements representing connector units. In composite elements, only local bindings between child elements, and delegations between the child and parent element ports are allowed. Remote port occurring in a non-unit element must be delegated to the parent element.

2.3 Connectors at Runtime

At runtime, each connector element is represented by a primary class which allows the element to be recognized and manipulated as an architectural element. Depending on the types of declared ports, the primary class has to implement the following interfaces: *ElementLocalClient* (if the element has a required port), *ElementLocalServer* (if the element has a provided port), *ElementRemoteClient* (if the element has a remote port and acts as a client in a remote connection), and *ElementRemoteServer* (if the element has a remote port and acts as a server in a remote connection). The primary class aggregates the control interfaces that can be used for querying references to element ports (server interfaces) and for binding ports to target references (client interfaces). The signatures of the control interfaces are shown in Figure 3.

2.4 Optional Features in Connectors

Since the local bindings among connector elements are realized by local calls, a sequence of connector elements with complementary ports with the same interface may result in a chain of elements through which method invocations have to pass. From this point of view, optional features utilizing interception or filtering to perform their function will fit well in the connector model. Interception is used to include activity related to invocation of specific methods in the call path, while filtering operates on the content passed along the call path.

Examples of optional features implemented through interception are logging, performance or behavior monitoring, or any other function requiring method-level interception, such as the stub and skeleton. Features implemented through filtering may include encryption, compression, and other data transformations. Figure 2 shows a connector architecture with an optional logging feature implemented by the *logger* element.

3 Overhead of Optional Features

The activity of optional features is the source of dynamic memory and execution overhead. The overhead tends to be significant, but is only present when an optional feature is enabled. For this reason, the dynamic overhead is not further analyzed in this paper.

The presence of an optional feature in a connector (or an application in general) is the source of static memory and execution overhead. The overhead is caused by the mechanism used to optionally include the implementation of a

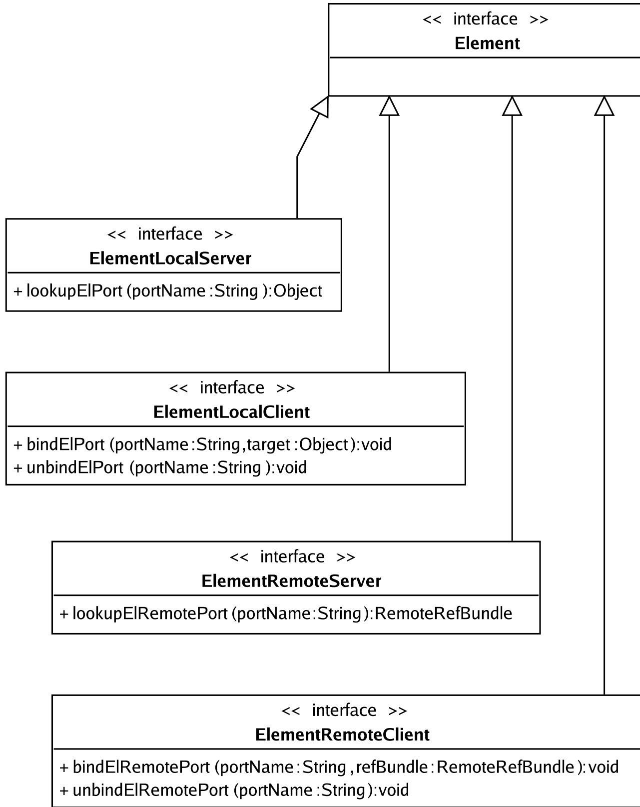


Fig. 3. Interfaces implemented by an element

feature in a connector and by itself it tends to be rather small, even insignificant. The approach presented in this paper aims to address situations where these isolated cases accumulate.

3.1 Static Memory Overhead

The static memory overhead associated with optional features in connectors is caused by the additional code required to implement the desired operation. Eliminating the static memory overhead of a disabled optional feature requires eviction of the code that implements it from memory.

The ability to evict code from memory depends on the environment and may not be always possible. In case of virtual machine environment with garbage collection such as Java, the code cannot be evicted explicitly – we can only attempt to ensure that it can be garbage-collected.

Considering the relatively light-weight connector runtime and the nature of the optional features, we assume the code implementing them will represent only a small fraction of all application code. Therefore we expect the static memory overhead to be negligible and do not specifically target it in our approach.

3.2 Static Execution Overhead

The static execution overhead associated with optional features is tied to the mechanism through which these features are included in the application call paths. This mechanism is based on invocation indirection, which is inherent to the connector model we are using (and to all component environments where the architecture is preserved at runtime). The concept of a connector architecture consisting of connector elements allows us to compose connectors from smaller blocks providing a clearly defined function, while the hierarchy serves to limit an element's awareness of its surroundings.

This allows adding optional features to connectors, because each element implementing a pair of complementary ports of the same type can serve as a proxy for another element. This is the case of the *logger* element in Figure 2. However, if multiple elements implementing optional features are chained together, the method invocation will have to go through all the elements in the chain, even if all optional features in these elements are disabled.

One may argue that in case of a single element, the execution overhead of one additional layer of indirection is negligible and can be tolerated. However, as mentioned above, connector elements are simple building blocks intended for composition. Therefore we expect connector architectures combining multiple optional features to achieve that through combination of multiple connector elements. In such case, the method invocation will accumulate overhead in each element of the chain, because before passing the invocation to the next element, each element must also decide whether to invoke the code implementing the feature. The overhead thus accumulated most probably will not be prohibitive, but it may not be negligible anymore.

4 Outline of the Solution

Eliminating the static execution overhead associated with a disabled optional feature on a specific application component interface requires removal of an element implementing the feature from a chain of elements intercepting method invocations on that interface.

This operation is similar to removal of an item from a single-linked list – the predecessor of the item must be provided with a link to its new successor. However, in case of connector architecture, the list items are not data but code. The problem thus gets complicated by the fact that connectors are distributed entities, and that each connector element is autonomous in deciding (anytime during execution) when it wants to be removed from the call chain and when it wants to be part of it again. Additionally, the connector architecture is hierarchical, which further complicates the management of the call chain.

Connector elements in a chain intercepting particular component interface are linked to each other using a pair of complementary ports. Their required ports are bound to ports of the same type provided by their successors. Since each element is aware of its internal architecture but not of its place in the surrounding architecture, the binding between ports of two sibling elements must be

established by the parent element. Based on its architecture, the parent element queries the child elements for references to their provided ports and provides these references to other child elements that require them.

Thus, when an element wants to be excluded from a call chain, it can simply realize it by providing the target reference associated with its required port as a reference to its own provided port. In other words, when queried for a reference to its provided port, an element returns a reference it has already obtained as a target for one of its required ports. Method invocations on the provided port are thus passed directly to the next element in the call chain.

Although the trick for excluding elements from the call path is simple, there are several problems that complicate its usage (a) for initial setup of connector architecture at startup and (b) for reconfiguration at runtime, because of the above mentioned connector element autonomy.

When creating the initial architecture (with some of the optional features disabled by default), it is necessary to bind the elements in certain order for the idea to work. The order would correspond to a breadth-first traversal of a graph of dependencies among the ports. The binding has to be done recursively for all levels of the connector architecture hierarchy, because through delegation and subsumption between the parent and child elements, the (virtual) graph of dependencies between the ports may cross multiple levels of the connector architecture hierarchy. Additionally, since the concept of connector elements is based on strong encapsulation, the child elements appear to their parent entity as a black-box. Consequently, there is neither central information about the connector architecture as a whole (viewed as a white-box), nor is it possible to explicitly construct the dependency graph for the entire connector.

In our approach, we address both mentioned situations by a reconfiguration process initiated within an element. The reconfiguration process starts when an event occurs that causes a reference to a provided port (exposed by an element) to be no longer valid and it is necessary to instruct all neighboring elements using this reference (through their required ports) to obtain an updated one. Because an element where such even occurs does not have information about its neighbors, it notifies its parent entity (the containing element or the connector runtime), which uses its architecture information to find the neighboring elements that communicate with the originating element, and are thus affected by the change of the provided reference. If there is a delegation leading to the originating element, the parent entity must also notify its own parent entity.

This reconfiguration process may trigger other reconfigurations when a depending element is excluded from the call chain – a change of the target reference on its required port causes a change of a reference to its provided port.

The reconfiguration process addresses the two situations – (a) initial setup of the connector architecture and (b) runtime reconfiguration – in the following way. In case of (a) we do not impose any explicit ordering on instantiation and binding of connector elements. When an element that wants to be excluded

from the call chain is asked to provide a reference which it does not have yet (because its required port has not been bound yet), it returns a special reference *UnknownTargetReference*. As soon as the required port gets bound and the target reference (that should have been returned for the provided port) becomes known, the element initiates the reconfiguration process for the affected provided port, which ensures propagation of the reference to the affected elements.

In (b) the inclusion/exclusion of an element in/from a call chain affects a reference provided by a particular provided port – either a reference to an internal object implementing the port (including the optional feature code in the call path) or a target reference of a particular required port should be provided. In both cases the change is propagated to the depending neighbor elements through the reconfiguration process.

5 Reconfiguration Process

The reconfiguration process outlined in the previous section allows us to eliminate the static execution overhead of disabled optional features in connectors. In this section we show what extensions must be introduced to the connector runtime and what functionality must be added to the element control interfaces presented in Section 2.3.

Additionally, we show that the algorithm always terminates, which is not an obvious fact due to reconfiguration process triggering other reconfigurations. Due to space constraints, we have omitted additional discussion of the algorithm. The discussion, along with a more detailed description of the algorithm and the proof of termination can be found in [5].

5.1 Reconfiguration Algorithm

The reconfiguration algorithm is executed in a distributed fashion by all entities of the connector architecture. Because it operates on a hierarchical structure with strong encapsulation, each participant only has local knowledge and a link to its parent entity to work with.

The link is realized through *ReconfigurationHandler* interface, which is implemented by all non-leaf entities of the connector architecture, i.e. composite elements and connector runtime, and is provided during instantiation to all non-root entities, i.e. composite and primitive elements. When the reconfiguration process needs to traverse the connector architecture upwards (along the delegated ports), the respective connector element uses that interface to initiate the reconfiguration process in its parent entity. In the case of our connector model, the *ReconfigurationHandler* interface contains two methods that serve for invalidating provided and remote server ports of the element initiating the reconfiguration process.

When disabling an optional feature, the element providing the feature remains in the connector architecture but is excluded from a call path passing through its ports. As described in Section 4, when an element wants to be excluded from

the call chain, it provides target reference associated with its required port as a reference to its provided port. This constitutes an internal dependency between the ports which the reconfiguration process needs to be able to traverse. Since this dependency is not allowed between all port types, we enumerate the allowed cases and introduce the notion of a *pass-through port*.

An element port is considered pass-through, iff one of the following holds:

1. the port is provided and is associated with exactly one required port of the same element; if the reference to the port is looked up, the target of the associated required port is returned instead,
2. the port is provided and is associated with exactly one remote client port of the same element; if the reference to the port is looked up, a single target from the reference bundle of the associated remote port is returned instead,
3. the port is remote server and is associated with one or more required ports of the same element, and if the reference to the port is looked up, the returned reference bundle contains also the targets of the associated required ports.

Pass-through target is a target object of a required port that is associated with a pass-through port and to which the invocations on the pass-through port are delegated.

The reconfiguration process can be initiated at any non-root entity of the connector architecture, whenever a connector element needs to change a reference to an object implementing its provided or remote server port. This can happen either when an element's required or remote client port associated with a pass-through port is bound to a new target, or when an element needs to change the reference in response to an external request.

The implementation of each element must be aware of the potential internal dependencies between the pass-through ports and their pass-through targets. Whenever a change occurs in an element that changes the internal dependencies, the element must initiate the reconfiguration process by notifying its parent entity about its provided and remote server ports that are no longer valid.

The implementation of the reconfiguration algorithm is spread among the entities of the connector architecture and it differs depending on the position of an entity in the connector hierarchy (root, node, and leaf entities). Bellow, we list the operations that implement the reconfiguration process. Due to space constraints, the pseudo-code of the operations along with additional comments can be found in [5].

Lookup Port. Serves for looking up a local reference to an object implementing a particular provided port.

Lookup Remote Port. Serves for looking up a bundle of remote references to objects providing entry-points to the implementation of a particular remote server port. Each reference in the bundle is associated with an access scheme.

Bind Port. Serves for binding a particular required port to a provided port. If there is a pass-through port dependent on the port being bound, reconfiguration

is initiated. Bind Port also checks for re-entrant invocations for the same ports to detect cyclic dependencies between pass-through ports.

Bind Remote Port. Serves for binding a particular remote client port to a remote server port. If there is a pass-through port dependent on the port being bound, reconfiguration is initiated.

Invalidate Port. Serves for invalidating a reference to an object implementing a particular provided port. Used by connector elements to indicate that the provided port should be queried for a new reference.

Invalidate Remote Port. Serves for invalidating a bundle of remote references to objects providing an entry point to the implementation of a particular remote server port. Used by connector elements to indicate that the remote server port should be queried for a new reference bundle.

Rebind Connector Units. Serves for establishing a remote binding – gathers reference from remote ports with server functionality (using Lookup Remote Port) and distributes the references to remote ports with client functionality (using Bind Remote Port).

Bind Component Interface. Serves for binding a particular component interface to a provided connector element port implementing the interface. Serviced by a method specific to a particular component-model.

5.2 Algorithm Termination

Given the recursive nature of the algorithm, an obvious question is whether it always terminates. We show that the answer is yes, even when an implementation of a connector architecture is invalid (i.e. there are cyclic dependencies between pass-through ports), in which case the algorithm detects the cycle and terminates. In proving that the algorithm always terminates, we will examine the reasons for the algorithm not to terminate and show that such situation cannot happen. We believe that the use of informal language does not affect the correctness of the proof.

As a requisite for the proof, we construct a call graph of the algorithm operations (see Figure 4). Implementation variants of the operations as well as invocations on different instances are not distinguished. This simplifies reasoning, because it abstracts away from unimportant details. The cycles in the graph mark the problem places that could prevent the algorithm from terminating. Even though the graph in Figure 4 does not reflect the algorithm as accurately as the more detailed graphs would, the structure of the cycles is preserved, therefore the graph is adequate for the purpose of the proof.

The first step is to analyze the trivial cycles associated with the nodes labeled LP, LRP, BP, BRP, IP, and IRP. In case of the LP, LRP, BP, and BRP nodes, these cycles correspond to delegation of the respective Lookup Port, Lookup Remote Port, Bind Port, and Bind Remote Port operations from the parent elements to the child elements, down the connector architecture hierarchy. The recursion of these operations is limited by the depth of the connector architecture

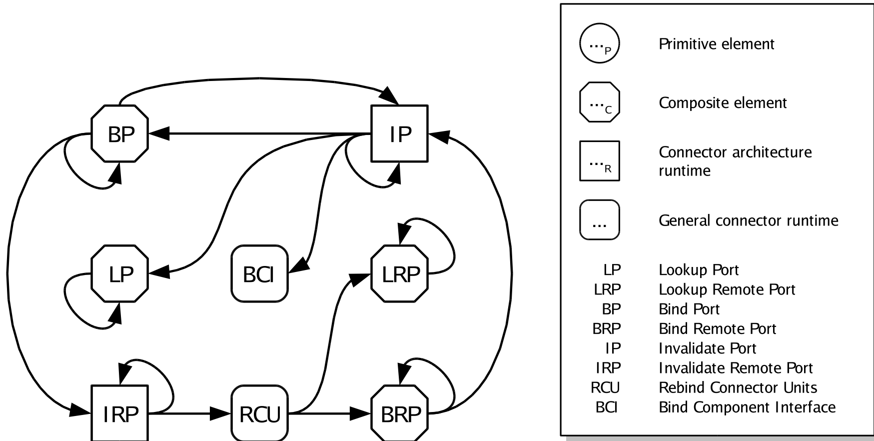


Fig. 4. Static call graph of the reconfiguration algorithm operations

hierarchy, because eventually the methods must reach a primitive element, which is a leaf entity of the connector architecture. In case of the IP and IRP nodes, the cycles correspond to the propagation of the Invalidate Port and Invalidate Remote Port operations from a connector element to its parent entity (another connector element or connector runtime). The recursion is again limited by the depth of the connector architecture hierarchy, but in this case these operations must eventually reach to root entity (connector runtime).

The cycles associated with the BP, BRP, IP, and IRP nodes can be also part of other, non-trivial, cycles. One class of them can be derived from the IP-BP cycle, which corresponds to the distribution of a new port reference to other elements. The Bind Port operation traverses the connector architecture downwards, while the Invalidate Port operation upwards. Because the number of required ports in a connector element is finite, then if there is a cyclic dependency between pass-through ports, the Bind Port operation will inevitably get called again for the same port before the previous call finished. Since the Bind Port operation has to be part of every such cycle, the operation is guarded against re-entrant invocation for the same port using a per port flag indicating that reconfiguration is already in progress for the port. The use of the flag corresponds to marking of the path during traversal to detect cycles. Therefore if a Bind Port operation finds the flag set already set in the port it was called for, there must be a cyclic dependency between pass-through ports and the algorithm is terminated.

The other non-trivial class of cycles can be derived from the IP-BP-IRP-RCU-BRP cycle. These cycles can be longer and more complex than in the previous case, but it is bounded by from the same reason as above – the bind port operation is guarded against re-entrant invocations and thus this cycle is also bounded by the number of required ports in a connector.

Because no other cycles are possible in the call graph, the algorithm always terminates in finite number of steps because the connector architecture is

finite. In case of invalid implementation of a connector architecture, the algorithm is terminated prematurely when a cycle between pass-through ports is detected.

6 Evaluation and Related Work

The approach presented in this paper is best related to other work in the context of its intended application. Including optional features in connectors can be related to instrumenting applications with code that is not directly related to their primary function. There are various instrumentation techniques, distinguished by the moment of instrumentation, the transparency of its usage, whether the instrumentation is performed on application source or binary, etc.

Eliminating the overhead associated with disabled optional features allows us to always include these features even in production-level applications, where they only impact the execution of an application when they are used. The most prominent examples of such features are are logging, tracing, or performance and behavior monitoring. With respect to tracing, our approach can be used for dynamic tracing of component-based applications, which would provide functionality similar to that of DTrace [6]. While DTrace provides tracing mainly on the level of system and library calls, our approach is targeted at tracing at the level of design-level elements, such as components.

Similar relation can be found in performance evaluation of component-based and distributed applications, where the original applications are instrumented with code for collecting performance data. Since the code is orthogonal to the primary function of the applications, the instrumentation can take place as late as during deployment or just before execution. Such functionality is provided by the COMPAS framework [7] by Adrian Mos, which instruments an EJB application with probes that can report performance information during execution of the application. The operational status of the probes can be controlled at runtime, but even when disabled, the static overhead of the instrumentation is still present. However, since the instrumentation mechanism serves only a single purpose (it is not designed to allow combining multiple features in contrast to the compositional approach in case of connector architecture), it potentially adds only a single level of indirection and its overhead is therefore negligible.

When evaluating performance of CCA [8] applications using the TAU [9] tools for CCA, a proxy component must be generated for each component that should be included in the evaluation. The integration of these into the original application requires modification of the architecture description to redirect the original bindings to the proxy components. The proxy components, even when inactive, still contribute certain overhead to the execution of the instrumented application. Using connectors for the same purpose would allow the instrumentation to be completely transparent to the component application, and the static overhead of the proxy components could be eliminated as well.

Similar goals, i.e. performance monitoring of distributed CORBA-based applications are pursued in [10] and in the WABASH [11] tool. The former approach uses BOA inheritance and TIE classes to add instrumentation code, while WABASH uses CORBA interceptors to achieve the same. In both cases, the instrumentation mechanism does not incur significant performance overhead, but requires source code of the application to perform the instrumentation, the availability of which cannot be always assured.

From a certain point of view, the changes performed in connector runtime could be seen as dynamic reconfiguration of middleware, and therefore be related to reflective middleware by Blair et al. [2]. However, the relation is only marginal, because the dynamic reconfiguration of middleware is a more general and consequently more difficult problem to solve. What makes our approach feasible is that the architecture of a connector in fact remains unchanged and that the reconfiguration is initiated by a connector element from inside of the architecture and only requires local information in each step.

7 Conclusion

In this paper, we have presented an approach to elimination of static execution overhead associated with disabled optional features in a particular model of architecture-based connectors. The approach is based on structural reconfiguration of runtime entities implementing a connector and utilizes runtime information on connector architecture to derive dependencies among the entities implementing a connector.

Even though the reconfiguration process which forms the basis of the approach has been presented on a specific connector model, the approach can be used in similarly structured environments, such as simple component models and componentized middleware, if the necessary information and facilities for manipulating implementation entities are available.

Apart from atomicity of a reference assignment, the reconfiguration algorithm makes no other technical assumptions and requires no operations specific to any particular programming language or platform. This makes it well suitable for use in heterogeneous execution environments.

While the reconfiguration algorithm itself would not be difficult to implement, for routine use in connectors [4] the implementation of the algorithm has to be generated. This requires implementing a generator of the algorithm implementation and integrating it with a prototype connector generator. This project is currently under way, but no case study with connectors utilizing the algorithm is available at the moment.

The algorithm undoubtedly improves the efficiency of a connector by eliminating unnecessary indirections, but the improvement has not yet been experimentally evaluated and it remains to be seen how many eliminated indirections are required to witness a non-trivial improvement. In case of features such as logging or monitoring, the main achievement is that such features can be included even in production-level applications without impact on normal execution.

More promising is the application of the algorithm during initial configuration of a connector which takes place at application startup. This may result in elimination of all connector code (mainly stub and skeleton) from the call path between locally connected components.

Acknowledgement. This work was partially supported by the Academy of Sciences of the Czech Republic project 1ET400300504 and by the Ministry of Education of the Czech Republic grant MSM0021620838.

References

1. Cantrill, B.: Hidden in plain sight. ACM Queue **4**(1) (2006) 26–36
2. Blair, G.S., Coulson, G., Grace, P.: Research directions in reflective middleware: the Lancaster experience. In: Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, RM 2004, Toronto, Canada, ACM (2004) 262–267
3. Dumant, B., Horn, F., Tran, F.D., Stefani, J.B.: Jonathan: an open distributed processing environment in Java. Distributed Systems Engineering **6**(1) (1999) 3–12
4. Bures, T., Plasil, F.: Communication style driven connector configurations. In Software Engineering Research and Applications: First International Conference, SERA 2003, San Francisco, USA, LNCS 3026, Springer (2004) 102–116
5. Bulej, L., Bures, T.: Addressing static execution overhead in connectors with disabled optional features. Tech. Report 2006/6, Dept. of SW Engineering, Charles University, Prague (2006)
6. Cantrill, B., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the General Track: 2004 USENIX Annual Technical Conference, 2004, Boston, USA, USENIX (2004) 15–28
7. Mos, A., Murphy, J.: COMPAS: Adaptive performance monitoring of component-based systems. In: Proceedings of the 2nd International Workshop on Remote Analysis and Measurement Software Systems, RAMSS 2004, Edinburgh, UK, IEE Press (2004) 35–40
8. Malony, A.D., Shende, S., Trebon, N., Ray, J., Armstrong, R.C., Rasmussen, C.E., Sottile, M.J.: Performance technology for parallel and distributed component software. Concurrency and Computation: Practice and Experience **17**(2-4) (2005) 117–141
9. Malony, A.D., Shende, S.: Performance technology for complex parallel and distributed systems. In: Proceedings of the 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000, Balatonfured, Hungary, Kluwer Academic Publishers (2000) 37–46
10. McGregor, J.D., Cho, I.H., Malloy, B.A., Curry, E.L., Hobatr, C.: Collecting metrics for CORBA-based distributed systems. Empirical Software Engineering **4**(3) (1999) 217–240
11. Sridharan, B., Mathur, A.P., Dasarathy, B.: On building non-intrusive performance instrumentation blocks for corba-based distributed systems. In: Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium, IPDS 2000, Chicago, USA, IEEE Computer Society (2000) 139–143.

Automated Selection of Software Components Based on Cost/Reliability Tradeoff*

Vittorio Cortellessa[1], Fabrizio Marinelli[1], and Pasqualina Potena[2]

[1] Dipartimento di Informatica
Università dell'Aquila
Via Vetoio, 1, Coppito (AQ), 67010 Italy
{cortelle,marinelli}@di.univaq.it
[2] Dipartimento di Scienze
Università "G.D'Annunzio"
Viale Pindaro, 42, Pescara, 65127 Italy
potena@sci.unich.it

Abstract. Functional criteria often drive the component selection in the assembly of a software system. Minimal distance strategies are frequently adopted to select the components that require minimal adaptation effort. This type of approach hides to developers the non-functional characteristics of components, although they may play a crucial role to meet the system specifications. In this paper we introduce the CODER framework, based on an optimization model, that supports "build-or-buy" decisions in selecting components. The selection criterion is based on cost minimization of the whole assembly subject to constraints on system reliability and delivery time. The CODER framework is composed by: an UML case tool, a model builder, and a model solver. The output of CODER indicates the components to buy and the ones to build, and the amount of testing to be performed on the latter in order to achieve the desired level of reliability.

1 Introduction

When the design of a software architecture reaches a good level of maturity, software engineers have to undertake selection decisions about software components. COTS have deeply changed the approach to software design and implementation. A software system is ever more rarely built "from scratch", as part of the system comes from buying/reusing existing components.

Even though in the last years numerous tools have been introduced to support decisions in different phases of the software lifecycle, the selection of the appropriate set of components remains a hard task to accomplish, very often left to the developers' experience. Without the support of automation, the selection is frequently driven from functional criteria related to the distance of the characteristics of available components from those specified in the architectural description. This is due to the deep understanding that software designers have developed on functional issues, as well as to the

* This work has been partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. EC - 6th Framework Programme. http://www.ist-plastic.org

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 66–81, 2006.
© Springer-Verlag Berlin Heidelberg 2006

introduction of sophisticated compositional operators (e.g. connectors with complex internal logics) that help to assembly systems satisfying the functional requirements.

As opposite, limited contributions have been brought to support the selection of components on the basis of their non-functional characteristics. As a consequence, software developers have no automated tools to support the analysis “aimed at characterizing the performance and reliability behavior of software applications based on the behavior of the “components” and the “architecture” of the application” [6]. This analysis might be used to answer questions such as: (i) which components are critical to the performance and reliability of the application? and (ii) how are the application performance and reliability influenced by the performance and reliabilities of individual components? If the software application is to be assembled from a collection of components, then answer to such questions can help the designers to make decisions such as which components should be picked off the shelf, and which components should be developed in-house [6].

A recent empirical study on COTS-based software development [17] shows that component selection is part of new activities integrating the traditional development process, and it is always based on the experience of project members. The same study evidences a similar practice that bases the COTS component selection either on the developer familiarity or on license issues and vendor reputation. None of the studied projects uses decision-making algorithms.

Beside all the above considerations, real software projects ever more suffer from limited budgets, and the decisions taken from software developers are heavily affected by cost issues. The best solutions might not be feasible due to high costs, and wrong cost estimations may have a critical impact for the project success. Therefore tools that support decisions strictly related to meet functional and non-functional requirements, while keeping the costs within a predicted budget, would be very helpful to the software developer’s tasks.

In this paper we introduce CODER (Cost Optimization under DElivery and Reliability constraints), a framework that helps developers to decide whether buying or building components of a certain software architecture. Once built a software architecture, each component can be either bought, and probably adapted to the new software system, or it can be developed in-house. This is a “build-or-buy” decision that affects the software cost as well as the ability of the system to meet its requirements.

CODER supports the component selection basing on cost, delivery time and reliability characteristics of the components. We assume that several instances of each software component may be available as COTS. Basically, the instances differ with respect to cost, reliability and delivery time. Besides, we assume that several in-house instances of each software component may be built. In fact, the developers of a system could build an in-house component by adopting different strategies of development. Therefore, the values of cost, reliability and delivery time of an in-house developed component could vary due to the values of the development process parameters (e.g. experience and skills of the developing team). CODER indicates the assembly of (in-house and COTS) components that minimizes the cost under constraints on delivery time and reliability of the whole system. In addition, for each in-house developed component CODER suggests the amount of testing to perform in order to achieve the required level of reliability.

The paper is organized as follows: in section 2 we provide the formulation of the optimization model that represents the CODER core; in section 3 we introduce the CODER structure and underlying mechanisms; in section 4 we illustrate the usage of CODER in the development of a mobile application; in section 5 we summarize recent work in software cost estimation vs. quality attributes and outline the novelty of our approach with respect to the existing literature; conclusions are presented in section 6. In [5] we have collected all the details that are not strictly necessary for this paper understanding.

2 The Optimization Model Formulation

In this section, we introduce the mathematical formulation of the optimization model that CODER generates and solves, and that represents the core of our approach ([1]).

Since our framework may support different lifecycle phases, we adopt a general definition of component: a component is a self-contained deployable software module containing data and operations, which provides/requires services to/from other components. A component instance is a specific implementation of a component.

The solution of the optimization model determines the instance to choose for each component (either one of the available COTS products or an in-house developed one) in order to minimize the software costs under the delivery time and reliability constraints. Obviously when no COTS products are available the in-house development of a component is a mandatory decision whatever being the cost incurred. Viceversa for components that cannot be in-house built (e.g. for lack of expertise) one of the available COTS products must be chosen.

Due to our additional decision variables, the model solution also provides the amount of testing to be performed on each in-house component in order to achieve a certain reliability level.

2.1 The Problem Formulation

Let S be a software architecture made of n components. Let J_i ($\bar{J}_i$) be the set of COTS (in-house developed) instances available for the i-th component, and $m = \max_i |J_i \cup \bar{J}_i|$.

Let us suppose to be committed to assemble the system by the time T while ensuring a minimum reliability level R and spending the minimum amount of money.

COTS Component Model Parameters

The parameters that we define for a COTS product $C_{ij} \in J_i$ are:

- the cost c_{ij};
- the delivery time d_{ij};
- the average number s_i of invocations;
- the probability μ_{ij} of failure on demand.

[1] For sake of readability, we report model details in [5], thus we ask readers interested to a deeper understanding of the model construction to refer to [5].

The estimate of the cost c_{ij} is outside the scope of this paper, however, the following expression can be used to estimate it:

$$c_{ij} = c_{ij}^{buy} + c_{ij}^{adapt}$$

where c_{ij}^{buy} is the purchase cost, and c_{ij}^{adapt} is the adaptation cost. The adaptation cost takes into account the fact that, in order to integrate a software component into a system, the component must support the style of the interactions of the system's architecture to correctly work together with other components. If a COTS product has another style of interaction, developers have to introduce glueware to allow correct interactions.

Yakimovich et al. in [24] suggest a procedure for estimating the adaptation cost. They list some architectural styles and outline their features with respect to a set of architectural assumptions. They define a vector of variables, namely the interaction vector, where each variable represents a certain assumption. An interaction vector can be associated either to a single COTS or to a whole software architecture. To estimate the adaptation cost of a COTS they suggest to compare its interaction vector with the software architecture one.

Furthermore, c_{ij}^{adapt} could include the cost needed to handle mismatches between the functionalities offered by alternative COSTs and the functional requirements of the system. In fact, it may be necessary to perform a careful balancing between requirements and COTS features, as claimed in [2].

The purchase cost c_{ij}^{buy} is typically provided by the vendor of the COTS component.

The delivery time d_{ij} might be decomposed in the sum of the time needed to the vendor to deliver the component, and the adaptation time.

For sake of model formulation, in this paper we do not explicitly preserve the above decompositions of cost and delivery time parameters, although we implicitly take into account them in the example of Section 4.

The parameter s_i represents the average number of invocations of a component within the execution scenarios considered for the software architecture. Note that this value does not depend on the component instance, because we assume that the pattern of interactions within each scenario does not change by changing the component instance. This value is obtained by processing the execution scenarios that, in the CODER framework, are represented by UML Sequence Diagrams (see Section 3). The number of invocations is averaged overall the scenarios by using the probability of each scenario to be executed. The latter is part of the operational profile of the application.

The parameter μ_{ij} represents the probability for the instance j of component i to fail in one execution [20]. A rough upper bound $1/N_{nf}$ of μ_{ij} can be obtained upon observing the component being executed for a N_{nf} number of times with no failures. However, several empirical methods to estimate COTS failure rates can be found in [17].

In-House Component Model Parameters

The parameters that we define for an in-house developed instance $C_{ij} \in \bar{J}_i$ are:

- the unitary development cost $\bar{c}_{ij}$;
- the estimated development time t_{ij};

- the average time τ_{ij} required to perform a test case;
- the average number s_i of invocations;
- the probability p_{ij} that the instance is faulty;
- the testability $Testab_{ij}$.

The unitary cost $\bar{c}_{ij}$ is intended as the per-day cost of a software developer, that may depend on the skills and experience required to develop C_{ij}. Well-assessed cost/time models are available to estimate the first three parameters (e.g. COCOMO [4]).

The parameter p_{ij} is an intrinsic property of the instance that depends on its internal complexity. The more complex the internal dynamics of the component instance is, the higher is the probability that a bug has been introduced during its development. In [18] an expression for p_{ij} has been proposed as a function of the component instance internal reachability.

The definition of testability that we adopt in our approach is the one given in [21], that is:

$$Testab_{ij} = P(failure|prob.\ distribution\ \ of\ \ inputs) \tag{1}$$

Their definition of testability expresses the conditional probability that a single execution of a software fails on a test case following a certain input distribution. In [5] we suggest a procedure to estimate it.

Model Variables. In general, a "build-or-buy" decisional strategy can be described as a set of 0/1 variables defined as follows ($\forall i = 1 \ldots n$):

$$x_{ij} = \begin{cases} 1 \text{ if the } C_{ij} \text{ instance is chosen } (j \in \bar{J}_i \text{ or } j \in J_i) \\ 0 \text{ otherwise} \end{cases}$$

Obviously, if the i-th component has only $\bar{m} < m$ instances then the x_{ij}'s are defined for $1 \le j \le \bar{m}$.

For each component i, exactly one instance is either bought as COTS or in-house developed. The following equation represents this constraint:

$$\sum_{j \in J_i \cup \bar{J}_i} x_{ij} = 1, \qquad \forall i = 1 \ldots n \tag{2}$$

Finally, let N_{ij}^{tot} be an additional integer decision variable of the optimization model that represents the total number of tests performed on the in-house developed instance j of the i-th component ([2]).

Basing on the testability definition, we can assume that the number N_{ij}^{suc} of successful (i.e. failure-free) tests performed on the same component can be obtained as:

$$N_{ij}^{suc} = (1 - Testab_{ij}) N_{ij}^{tot}, \qquad \forall i = 1 \ldots n, j \in \bar{J}_i \tag{3}$$

[2] The effect of testing on cost, reliability and delivery time of COTS products is instead assumed to be accounted in the COTS parameters.

Cost Objective Function (COF). The development cost of the in-house instance C_{ij} can be expressed as: $\bar{c}_{ij}(t_{ij} + \tau_{ij}N_{ij}^{tot})$. The objective function to be minimized, as the sum of the costs of all the component instances selected from the "build-or-buy" strategy, is given by:

$$COF = \sum_{i=1}^{n} \left(\sum_{j \in \bar{J}_i} \bar{c}_{ij}(t_{ij} + \tau_{ij}N_{ij}^{tot})x_{ij} + \sum_{j \in J_i} c_{ij}x_{ij} \right) \tag{4}$$

Delivery Time Constraint (DT). A maximum threshold T has been given on the delivery time of the whole system. In case of a COTS product the delivery time is given by d_{ij}, whereas for an in-house developed instance C_{ij} the delivery time shall be expressed as $t_{ij} + \tau_{ij}N_{ij}^{tot}$. Therefore the following expression represents the delivery time DT_i of the component i:

$$DT_i = \sum_{j \in \bar{J}_i} (t_{ij} + \tau_{ij}N_{ij}^{tot})x_{ij} + \sum_{j \in J_i} d_{ij}x_{ij} \tag{5}$$

Without loss of generality, we assume that sufficient manpower is available to independently develop in-house component instances. Therefore the delivery constraint can be reformulated as follows:

$$\max_{i=1\ldots n} (DT_i) \leq T \tag{6}$$

which can be decomposed in the set of constraints $DT_1 \leq T, \ldots, DT_n \leq T$.

Reliability Constraint (REL). We consider systems that may incur only in crash failures, that are failures that (immediately and irreversibly) compromise the behaviour of the whole system ([3]).

A minimum threshold R has been given on the reliability on demand [20] of the whole system. The reliability of the whole system can be obtained as a function of the probability of failure on demand of its components, as we show in this section.

The probability of failure on demand $\mu_{ij}, j \in J_i$, for COTS components has been discussed in section 2.1.

The probability of failure on demand θ_{ij} of the in-house developed instance $C_{ij}, j \in \bar{J}_i$, can be formulated as follows:

$$\theta_{ij} = \frac{Testab_{ij} \cdot p_{ij}(1 - Testab_{ij})^{N_{ij}^{suc}}}{(1 - p_{ij}) + p_{ij}(1 - Testab_{ij})^{N_{ij}^{suc}}} \tag{7}$$

A proof of this formulation is given in [5].

Now we can write the average number of failures $fnum_i$ of the component i as follows:

$$fnum_i = \sum_{j \in \bar{J}_i} \theta_{ij}s_i x_{ij} + \sum_{j \in J_i} \mu_{ij}s_i x_{ij} \tag{8}$$

[3] Note that, although promising formulations of the component capability of propagating errors have been devised (see for example [1]), no closed form expression for system reliability embedding error propagation has yet been found.

In agreement with [11], the probability that no failure occurs during the execution of the i-th component is given by $\phi_i = e^{-fnum_i}$, which represents the probability of no failures occurring in a Poisson distribution with parameter $fnum_i$.

Therefore the probability of a failure-free execution of the system is given by $\prod_{i=1}^{n} \phi_i$. The reliability constraint is then given by:

$$\prod_{i=1}^{n} \phi_i \geq R \tag{9}$$

Model Summary. The objective function (4), under the main constraints (2), (6) and (9), plus the obvious integrality and non-negativity constraints on the model variables, represent the optimization model adopted within the CODER framework.

The model solution provides the optimal "build-or-buy" strategy for component selection, as well as the number of tests to be performed on each in-house developed component. The solution guarantees a system reliability on demand over the threshold R, a system delivery time under the threshold T while minimizing the whole system cost. The applied reliability model is a light-weighted one, as we work in favor of model solvability. However, it can be replaced by a profound reliability growth model from literature [7] to increase the result accuracy. This can be done without essentially changing the overall model structure, with the side effect of increasing complexity. In [5] we report the mathematical formulation of the whole model.

With regard to the accuracy of the model, there are some input parameters (e.g. the probability of failure on demand, the cost) that may be characterized by a not negligible uncertainty (i.e. only a range for the costs may be available [14]). The propagation of this uncertainty should be analyzed, but it is outside the scope of this paper. However, several methods to perform this type of analysis can be found, e.g. it has been done in [8] for a reliability model.

3 The CODER Framework

In Figure 1 the CODER framework is shown within its working environment.

The input to the framework is an UML model constituted by: (i) a Component Diagram representing the software architecture, (ii) a set of Sequence Diagrams representing the possible execution scenarios.

CODER accepts UML models in XMI format [25]. In theory, any tool exporting diagrams to XMI can be used to generate input models for CODER. In practice this is not the case because XMI exporting formats may sensibly differ from each other. For this paper example (in Section 4) we have used ArgoUML [26] to build and export UML diagrams.

The CODER framework is made of two components, which are a model builder and a model solver.

The model builder first allows users to annotate the UML diagrams with additional data that represent the optimization model parameters (see Section 2), such as failure probabilities of software components. Then it transforms the annotated model into an optimization model in the format accepted from the solver.

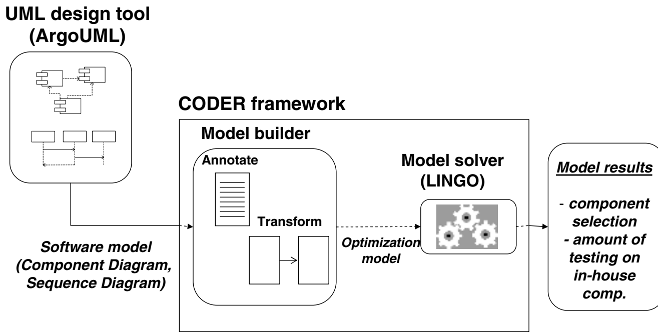


Fig. 1. The CODER framework and its environment

The model solver processes the optimization model received from the builder and produces the results, that consist in the selection of components and the amount of testing to perform on in-house components.

The optimization model solver that we have adopted in CODER is LINGO [27]. The integration between the model builder and the model solver has been achieved as follows. LINGO makes use of a callable Dynamic Link Library (DLL) to provide a way to bundle its functionalities into a custom application. In particular, the DLL gives the ability to run a command script containing an optimization model and a series of commands that allows to gather data, to populate and solve the model. The integration of data between the calling application (i.e. the model builder in our case) and the solver can be obtained by means of the *Pointer* functions in the data section of the script. These functions act as a direct memory links and permit direct and fast memory transfers of data in and out of the solver memory area.

As a result of this integration, LINGO can be directly run from the main interface of the model builder, as shown in Figure 2. The main interface can be partitioned in 3 areas: (i) the working area (upper right side of Figure 2), where the imported UML diagrams are shown, and where components and lifelines can be selected for annotations; (ii) the annotation area, where the model parameters related to software components can be entered (lower side of Figure 2); (iii) the model constraint area, where values of model constraint bounds can be assigned (upper left side of Figure 2). The four ellipses of Figure 2 highlight, respectively, from the top to the bottom of the figure: the button to run the model solver LINGO, the title of the area where constraint bounds can be entered and the titles of areas where COTS and in-house component parameters can be entered.

Summing up, CODER allows to specify ranges for model parameters and sets of alternative optimization models can be automatically generated (by sampling parameters in the given ranges) and solved. The output of CODER, for each model, is a suggested selection of available components and the suggested amount of testing to perform on each in-house developed component. In the next section, we apply the model to an example.

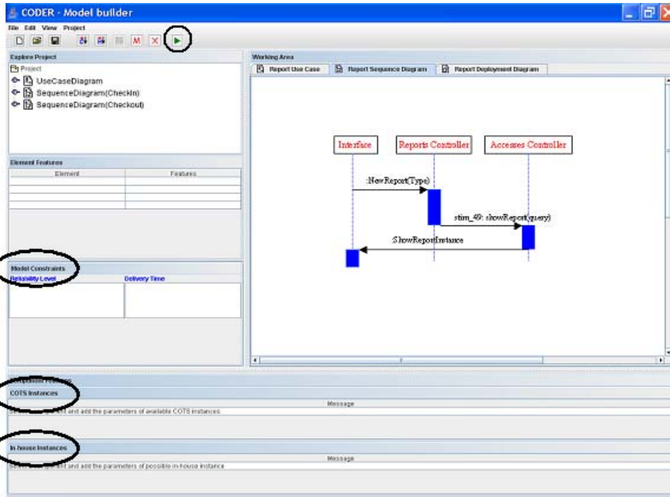


Fig. 2. A screenshot of the CODER model builder

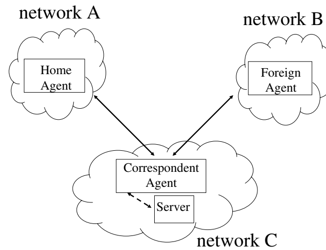


Fig. 3. The example architecture

4 Using CODER in the Development of a Mobile Application

We have considered an application that allows the mobility of a user without loosing its network connection based on Mobile IP [19].

Figure 3 shows the architecture of the application. A user of the network A can exchange data with users of the network C through a server located in C. A user of network A can also move to network B and continue to interact with a user of network C without generating a new connection. Four software components are deployed: *Home Agent* (running on network A), *Foreign Agent* (running on network B), *Correspond Agent* and *Server* (running on network C).

The scenario that we consider can be described as follows: interactions between users in A and users in C change only when a user moves from A to B; the effect of this move is that *Foreign Agent* provides the user's new address to *Home Agent*; as soon as users in C attempt to interact with the moving user through her/his old address in

Table 1. First Configuration : parameters for COTS products

	Component name	COTS alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_0	*Correspond Agent*	C_{01}	12	4	200	0.0005
		C_{02}	14	3		0.00015
		C_{03}	15	3		0.0001
C_1	*Server*	C_{11}	6	4	40	0.0003
		C_{12}	12	3		0.0001
C_2	*Home Agent*	C_{21}	12	2	80	0.00015
C_3	*Foreign Agent*	C_{31}	7	4	25	0.0002
		C_{32}	10	3		0.00015
		C_{33}	8	7		0.00015

Table 2. First Configuration : parameters for in-house development of components

	Component name	Development Time t_{i0}	Testing Time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Faulty Probability p_{i0}	Testability $Testab_{i0}$
C_0	*Correspond Agent*	10	0.007	1	200	0.03	0.00001
C_1	Server	5	0.007	1	40	0.01	0.001
C_2	*Home Agent*	6	0.007	1	80	0.04	0.001
C_3	*Foreign Agent*	5	0.007	1	25	0.05	0.002

A, *Home Agent* provides the user's new address to *Correspond Agent* so that, without interruption, users in C switch their interactions towards network B [19].

We show the support that the CODER framework can provide to select components during the development of this application. We apply our approach on two different configurations. In order to keep our model as simple as possible, in both configurations we assume that only one in-house instance for each component can be developed.

The number of COTS instances does not change across configurations, but each configuration is based on a different set of component parameters. We have solved the optimization model in both configurations for a set of values of reliability and delivery time bounds.

4.1 First Configuration

Table 1 shows the parameter values for the COTS available instances, likewise Table 2 does for in-house developed ones, where $\bar{J}_i = \{0\}(i = 0, ..., 3)$, $J_0 = \{1, 2, 3\}$, $J_1 = \{1, 2\}$, $J_2 = \{1\}$ and $J_3 = \{1, 2, 3\}$.

The third column of Table 1 lists, for each component, the set of COTS alternatives available at the time of system development. For each alternative: the buying cost c_{ij} (in KiloEuros, KE) is given in the fourth column, the average delivery time d_{ij} (in days) is given in the fifth column, the average number of invocations of the component in the system s_i is given in the sixth column, finally the probability of failure on demand μ_{ij} is given in the seventh column.

For each component in Table 2: the average development time t_{i0} (in days) is given in the third column and the average time required to perform a single test τ_{i0} (in days) is given in the fourth column, the unitary development cost $\bar{c}_{i0}$ (in KE per day) is given in the fifth column, the average number of invocations s_i is given in the sixth column,

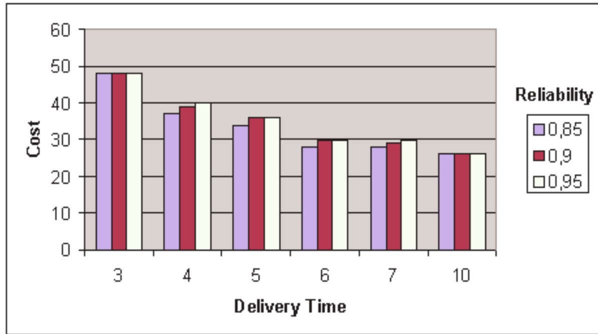


Fig. 4. Model solutions for first configuration

the probability p_{i0} that the component instance is faulty at the execution time is given in the seventh column, and finally the component testability $Testab_{i0}$ is given in the last column.

Note that each component can be in-house built. This configuration is characterized by the fact that the in-house instance of each component is less reliable than all the COTS available components, but it is less expensive than all the latter ones.

In Figure 4 we report the results obtained from solving the optimization model for multiple values of bounds T and R. Each bar represents the minimum cost for a given value of the delivery time bound T and a given value of the reliability bound R. The former spans from 3 to 10 whereas the latter from 0.85 to 0.95.

As expected, for the same value of the reliability bound R, the total cost of the application decreases while increasing the delivery time bound T (i.e. more time to achieve the same goal). On the other hand, for the same value of T the total cost almost always decreases while decreasing the reliability bound R (i.e. less reliable application required) and, in two cases, it does not increase.

With regard to the component selection: for $T < 10$, the model solution proposes different combinations of COTS and in-house instances almost always without test on the latter ones; for $T = 10$, the model proposes, for all R values, the same solution made of all in-house instances without test. In [5] we report the solution vectors for each pair of bounds (T,R).

4.2 Second Configuration

Similarly to the first configuration, Table 3 shows the parameter values for the COTS available components, whereas Table 4 shows the ones for the in-house instances.

Again note that each component can be in-house built. The component parameters in this configuration have been set to induce a certain amount of testing on in-house instances. In particular: the in-house instance of C_0 is less reliable, but earlier available and less expensive than all the available COTS instances for this component; the C_1 and C_2 in-house instances are less reliable than all the corresponding COTS available instances, but are less expensive than these last ones; the in-house instance of C_3 is

Table 3. Second Configuration : parameters for COTS products

	Component name	COTS alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_0	*Correspond Agent*	C_{01}	12	4	200	0.00015
		C_{02}	14	3		0.00015
		C_{03}	15	3		0.00001
C_1	*Server*	C_{11}	18	4	40	0.0001
		C_{12}	18	3		0.00003
C_2	*Home Agent*	C_{21}	15	2	80	0.00001
C_3	*Foreign Agent*	C_{31}	9	4	25	0.0002
		C_{32}	14	3		0.00015
		C_{33}	9	7		0.00002

Table 4. Second Configuration: parameters for in-house development of components

	Component name	Development Time t_{i0}	Testing Time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Faulty Probability p_{i0}	Testability $Testab_{i0}$
C_0	*Correspond Agent*	1	0.007	1	200	0.08	0.008
C_1	Server	10	0.007	1	40	0.08	0.009
C_2	*Home Agent*	10	0.007	1	80	0.08	0.007
C_3	*Foreign Agent*	6	0.007	1	25	0.05	0.004

as reliable as (but more expensive than) the first COTS instance, whereas all the other COTS instances are more reliable than it.

In Figure 5 we report again the results obtained from solving the optimization model for multiple values of bounds T and R. Here the former spans from 3 to 15 whereas the latter from 0.90 to 0.98. In [5] we report the solution vectors for each pair of bounds (T,R).

Similarly to the first configuration, for the same value of the reliability bound R, the total cost of the application decreases while increasing the delivery time bound T. On the other hand, for the same value of T the total cost decreases while decreasing the reliability bound R.

As shown in [5], the component selection for this configuration is more various. While T increases, the model tends to select in-house components because they are cheaper than the available COTS instances. This phenomenon can be observed even for low values of T. The total cost decreases while T increases because ever more in-house instances can be embedded into the solution vectors. The in-house instances remain cheaper than the corresponding COTS instances even in cases where a non negligible amount of testing is necessary to make them more reliable with respect to the available COTS.

5 Related Work

The correlation between costs and non-functional attributes of software systems has always been of high interest in the software development community. After a phase of experimental assessment, in the last years new methodologies and tools have been introduced to systematically model and evaluate issues related to this aspect from the architectural phase.

Fig. 5. Model solutions for second configuration

The Architecture Tradeoff Analysis Method (ATAM) [13] provides to software developers a framework to reason about the software tradeoffs at the architectural level. The Attribute-based Architectural Styles (ABAS) [16], used within ATAM, help software architects to reason (quantitatively and qualitatively) about the quality attributes and the stimulus/response characteristics of the system. ATAM/ABAS framework is based on roughly approximated cost-characteristic curves, usually elicited from experience, that show how costs will behave with respect to each architectural decision. Our context differs from ATAM/ABAS because it is model-based, as opposed to experience-based, and the model we propose focuses on component selection decisions.

A significant breakthrough in this area has been the Costs Benefit Analysis Method (CBAM) [14]. CBAM, laying on the artifacts produced from ATAM, estimates costs, (short-term and long-term) benefits and uncertainty of every potentially problematic architectural design decision devised from ATAM. The estimates come out from information collected from stakeholders in a well assessed elicitation process. Architectural decisions are represented in a space whose dimensions are costs, benefits and (some measure of) uncertainty. The graphical representation of decisions is an excellent mean to support the developers' choices. Architectural strategies (ASs) typically have effects on several quality attributes (QAs). In order to evaluate the benefits of ASs on the whole software system, CBAM framework proposes that stakeholders assign contributions of ASs to QAs, and quality attribute score to QAs. The benefit of an AS is then computed as the sum of its contributions weighted on the QAs quality attribute scores. CBAM framework, however, deals neither with the elicitation of such contributions and scores nor with the assessment of ASs implementation costs. Actually, cost estimation often has to take into account some critical time-to-market goals such as delivery times and shared use of resources.

Although CBAM is a very promising technique to support software developers giving priorities among architectural decisions on the basis of their costs and benefits, it requires to stakeholders to estimate a large number of scores, contributions and costs by resorting to qualitative judgements based of their own expertise. In this context an analytical approach taking into account all architectural alternatives and tradeoffs among

qualitative attributes is extremely suitable. A key issue is to capture the relationships among costs and quality attributes, as well as across different quality attributes.

An optimization model may play the role of *decision support* in the early development phases, where the decisions are usually based on stakeholders' estimates. Later on, it can be an actual *decision-making* tool, when the software architecture and the bounds on the quality attributes have been devised, and implementation choices (such as resource allocation and amount of testing) may heavily affect costs and quality of the system. A classical approach for cost management is the portfolio-optimization, based on a knapsack-like integer linear programming model [15]. The models and techniques that we refer to in the remainder of this section follow this approach.

Optimization techniques appeared first in the area of software development in [10], where a variant of the 0-1 knapsack model is introduced to select the set of software requirements that yields the maximum value while minimizing costs. The concept of value is kept quite general and may be interpreted as an implementation priority. The knapsack model is first used to maximize the total value of requirements under budget constraints, thereafter to minimize the total cost without loosing requirement value.

The same authors in [12] introduce an interesting generalization of the model assumptions. The idea is that each COTS has a generic quality attribute, the objective function is the system quality as the weighted sum of COTS qualities, and the maximization is budget-constrained.

In the reliability domain, an interesting formulation of a cost minimization model has been given in [11]. Again 0-1 variables allow to select alternative COTS components, under a constraint on the failure rate of the whole system. The latter quantity is modeled as a combination of the failure rates of single components, their execution times, and a rough measure of the system workload. This is the closest model formulation to the one that we propose here.

In [22,23] the reliability constraints also cope with hardware failures, but the nonlinear complexities of the models impose heuristic solutions.

An extensive optimization analysis of the tradeoff between costs and reliability of component-based software systems has been presented in [9]. A reliability constrained cost minimization problem is formulated, where the decision variables represent the component failure intensities. Three different types of cost functions (i.e., linear, logarithmic exponential, inverse power) have been considered to represent the dependency of the component cost on the component failure intensity, that is the cost to attain a certain failure intensity. An exponential function has been used to model the system reliability as a combination of component failure intensities, operational profile (i.e. probability of component invocation) and time to execute the invoked service. The goal of this type of analysis is quite different from the one of this paper. The model in [9] works after the components have been chosen, as its solution provides insights about the failure intensities that the (selected) components have to attain to minimize the system cost.

The formulation of our model that we proposed in section 2 is close to the one in [11] with an additional constraint on the system delivery time. However, none of the existing approaches, supports "build-or-buy" decisions.

The following major aspects characterize the novelty of our approach:

- From an automation viewpoint, CODER is (at the best of our knowledge) the first thorough framework that supports a process of component selection based on cost, reliability and delivery time factors.
- CODER is not tied to any particular architectural style or to any particular component-based development process. Values of cost and reliability of in-house developed components can be based on parameters of the development process (e.g. a component cost may depend on a measure of developer skills).
- From a modeling viewpoint, we introduce decision variables that represent the amount of testing performed on each in-house component. The cost objective function, the reliability and delivery time constraints depend on these variables, therefore our model solution not only provides the optimal combination of COTS/in-house components, but also suggests the amount of testing to be performed on in-house components in order to attain the required reliability.

6 Conclusions

We have presented a framework supporting "build-or-buy" decisions in component selection based on cost, reliability and delivery time factors. The framework not only helps to select the best assembly of COTS components but also indicates the components that can be conveniently developed in-house. For the latter ones, the amount of testing to perform is also provided.

The integration of an UML tool (like ArgoUML), a model builder, and a model solver (like LINGO) has been quite easy to achieve due to XML interchange formats on one side, and to the Dynamic Link Library of LINGO on the other side. The CODER framework has been conceived to be easily usable from developers, and it indeed shows two crucial usability properties: transparency and automation. The software is annotated without modifying the original UML model, but producing a new annotated model, thus attaining transparency with respect to software modeling activities. Besides, the model building and solving is a completely automated tool supported process.

The results that we have obtained on the example shown in this paper provides evidence of the viability of such approach to the component selection. The components selected from the framework evidently constitute an optimal set under the existing constraints. It would be hard to obtain the same results without tool and modeling support. In addition, the tool also provides the amount of testing to perform, thus addressing the classical problem of: "How many tests are enough?" [18].

We are investigating the possibility of embedding in CODER other types of optimization models that may allow to minimize costs under different non-functional constraints (e.g. under security constraints). In general, these types of models are well suited to study the tradeoffs between different non-functional attributes, that are usually very hard to model and study in current (distributed, mobile) software systems. Furthermore, we intend to enhance CODER by introducing the multi-objective optimization [3] to provide the configuration of components that minimizes, for example, both the cost of construction of the system and its probability of failure on demand.

References

1. W. Abdelmoez et al., "Error Propagation in Software Architectures", *Proc. of METRICS*, 2004.
2. Alves, C. and Finkelstein, A. "Challenges in COTS decision-making: a goal-driven requirements engineering perspective", *Proc. of SEKE 2002*, 789-794, 2002.
3. Censor, Y., "Pareto Optimality in Multiobjective Problems",*Appl. Math. Optimiz.*, vol. 4, 41-59, 1977.
4. Boehm, B. "Software Engineering Economics", *Prentice-Hall*, 1981.
5. Cortellessa, V., Marinelli, F., Potena, P. "Appendix of the paper: Automated selection of software components based on cost/reliability tradeoff", *Technical Report*, Dip. Informatica, Università de L'Aquila, http://www.di.univaq.it/cortelle/docs/TECHNICALREPORT.pdf.
6. Gokhale, S.S., Wong, W.E, Horgan, J.R., Trivedi, K.S. "An analytical approach to architecture-based software performance and reliability prediction", *Performance Evaluation*, vol. 58 (2004), 391-412.
7. Goseva-Popstojanova, K. and Trivedi, K.S. "Architecture based-approach to reliability assessment of software systems" , *Performance Evaluation*, vol. 45 (2001),179-204.
8. Goseva-Popstojanova, K. and Kamavaram, S. "Uncertainty Analysis of Software Reliability Based on Method of Moments" , *FastAbstract ISSRE 2002*.
9. Helander, M.E., Zhao, M., Ohlsson, N. "Planning Models for Software Reliability and Cost", *IEEE Trans. in Software Engineering*, vol. 24, no. 6, June 1998.
10. Jung H.W. "Optimizing Value and Cost in Requirement Analysis", *IEEE Software*, July/August 1998, 74-78.
11. Jung H.W. et al. "Selecting Optimal COTS Products Considering Cost and Failure Rate", *Fast Abstracts of ISSRE*, 1999.
12. Jung H.W. and Choi B. "Optimization Models for Quality and Cost of Modular Software Systems", *European Journal of Operational Research*, 112 (1999), 613-619.
13. Kazman, R. et al. "Experience with Performing Architecture Tradeoff Analysis", *Proc. of ICSE99*, 1999.
14. Kazman, R. et al. "Quantifying the Costs and Benefits of Architectural Decisions", *Proc. of ICSE01*, 2001.
15. Kellerer, H. et al. "Knapsack Problems", *Springer-Verlag*, 2004.
16. Klein, M. and Kazman, R. "Attribute-based Architectural Styles", *CMU-SEI-99-TR-22, SEI, CMU*, 1999.
17. Li, J. et al. "An Empirical Study of Variations in COTS-based Software Development Processes in Norwegian IT Industry", *Proc. of METRICS'04*, 2004.
18. Menzies, T. and Cukic, B. "How Many Tests are Enough", *Handbook of Software Engineering and Knowledge Engineering*, Volume 2, 2001.
19. Perkins, C. "IP Mobility Support", *RCF 3344 in IETF*, 2002.
20. Trivedi K., "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", J. Wiley and S., 2001.
21. Voas, J. M. and Miller, K. W. "Software testability: The new verification", *IEEE Software*, pages 17-28, May 1995.
22. Wattanapongsakorn N. "Reliability Optimization for Software Systems With Multiple Applications", *Fast Abstracts of ISSRE01*, 2001.
23. Wattanapongsakorn N. and Levitan S. "Reliability Optimization Models for Fault-Tolerant Distributed Systems", *Proc. of Annual Reliability and Maintainability Symposium*, 2001.
24. Yakimovich, D., Bieman, J. M., Basili, V. R."Software architecture classification for estimating the cost of COTS integration.", *Proc. of ICSE*, pages 296-302, June 15-21, 1999.
25. OMG, "MOF 2.0/XMI Mapping Specification", v2.1 formal/05-09-01.
26. argouml.tigris.org
27. www.lindo.com

On the Modular Representation of Architectural Aspects

Alessandro Garcia[1], Christina Chavez[2], Thais Batista[3], Claudio Sant'anna[4], Uirá Kulesza[4], Awais Rashid[1], and Carlos Lucena[4]

[1] Computing Department, Lancaster University, United Kingdom
a.garcia@lancaster.ac.uk, marash@comp.lancs.ac.uk
[2] Computer Science Department, Federal University of Bahia, Brazil
flach@dcc.ufba.br
[3] Computer Science Department, Federal University of Rio Grande do Norte, Brazil
thais@ufrnet.br
[4] Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Brazil
claudios@les.inf.puc-rio.br, uira@les.inf.puc-rio.br, lucena@inf.puc-rio.br

Abstract. An architectural aspect is a concern that cuts across architecture modularity units and cannot be effectively modularized using the given abstractions of conventional Architecture Description Languages (ADLs). Dealing with crosscutting concerns is not a trivial task since they affect each other and the base architectural decomposition in multiple heterogeneous ways. The lack of ADL support for modularly representing such aspectual heterogeneous influences leads to a number of architectural breakdowns, such as increased maintenance overhead, reduced reuse capability, and architectural erosion over the lifetime of a system. On the other hand, software architects should not be burdened with a plethora of new ADL abstractions directly derived from aspect-oriented implementation techniques. However, most aspect-oriented ADLs rely on a heavyweight approach that mirrors programming languages concepts at the architectural level. In addition, they do not naturally support heterogeneous architectural aspects and proper resolution of aspect interactions. This paper presents AspectualACME, a simple and seamless extension of the ACME ADL to support the modular representation of architectural aspects and their multiple composition forms. AspectualACME promotes a natural blending of aspects and architectural abstractions by employing a special kind of architectural connector, called *Aspectual Connector*, to encapsulate aspect-component connection details. We have evaluated the applicability and scalability of the AspectualACME features in the context of three case studies from different application domains.

Keywords: Architecture Description Languages, Aspect-Oriented Software Development, Architectural Connection.

1 Introduction

Aspect-Oriented Software Development (AOSD) [8] is emerging as a promising technique to promote enhanced modularization and composition of crosscutting

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 82 – 97, 2006.
© Springer-Verlag Berlin Heidelberg 2006

concerns through the software lifecycle. At the architectural level, aspects provide a new abstraction to represent concerns that naturally cut across modularity units in an architectural description, such as interfaces and layers [1, 6, 9, 15]. However, the representation of architectural aspects is not a straightforward task since they usually require explicit representation mechanisms to address the heterogeneous manifestation of some widely-scoped properties, such as error handling strategies, transaction policies, and security protocols [5, 6,,10, 11]. By heterogeneous manifestation of widely-scoped properties – or, simply, *heterogeneous crosscutting* –, we mean that some properties impact multiple points in a software system, but the behavior that is provided at each of those points is different. Such architectural crosscutting concerns may interact with the affected modules in a plethora of different ways. Moreover, aspects may interact with each other at well-defined points in an architectural description. Hence, it is imperative to provide software architects with effective means for enabling the modular representation of aspectual compositions.

Software Architecture Description Languages (ADLs) [16] have been playing a central role on the early systematic reasoning about system component compositions by defining explicit connection abstractions, such as interfaces, connectors, and configurations. Some Aspect-Oriented Architecture Description Languages (AO ADLs) [19-22] have been proposed, either as extensions of existing ADLs or developed from scratch employing AO abstractions commonly adopted in programming frameworks and languages, such as aspects, join points, pointcuts, advice, and inter-type declarations. Though these AO ADLs provide interesting first contributions and viewpoints in the field, there is little consensus on how AOSD and ADLs should be integrated, especially with respect to the interplay of aspects and architectural connection abstractions [1, 6, 24, 17]. In addition, such existing proposals typically provide heavyweight solutions [1, 25], making it difficult their adoption and the exploitation of the available tools for supporting ADLs. More importantly, they have not provided mechanisms to support the proper modularization of heterogeneous architectural aspects and their compositions.

This paper present AspectualACME, a general-purpose aspect-oriented ADL that enhances the ACME ADL [14] in order to support improved composability of heterogeneous architectural aspects. The composition model is centered on the concept of aspectual connector, which takes advantage of traditional architectural connection abstractions – connectors and configuration – and extends them in a lightweight fashion to support the definition of some composition facilities such as: (i) heterogeneous crosscutting interfaces at the connector level, (ii) a minimum set of aspect interaction declarations at the attachment level, and (iii) a quantification mechanism for attachment descriptions. Our proposal does not create a new aspect abstraction and is strictly based on enriching the composition semantics supported by architectural connectors instead of introducing elements that elevate programming language concepts to the architecture level. This paper also discusses the applicability and scalability of the proposed ADL enhancements in the context of three case studies from different domains, and the traceability of AspectualACME models to detailed aspect-oriented design models.

The remainder of this paper is organized as follows. Section 2 introduces the case study used through the paper, and illustrates some problems associated with the lack of explicit support for modularizing heterogeneous architectural aspects and their interactions. Section 3 presents AspectualACME. Section 4 describes the evaluation of our approach. Section 5 compares our proposal with related work. Finally, Section 6 presents the concluding remarks and directions for future work.

2 Health Watcher: A Case Study

In this section we present the basic concepts of the ACME ADL [14] (Section 2.1) and discuss the architecture design of the case study that we are going to use as running example through the paper (Section 2.2), with emphasis on the heterogeneous crosscutting nature of some architectural concerns (Section 2.3) and their interactions (Section 2.4).

2.1 ACME in a Nutshell

ACME is a general purpose ADL proposed as an architectural interchange language. Architectural structure is described in ACME with components, connectors, systems, attachments, ports, roles, and representations. *Components* are potentially composite computational encapsulations that support multiple interfaces known as *ports*. *Ports* are bound to ports on other components using first-class intermediaries called *connectors* which support the so-called *roles* that attach directly to ports. *Systems* are the abstractions that represent configurations of components and connectors. A system includes a set of components, a set of connectors, and a set of attachments that describe the topology of the system. *Attachments* define a set of port/role associations. *Representations* are alternative decompositions of a given element (component, connector, port or role) to describe it in greater detail. *Properties* of interest are *<name, type, value>* triples that can be attached to any of the above ACME elements as annotations. Properties are a mechanism for annotating designs and design elements with detailed, generally non-structural, information. Architectural *styles* define sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain. The ACME type system provides an additional dimension of flexibility by allowing type extensions via the *extended with* construct. These ACME concepts are illustrated through this paper.

2.2 Health Watcher Architecture

The HealthWatcher (HW) system is a Web-based information system developed by the Software Productivity research group from the Federal University of Pernambuco [27]. It supports the registration of complaints to the health public system. Figure 1 illustrates a partial, simplified ACME [14] textual and graphical representation of the

HW architectural description, which combines a client-server style with a layered style [30]. It is composed of five main architectural concerns: (i) the GUI (Graphical User Interface) component provides a Web interface for the system, (ii) the Distribution component externalizes the system services at the server side and support their distribution to the clients, (iii) the Business component defines the business elements and rules, (iv) the TransactionManager and Data components address the persistency concern by storing the information manipulated by the system, and (v) the ErrorHandling component which is charge of supporting forward error recovery through exception handling.

Figure 1 also illustrates a set of provided/required ports and connectors which make explicit the interactions between the architectural components. The saveEntity required port from the GUI component, for example, is linked to the distributedInterface provided port from the Distribution component by means of a connector. Despite many of the interactions between the architectural components have been appropriately represented using the port and connector abstractions, it is not possible to use these common ADL abstractions to represent the crosscutting relationships between two component services. Consider, for example, the transactionService provided port of the Transaction Manager component. It affects the execution of the savingService provided port of the Business component, by delimiting the occurrence of a business transaction (operations of begin, end and rollback transaction) before and after the execution of every operation invoked on savingService port. There is no existing abstraction in current ADLs which explicitly captures this crosscutting semantic between architectural component services. Because of lack of support to represent such kinds of crosscutting interactions between components, Figure 1 alternatively models it by defining the useTransaction required port. This description, however, does not make explicit the existence of crosscutting relationships between the components.

2.3 Heterogeneous Architectural Crosscutting

Exception handling is considered a widely-scoped influencing concern in the HW architectural specification [28], which is mostly realized by the Error Handling component. This component consists of the system exception handlers, and it provides the services in charge of determining at runtime the proper handler for each of the exceptions exposed by the system components, such as Distribution, Persistence [24], and TransactionManager. In fact, Figure 1 shows that the Error Handling component has a crosscutting impact on the HW architecture since it affects the interfaces of several components in the layered decomposition. Almost all the architectural interfaces need to expose erroneous conditions, which in turn need to be handled by the error handling strategy. Figure 1 gives some examples of exceptional interfaces in the component's ports savingService and distributedInterface. Hence, the broadly-scoped effect of this component denotes its crosscutting nature over the modular architecture structure of the HW system.

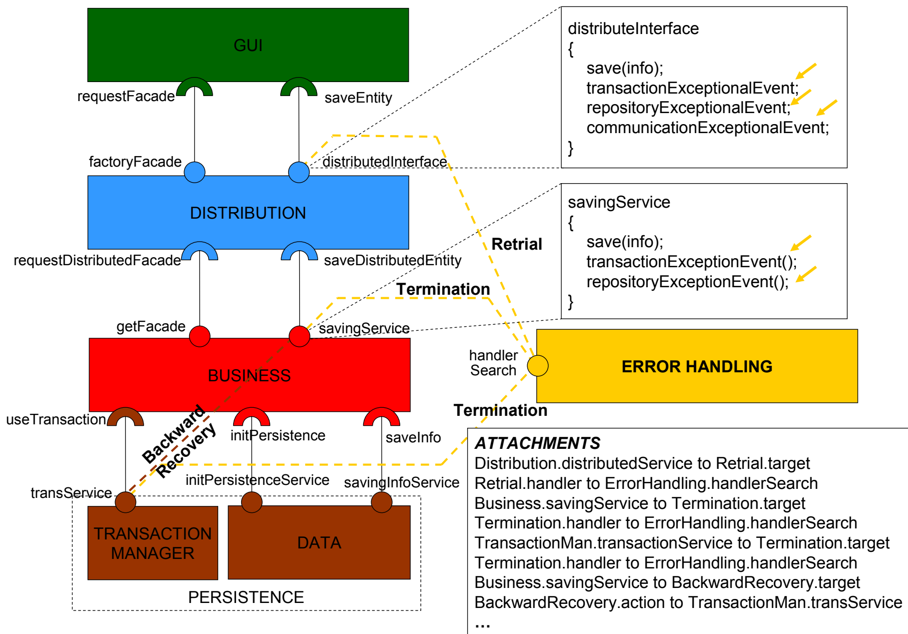


Fig. 1. Error Handling in the HW Architecture: A Heterogeneous Crosscutting Concern

However, the influence of this crosscutting concern is not exactly the same over each affected HW component; it crosscuts a set of interfaces in heterogeneous ways, depending on the way the exception should be handled in the target component. In the HW system, there is at least two forms of interaction between a faulty component and the Error Handling component: the termination protocol (Termination connector), and the retry protocol (Retrial connector). However, the heterogeneous crosscutting composition of ErrorHandling and the affected architectural modules can not be expressed in a modular way. For instance, the connector Termination needs to be replicated according to the number of affected interfaces, and separated connectors for expressing the Retrial collaboration protocols need to be created. For simplification, Figure 1 only contains some examples of those connectors; the situation is much worse in the complete description of the HW architecture since almost all the interfaces expose exceptions. Also the attachment section contains a number of replicated, similar attachments created only for the sake of combining the replicated error handling connectors (Figure 1). Finally, the "provided" interface handlingStrategy needs to be connected with the "provided" interfaces containing exceptional events, which is not allowed in conventional ADLs.

2.4 Aspect Interaction

In addition, there are other architectural breakdowns when using conventional ADLs to define interactions between crosscutting concerns. For example, the TransactionManager

is another architectural aspect that crosscuts several elements in the Business layer in order to determine the interfaces that execute transactional operations. Most of these affected interfaces are also connected with the error handling connectors (Section 2.2). Figure 1 illustrates this situation for the savingService interface. The problem is that it is impossible to express some important architectural information and valid architectural configurations involving the interaction of the ErrorHandling and TransactionManager aspects. For example, although the attachments section allows the architect to identify that both aspects are actuating over the same architectural elements, it is not possible to declare which aspect has precedence over others affecting the same interfaces or whether only one or both of the backward and forward recovery strategies should be used.

3 AspectualACME

This Section presents the description of AspectualACME. We present the ACME extension to support the modeling of the crosscutting interactions (Section 3.1) and the definition of a quantification mechanism (Section 3.2). This section ends with a discussion about the AspectualACME support for modeling heterogeneous architectural aspects (Section 3.3) and aspect interaction (Section 3.4).

3.1 Aspectual Connector

As software architecture descriptions rely on a *connector* to express the interactions between components, an equivalent abstraction must be used to express the crosscutting interactions. We define an *aspectual component* as a component that represents a crosscutting concern in a crosscutting interaction. The traditional connector is not enough to model the crosscutting interaction because the way that an aspectual component composes with a regular component is slightly different from the composition between regular components only. A crosscutting concern is represented by provided services of an aspectual component and it can affect both provided and required services of other components which can be, in turn, regarded as structural join points [8] at the architectural level. As discussed in Sections 2.2 and 2.3, since ADL valid configurations are those that connect provided and required services, it is impossible to represent a connection between a provided service of an aspectual component and a provided service without extensions to the traditional notion of architectural connections. Although ACME itself does not support a syntactic distinction between provided and required ports, this distinction can be expressed using properties or declaring port types.

In order to express the crosscutting interaction, we define the *Aspectual Connector (AC)*, an architectural connection element that is based on the connector element but with a new kind of interface. The purpose of such a new interface is twofold: to make a distinction between the elements playing different roles in a

crosscutting interaction – i.e., affected base components and aspectual components; and to capture the way both categories of components are interconnected. The AC interface contains: (i) *base roles*, (ii) *crosscutting roles,* and (iii) a *glue* clause. Figure 2 depicts a high-level description of a traditional connector (Fig. 2a) and an aspectual connector (Fig. 2b).

```
Connector homConnector = {
    Role aRole1;
    Role aRole2;
}
```

(a) Regular connector in ACME

```
AspectualConnector homConnector = {
    BaseRole aBaseRole;
    CrosscuttingRole aCrosscuttingRole;
    Glue glueType;
}
```

(b) Aspectual connector in AspectualACME

Fig. 2. Regular and Aspectual Connectors

The *base role* may be connected to the port of a component (provided or required) and the *crosscutting role* may be connected to a port of an aspectual component. The distinction between base and crosscutting roles addresses the constraint typically imposed by many ADLs about the valid configurations between provided and required ports. An aspectual connector must have at least one base role and one crosscutting role. The composition between components and aspectual components is expressed by the *glue* clause. The aspectual glue specifies the way an aspectual component affects one or more regular components. There are three types of aspectual glue: *after*, *before*, and *around*. The semantics is similar to that of advice composition from AspectJ [29].

```
AspectualConnector aConnector = {
   BaseRole aBaseRole1, aBaseRole2;
   CrosscuttingRole aCrosscuttingRole1,
                     aCrosscuttingRole2;
   Glue { aCrosscuttingRole1 before aBaseRole1;
          aCrosscuttingRole2 after aBaseRole2;   }
}
```

Fig. 3. Heterogeneous aspectual connector

The glue clause can be simply a declaration of the glue type (Figure 2b), or a block with multiple declarations, where each relates a crosscutting role, a base role and a specific glue type (Figure 3). The description of heterogeneous aspectual interactions (Section 3.3) requires more elaborated glue clauses.

Although the idea of the aspectual connector is derived from the traditional connector, it is not modeled as a subtype of the traditional connector, since the aspectual connector can be used in a connection between two provided ports. This would result in an invalid configuration (ill-formed connection) using the traditional connector and its subtypes.

Figure 4 contains a graphical notation that we propose to represent Aspectual Connectors. C1 is an aspectual connector that defines a crosscutting and heterogeneous interaction involving the Aspectual Component, Component 1, and Component 2.

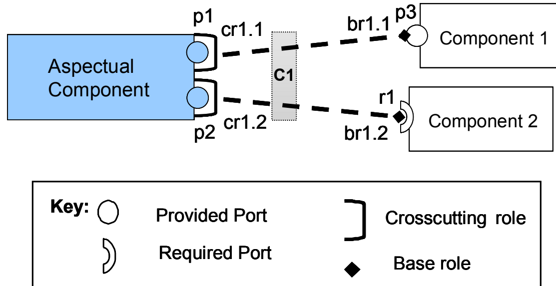


Fig. 4. Graphical Notation to the Aspectual Connector

3.2 Quantification Mechanism

A base role of an aspectual connector may be bound to several ports of possibly different components. These ports represent *structural join points* that may be affected by aspectual components. To express these bindings, many attachments should be defined, where each one binds the *same* base role instance to a different component port. We propose an extension to the attachments part of an ACME configuration to allow the use of patterns. Wildcards such as '*', can be used in attachments to concisely describe sets of ports to be attached to the same base role.

```
System Example = {
Component aspectualComponent = { Port aPort }
AspectualConnector aConnector = {
    BaseRole aBaseRole;
    CrosscuttingRole aCrosscuttingRole;
    glue glueType;
}
Attachments {
   aspectualComponent.aPort to aConnector.aCrosscuttingRole
   aConnector.aBaseRole to *.prefix* }
}
```

Fig. 5. ACME Description of the Composition

The attachment "`aConnector.aBaseRole to *.prefix*`" (Figure 5) specifies the binding between `aConnector.aBaseRole` and ports from the "set of component ports where the port name begins with `prefix`". By avoiding explicit

enumeration of ports and definition of multiple attachments, this extension promotes economy of expression and improves writability in architectural configurations.

3.3 Heterogeneous Architectural Aspects

Figure 2b presented a simple aspectual connector that has a homogeneous crosscutting impact on the architectural decomposition. Figure 3 shows how AspectualACME supports heterogeneous crosscutting. Multiple base and crosscutting roles can be used to define the different ways a crosscutting concern can affect the component interfaces. Different or similar glue types can be used in the definition of the pairs of base and crosscutting roles. Figure 6a is an example of heterogeneous aspectual connector for the error handling concern discussed in Section 2.2. Note that the two ways of interacting with the ErrorHandling component – i.e. retrial and termination – can now be modularized in a single architectural element. In addition, quantification mechanisms can be used in the attachments specification to describe in single statements which component ports are affected by those two crosscutting roles specified in the ForwardRecovery connector (Figure 6b).

```
AspectualConnector ForwardRecovery = {
   BaseRole toBeTerminatedTarget, toBeRetriedTarget;
   CrosscuttingRole termination, retrial;
   Glue {termination after toBeTerminatedTarget;
         retrial after toBeRetriedTarget;
   }
}
```

(a) an example of heterogeneous aspectual connector

```
Attachments {
  ForwardRecovery.toBeTerminatedTarget to *.*Service
  ForwardRecovery.termination to ErrorHandling.handlerSearch
  Distribution.distributedInterface to
      ForwardRecovery.toBeRetriedTarget
  ForwardRecovery.retrial to ErrorHandling.handlerSearch
  ...// to be continued in Figure 5b
}
```

(b) specification of join points using AspectualACME quantification mechanisms

Fig. 6. Supporting Heterogeneous Crosscutting

Figure 7 presents a graphical notation for the HW example, where the ForwardRecovery is defined as a heterogeneous aspectual connector. The yellow vertical rectangle indicates that ForwardRecovery is a heterogeneous aspectual connector.

3.4 Aspect Interaction

AspectualACME also allows the specification of aspectual architecture-level interaction between two or more aspectual connectors which have join points in

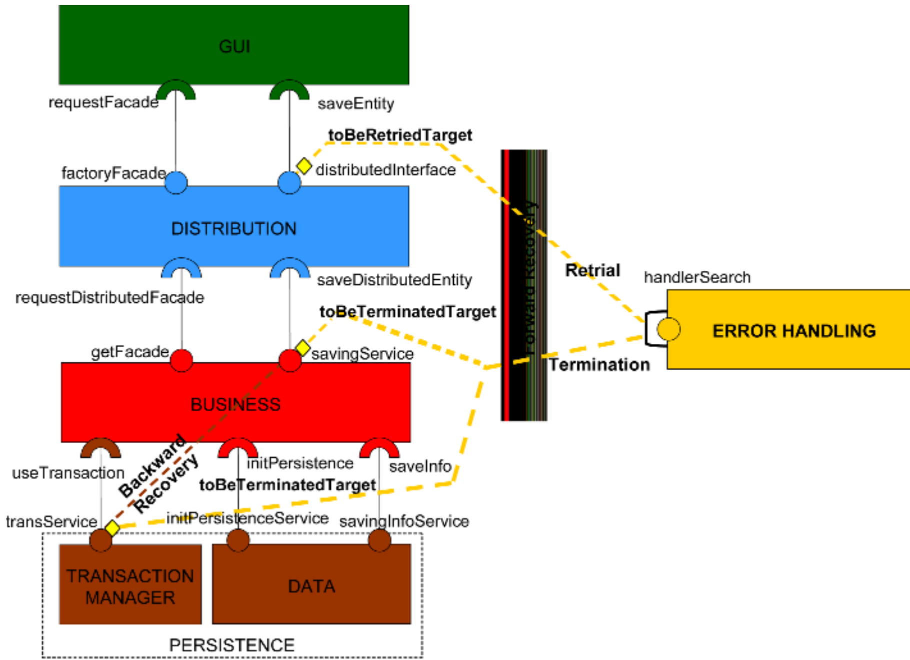


Fig. 7. An Example of Aspectual Connector: Forward Recovery

common. Such interactions are declared in the configuration description since the attachments part is the place where join points are identified. The ADL supports two basic kinds of composition operators: precedence and XOR (Figure 8b). The architect can specify that the precedence is either valid for the whole architecture or only at specific join points. Figure 8b illustrates both situations: (i) in general, the Retrial connector has precedence over the Termination connector at all the join points they have in common, and (ii) at the port savingService, it is always tried first forward recovery through termination-based error handling and, second, the backward recovery with abort in case the exception was not successfully handled. When there is a precedence relation between two connectors X and Y, where the execution of Y depends on the satisfaction of a condition associated with X, the architect can explicitly document it using a condition statement together with an around glue in X. Figure 8b also illustrates the use of XOR: at a given join point, only one of the either termination or retrial should be non-deterministically chosen. Finally, it is important to highlight that the elements participating in a precedence or XOR clause can be components instead of connectors: it means that the relationship applies to all the connectors involving the two components (see Section 4.1).

```
AspectualConnector BackwardRecovery = {
  BaseRole target;
  CrosscuttingRole transBegin, transAbort, transCommit;
  Glue {transBegin before target;
        transCommit after target;
        transAbort after target;
  }
}
```

(a) an example of aspectual connector

```
Attachments {
  //continued from Figure 4c
  Business.savingService to BackwardRecovery.target
  BackwardRecovery.transBegin to TransactionManager.transService
  BackwardRecovery.transCommit to TransactionManager.transService
  BackwardRecovery.transAbort to TransactionManager.transService
  Distribution.distributedInterface to ForwardRecovery.retriedTarget
  ForwardRecovery.Retrial to ErrorHandling.handlerSearch
  Precedence {
    ForwardRecovery.retrial, ForwardRecovery.termination;
    savingService:
         ForwardRecovery.termination, BackwardRecovery.transAbort;
  }
  XOR {
    ForwardRecovery.resumption, ForwardRecovery.termination;
  }
}
```

(b) specification aspectual interactions

Fig. 8. Supporting Aspect Interaction Declarations

4 Evaluation

This Section presents the evaluation of AspectualACME in three case studies with respect to the usefulness of the proposed composition enhancements. We have evaluated the applicability and scalability of the notion of Aspectual Connectors (Section 3.1) and the extensions provided in AspectualACME (Sections 3.2 to 3.4) in the context of three case studies: the HealthWatcher system [28] (Section 2), a context-sensitive tourist information guide (TIG) system [9, 1], and AspectT – a multi-agent system framework [10, 11, 12]. As indicated in Table 1, the TIG architecture encompassed the manifestation of three heterogeneous architectural aspects: replication, security, and performance. The AspectT architecture included five main heterogeneous architectural aspects: autonomy, adaptation, learning, code mobility, and interaction. The choice of such case studies was driven by the heterogeneity of the aspects, and the different ways they affect the dominant architectural decomposition and each other.

Our approach has scaled up well in all the case studies mainly by the fact that AspectualACME follows a symmetric approach, i.e. there is no explicit distinction between regular components and aspectual components. The modularization of the crosscutting interaction into connectors facilitated, for example, the reuse of the persistence component description from the first to the second case study. Persistence was a crosscutting concern only in the HealthWatcher architecture (Figure 7). Hence,

we have not applied an aspectual connector in the TIG architectural specification. The definition of quantification mechanisms (Section 3.2) in attachments also has shown to be the right decision choice as it improves the reusability of connectors. The other reason was that is was easier to determine how multiple interacting aspects affect each other by looking in a single place in the architectural description – i.e. the attachments specification.

Table 1. Examples of Heterogeneous Architectural Aspects and their Interactions

Case Study	Heterogeneous Aspects	Aspect Interactions	
		# Total	Some Examples
Health Watcher	Error Handling, Transaction Management, Distribution	13	*Precedence*: Error Handling, Transaction Manag. *XOR*: ForwardRecovery.resumption, Forward Recovery.termination
TIG	Replication, Performance, Security	7	*Precedence*: Security, Performance *XOR*: Replication.passive, Replication.active
AspectT	Autonomy, Adaptation, Learning, Code Mobility, Interaction	15	*Precedence*: Interaction, Autonomy, Adaptation *Precedence*: Autonomy.execution, Autonomy.proactiveness *XOR*: Mobility, Collaboration

Table 1 presents a summary on how AspectuaACME has been used through the three case studies to capture certain heterogeneous architectural aspects. It also describes how many aspectual interactions have been explicitly captured in those studies, followed by some examples of Precedence and XOR interactions. In our evaluation, we have noticed that two or more crosscutting roles of the same heterogeneous aspectual connector can naturally be linked to the same join point (a component port). Hence, the proposed aspect interaction mechanisms (Section 3.4) can be used to define their relationships. For example, Table 1 shows a XOR relationship in the HW architecture involving two crosscutting roles of the same connector: ForwardRecovery. Other interesting possibilities have been also explored in the case studies, such as declaring that all the connectors of Error Handling aspect have precedence over all the connectors of Transaction Management in the HW system. Also, we have observed that the explicit definition of such aspectual relationships in the architectural stage enhances the documentation of design choices that need to be observed later on the design of applications, and variation points in a certain product-line design [31].

5 Related Work

There is a diversity of viewpoints on how aspects (and generally concerns) should be modeled in ADLs. However, so far, the introduction of AO concepts into ADLs has been experimental in that researchers have been trying to incorporate mainstream AOP concepts into ADLs. In contrast, we argue that most of existing ADLs

abstractions are enough to model crosscutting concerns. For this purpose, it is just necessary to define a new configuration element based on the traditional connector concept.

Most AO ADLs are different from AspectualACME because they introduce a lot of concepts to model AO abstractions (such as, aspects, joinpoints, and advices) in the ADL. Navasa et al 2005 [19] present a proposal to introduce the aspect modeling in the architecture design phase. Aspects are used to facilitate the architecture evolution by allowing easily either to modularize crosscutting concerns, or to incorporate new requirements in the system architecture. The composition between the architectural components and the aspects is based on an exogenous control-driven co-ordination model. The incorporation of the authors' model to existing ADLs, such as ACME, is still under investigation. Navasa et al 2002 [18] do not propose an AO ADL, but define a set of requirements which current ADLs need to address to allow the management of crosscutting concerns using architectural connection abstractions. The requirements are: (i) definition of primitives to specify joinpoints in functional components; (ii) definition of the aspect abstraction as a special kind of component; and (iii) specification of connectors between joinpoints and aspects. The authors suggest the use of existing coordination models to specify the connectors between functional components and aspects. Differently from our lightweight approach, they suggest the definition of AO specific ADL constructs. Furthermore, they do not mention in their proposal the need for supporting important AO properties such as quantification, interaction between aspects and heterogeneous aspects.

DAOP-ADL [22] defines components and aspects as first-order elements. Aspects can affect the components' interfaces by means of: (i) an evaluated interface which defines the messages that aspects are able to intercept; and (ii) a target events interface responsible for describing the events that aspects can capture. The composition between components and aspects is supported by a set of aspect evaluation rules. They define when and how the aspect behavior is executed. Besides, they also include a number of rules concerning with interaction between aspects. With regards to precedence, aspects can be evaluated in two ways: sequentially or concurrently. In addition, aspects can share information using a list of input and/or output global properties. Nevertheless, DAOP-ADL does not provide mechanisms to support quantification at the attachment level and explicit modularization of heterogeneous architectural aspects.

Similarly to our proposal, FuseJ [26] defines a unified approach between aspects and components, that is, FuseJ does not introduce a specialized aspect construct. It provides the concept of a gate interface that exposes the internal implementation functionality of a component and offers access-point for the interactions with other components. In a similar way to our proposal, FuseJ concentrates the composition model in a special type of connector that extends regular connectors by including constructs to specify how the behaviour of one gate crosscuts the behaviour of another gate. However, differently from our work, our compositional model works in conjunction with the component traditional interface while FuseJ defines the gate interface that exposes internal implementation details of a component. However, FuseJ provides explicit support neither for defining the interaction between aspects nor for modularizing heterogeneous aspects. Moreover, it only allows quantification over the same gate methods. In addition, FuseJ does not work with the notion of

configuration. It includes the definition of the connection inside the connector itself. This contrasts with the traditional way that ADLs works – that declares a connector and binds connectors' instances at the configuration section.

Pessemier et al [21] defines the Fractal Aspect Component (FAC), a general model for mixing components and aspects. Their aim is to promote the integration between aspect-oriented programming (AOP) and component-based software engineering (CBSE) paradigms. FAC model proposes three new abstractions: (i) aspect components – that modularize a crosscutting concern by implementing the service of a regular component as a piece of an around advice; (ii) aspect bindings – which define bindings between regular and aspectual components; and (iii) aspect domains – that represents the reification of regular components affected by aspect components. FAC model is implemented under Fractal [2], an extensible and modular component model, and its respective ADL. There are similarities between the aspect component from the FAC model and our aspectual connector. Both are used to specify crosscutting concerns existing in the system architecture. The aspect bindings of FAC define a link between a regular and an aspect component. This latter can modify/extend the behavior of the former by affecting its exposed join points. In our approach, this is addressed by the definition of: (i) base and crosscutting roles – which allow specifying the binding between two components; and (ii) the glue clause – that define the semantic of crosscutting composition between them.

6 Conclusions and Future Work

This paper has addressed current issues related to aspect-oriented architecture modeling and design. The analysis of heavyweight solutions provided by some AO ADLs yielded to the design of AspectualACME, a general-purpose aspect-oriented ADL that supports improved composability of heterogeneous architectural aspects. The composition model is centered on the concept of aspectual connector, which takes advantage of traditional architectural connection abstractions – connectors and configuration – and, based on them, provides a lightweight support for the definition of some composition facilities such as: (i) heterogeneous crosscutting interfaces at the connector level, (ii) a minimum set of aspect interaction declarations at the attachment level, and (iii) a quantification mechanism for attachment descriptions. In this way, AspectualACME encompasses a reduced set of minor extensions, thereby avoiding the introduction of additional complexity in architectural descriptions. The paper also discussed the applicability and scalability of the proposed ADL enhancements in the context of three case studies from different domains. Our approach has scaled up well in all the case studies mainly by the fact that AspectualACME follows a symmetric approach, i.e. there is no explicit distinction between regular components and aspectual components. Also, we have observed that explicit aspect interaction declarations in the architectural stage enhances the documentation of design choices that need to be observed later on the design of applications.

As future work, we plan to further elaborate on several issues related to the expressiveness of the AspectualACME language, as well as on traceability issues. Architectural descriptions in AspectualACME can be mapped to aspect-oriented design languages that support aspect-oriented modeling at the detailed design level,

such as aSideML [5] and Theme/UML [8]. Tools need to be developed to support the creation of AspectualACME descriptions and their transformation to design level descriptions. Once these tools are available, designers may fully exploit the benefits from the aspect-oriented ADL and explicitly "design" aspectual connectors.

Acknowledgments

This work has been partially supported by CNPq-Brazil under grant No.479395/2004-7 for Christina. Alessandro is supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. This work has been also partially supported by CNPq-Brazil under grant No.140252/03-7 for Uirá, and grant No.140214/04-6 for Cláudio. The authors are also supported by the ESSMA Project under grant 552068/02-0.

References

1. Batista, T., Chavez, C., Garcia, A., Sant'Anna, C., Kulesza, U., Rashid, A., Filho, F. Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. Workshop on Early Aspects ICSE'06, pages 3-9, May 2006, Shanghai, China.
2. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B. An open component model and its support in Java. In Proc. of the Intl Symposium on Component-based Software Engineering, Edinburgh, Scotland, May 2004.
3. Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proc. 5th Intl. Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, 20-24 March 2006.
4. Chavez, C. A Model-Driven Approach for Aspect-Oriented Design. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, April 2004.
5. Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C. Taming Heterogeneous Aspects with Crosscutting Interfaces. Journal of the Brazilian Computer Society, vol.12, N.1, June 2006.
6. Chitchyan, R., et al. A Survey of Analysis and Design Approaches. AOSD-Europe Report D11, May 2005.
7. Clarke, S. and Walker, R. Generic aspect-oriented design with Theme/UML. In [8], pages 425-458.
8. Filman, R.,Tzilla E., Siobhan Clarke, and Mehmet Aksit, editors. Aspect-Oriented Software Development. Addison-Wesley, Boston, 2005.
9. Garcia, A., Batista, T., Rashid, A., Sant'Anna, C. Driving and Managing Architectural Decisions with Aspects. Proc. SHARK.06 Workshop at ICSR.06, Turin, June, 2006.
10. Garcia, A., Kulesza, U., Lucena, C. Aspectizing Multi-Agent Systems: From Architecture to Implementation. In: Software Engineering for Multi-Agent Systems III, Springer-Verlag, LNCS 3390, December 2004, pp. 121-143.
11. Garcia, A., Lucena, C. Taming Heterogeneous Agent Architectures with Aspects. Communications of the ACM, March 2006. (accepted)
12. Garcia, A., Lucena, C., Cowan, D. Agents in Object-Oriented Software Engineering. Software: Practice & Experience, Elsevier, Volume 34, Issue 5, April 2004, pp. 489-521.

13. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions on Aspect-Oriented Software Development, Springer, LNCS, pp. 36 - 74, Vol. 1, No. 1, February 2006.
14. Garlan, D. et al. ACME: An Architecture Description Interchange Language, Proc. CASCON'97, Nov. 1997.
15. Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., Kulesza, U. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. 8th Workshop on Aspect-Oriented Modelling (AOM.06), AOSD.06, Bonn, Germany.
16. Medvidovic, N., Taylor, R. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Soft. Eng., 26(1):70-93, Jan 2000.
17. Mehta N., Medvidovic, N. and Phadke, S. Towards a Taxonomy of Software Connectors. Proc. of the 22nd Intl Conf. on Software Engineering (ICSE), Limerick, Ireland, pp. 178 – 187, 2000.
18. Navasa, A. et al. Aspect Oriented Software Architecture: a Structural Perspective. Workshop on Early Aspects, AOSD’2002, April 2002.
19. Navasa, A., Pérez, M. A., Murillo, J. M. Aspect Modelling at Architecture Design. EWSA 2005, pp. 41-58, LNCS 3527, Pisa, Italy, 2005.
20. Pérez, J., et al., E. PRISMA: Towards Quality, Aspect-Oriented and Dynamic Software Architectures. In Proc. of 3rd IEEE Intl Conf. on Quality Software (QSIC 2003), Dallas, Texas, USA, November (2003).
21. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L. A Model for Developing Component-based and Aspect-oriented Systems. In 5th International Symposium on Software Composition (SC'06), Vienna, Austria, March 2006.
22. Pinto, M., Fuentes, L., Troya, J., "A Dynamic Component and Aspect Platform”, The Computer Journal, 48(4):401-420, 2005.
23. Quintero, C., et al. Architectural Aspects of Architectural Aspects. Proc. of European Workshop on Software Architecture (EWSA2005)- Pisa, Italy, June 2005, LNCS 3527.
24. Rashid, A., Chitchyan, R. Persistence as an Aspect. Proc. of the 2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD'03), USA, March 2003.
25. Rashid, A., Garcia, A., Moreira, A. Aspect-Oriented Software Development Beyond Programming. Proc. of ICSE.06, Tutorial Notes, May 2006, Shanghai, China.
26. Suvée, D., De Fraine, B. and Vanderperren, W. (2005) FuseJ: An architectural description language for unifying aspects and components. Software-engineering Properties of Languages and Aspect Technologies Workshop @ AOSD2005.
27. SPG – Software Productivity Group at UFPE. http://twiki.cin.ufpe.br/twiki/bin/view/SPG, 2006.
28. Soares, S., Laureano, E. and Borba, P.. Implementing Distribution and Persistence Aspects with AspectJ. In Proc. of OOPSLA'02, Seattle, WA, USA, 174-190, November 2002. ACM Press.
29. The AspectJ Team. “The AspectJ Programming Guide”. http://eclipse.org/aspectj/
30. Kulesza, U., et al. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. Proc. of the Intl Conf. on Software Maintenance (ICSM’06), Philadelphia, USA, September 2006.
31. Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. Proc. of the 9th Intl Conf. on Software Reuse (ICSR’06), Turin, Italy, June 2006.

Configurations by UML

Øystein Haugen and Birger Møller-Pedersen

Department of Informatics, University of Oslo
{oysteinh | birger}@ifi.uio.no

Abstract. Compared to UML 1.x, UML 2.x has improved the mechanisms for describing the architecture of systems. We show how to make UML 2.x describe configurations, not only in terms of setting values of system properties but also in terms of rearranging elements of the architecture. We also argue that the instance model of UML 2.x can be replaced by our notion of configurations and that this may imply a generalization of the notions of snapshot and constructor.

1 Introduction

By a *configuration* we mean the relative arrangement of parts [9]. In the field of computing we also consider binding of free variables and deciding the number of objects as being covered by configuring. One may wonder what distinguishes a configuration from an architecture as the latter term is often defined in very much the same way. It is not necessarily very important to make the distinction, but we consider a configuration more particular than an architecture. An architecture is most often intended to cover a set of concrete implementations or a set of concrete systems, and thus an architecture may define a set of configurations.

Applying UML 2 [10] to the modeling of system families (product lines), the need for modeling configurations has emerged. There are more or less well-established ways of modeling variations[6], and in the Families project [4] we have surveyed these and also shown how standard UML 2 mechanisms like ports with well-defined interfaces, specialization (with redefinition of types) and templates can be used for variation modeling [1].

Given models of variation in terms of explicit variation point model elements, specific systems are made by resolving feature variation models and thereby deciding on the variations in the family model [3, 7].

There is, however, a class of system families where the specific systems are merely configurations of the general system family. There are no explicit variation elements, but the specific systems are characterized by different configurations of structural properties of the general system family and by different values on attributes of parts of the system.

In this paper we will give a list of configuration challenges based on an example access control system, and analyze how existing UML 2 may handle such a challenge. Furthermore we shall see how this leads to generalized notions of snapshot and constructor without actually adding much to UML. We finally conclude that the UML 2 instance model may for these purposes be redundant.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 98 – 112, 2006.
© Springer-Verlag Berlin Heidelberg 2006

This paper is partially funded by the Families project under Eureka Σ! 2023 Programme, ITEA project ip02009. We would like to thank the Families project members for the inspiring discussions. We also thank Professor Stein Krogdahl for his thorough commenting of an earlier draft of the paper.

The remainder of the paper is organized as follows: In Section 2 we present the general Access Control System that constitutes our motivating baseline. In Section 3 we present the challenges that should result in a description that clearly is derived from the baseline, but still have introduced a number of new elements. Section 4 answers the challenges. In Section 5 we argue that our approach can also be applied to system snapshots and as constructors. Thus we end up with a notation that can not only describe structures, but also the corresponding behavior. We also argue that the UML instance model is superfluous. Section 6 concludes the paper.

2 The Access Control System Example

We will use an example (described in [2]) – an access control system – to illustrate the need for configuration modeling. In connection with UML 2 this example has been reused in [8].

Access control has to do with controlling the access of users to a set of access zones. Only a user with known identity and correct access rights shall be allowed to enter into an access zone. Access control systems will provide services like

1. User Access: The user will enter an access zone through an access point. The authentication of a user shall be established by some means for secret personal identification (PIN code). The authorization is based on the identity of the user and the access rights associated with that user relative to the security level of the access zone.
2. New User: A supervisor will have the ability to insert new users into the system.
3. PIN change: Users shall be able to change their personal code.

Fig. 1 is a simple *domain* class model for access control systems: An AccessPoint controls access to a Door, through which the user enters the AccessZone. The Authorizer controls access through an AccessPoint, and thus governs access to each AccessZone. Users interact with a Panel of the AccessPoint or the Console. There may be a Panel on either side of the Door.

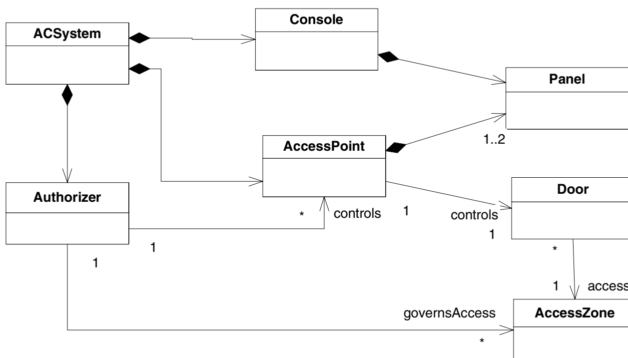


Fig. 1. Domain model for access control

A system family *design* model for access control systems typically include the modeling of the *common architecture* of all systems in the system family and the modeling of *variations* on parts of this architecture.

The class ACSystem (see Fig. 2) defines the common architecture of all access control systems: Each access control system contains a number of access points (from 2 to 100), represented by the *part* ap typed by the class AccessPoint (ap: AccessPoint [2...100]). The access points interact with the users (via the *port* User) who request access and are granted or denied access to the access zones, and access points control the doors via the *port* Door. The access points communicate with objects of type Authorizer to verify the validity of an access request. The ACSystem further has a Console that allows a supervisor to interact with the system (via the *port* Supervisor). Ports and parts are connected by means of *connectors*, and these specify potential communication.

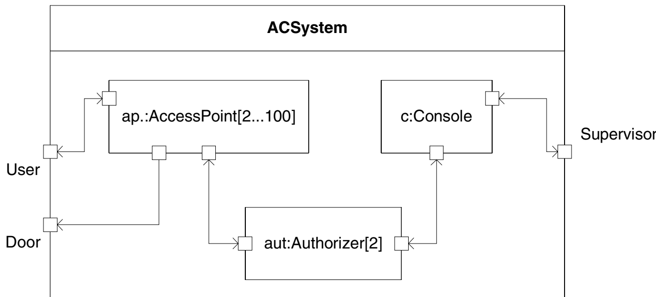


Fig. 2. Composite structure of the access control system class

Any object of class AccessPoint will have two integer attributes, one for the floor number and one for the security level. The floor number should always be between 0 and 10, and the security level between 1 and 4, see Fig. 3.

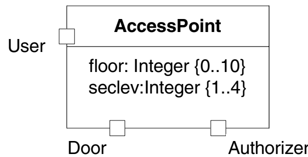


Fig. 3. Constraints in general on objects of class AccessPoint

For the access control system we consider the following variations: two kinds of access points. *Blocking access points* are access points where an operator may block/unblock the access points, while *logging access points* are access points, that log what is going on by sending signals to a logging device. The variations in types of access points are represented by two subclasses of AccessPoint, see Fig. 4.

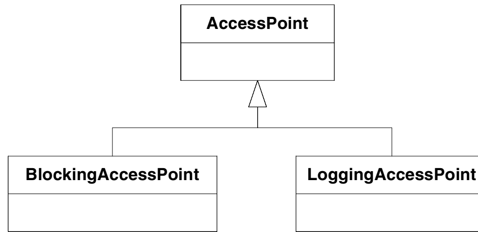


Fig. 4. Different kinds of access points

3 The Challenges – *MyACSystem*

The Access Control system sketched is a very general one. In any specific case, we shall have to add more information and be more specific. Let us assume that we want to describe an access control system MyACSystem for a specific building such that:

1. The number of access points on each floor differ, but within each floor the access points will have the same security level.
2. On the ground floor the security level is high and the access points will be LoggingAccessPoints.
3. On all other floors the security level is lower, and the access points are BlockingAccessPoints.
4. The access points on the ground floor are connected to one specific authorizer, while the other access points are connected to another authorizer. The reason for this is due to the difference in security level.

One question is whether MyACSystem can be modeled by the present UML 2. As mentioned in the introduction, variation on types is readily covered by class specialization, so the challenge here is the configuration in terms of specifying more accurately parts of the structure of the system and more closely values of attributes.

We argue that the challenges listed above are examples of challenges that we find in most real systems. Configuring a real system is sometimes comparable in size with a design job. Here we argue that configuring may be seen as a continuation of the design and that the same concepts can be applied.

The first challenge is about cardinal numbers and that the identified parts may have significant subsets where the objects of different subsets have important distinguishing characteristics. Assume that we were configuring an airline seat management system to be used for the in-plane entertainment system. We have a set of seats with some general properties, but then we want to characterize e.g. the seat width or earplug position, and that differs whether the seat is a window, middle or aisle seat.

The second and third challenge recognizes that all the access points may not be of the same class even though they are of the same abstract class (or implement the same interface) and that it may be advantageous to type subsets with this new information. In the seat management system we may want to distinguish between business seats and economy seats as they typically have very different entertainment possibilities.

The fourth challenge may look simple and trivial, but this challenge represents the very architectural challenge. We do not want to express only properties that refer to individual objects, but we also want to express how the objects may be logically interlinked for communication. In UML 2 this major architectural concept is achieved in the composite structure by connectors between parts. It should be well known that this cannot properly be achieved through classes and associations alone [5].

In our assumed seat management system it is obvious that the business class seats need a very different cable set than do the economy seats. We may for the sake of the discussion assume that each business seat is connected to its own media provider center while all the economy seats are connected to one special economy media provider.

The answer to these challenges must not only be technically sound, but it should also be visually pleasing and give intuitively the correct understanding.

4 Answering the Challenges

In this Section we shall try and answer the challenges from Section 3.

4.1 A Specialized Class or an Instance?

Firstly we need to decide whether MyACSystem is one of its kind, or a more specialized class of access control system of which there in principle may be more than one.

In UML there is a significant language difference between modeling a system that is the only one of its kind, and modeling a specialized class of systems. In case of only one, the modeler may use the "instance model" (with a separate set of modeling concepts), otherwise the modeler will use the same set of concepts as when modeling the general access control system class.

For now we will assume that our more particular system description is *not* a description of one of its kind, but still a description that a number of systems will potentially satisfy. One clue that this is the case may be that we have said nothing about the number of access points on each floor.

Thus our task is to model a class MyACSystem satisfying the informal description given above. The underlying assumption is that all the systems that satisfy the more specific system description should also satisfy the general system description. As this is exactly what class specialization of UML2.1 provides, we make MyACSystem a subclass of ACSystem (see Fig. 5).

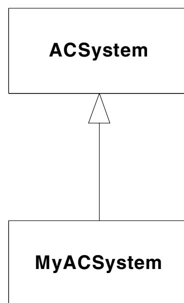


Fig. 5. Inheritance used for the specific system class

The UML 2 language specification states that the special class inherits all the features and constraints of the general class. The question is what the specialization can express regarding the properties defined in the general class.

4.2 Specializing the Type of a Property

With a liberal interpretation of the UML 2 language specification we may in a subclass specialize the *type* of a property defined in the superclass. In our example this means that the ap set of AccessPoints may be specialized to consist of LoggingAccessPoints and that the Authorizers may be SpecialAuthorizers. Such a system is described in Fig. 6.

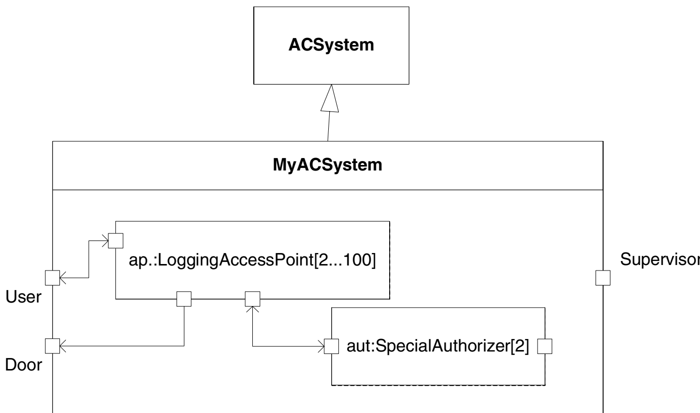


Fig. 6. System with LoggingAccessPoints and SpecialAuthorizers

We could have achieved the same effect by using class redefinition as supported by UML 2. We may redefine the types AccessPoint and Authorizer in the specialization to be LoggingAccessPoint and SpecialAuthorizer, respectively (see Fig. 7).

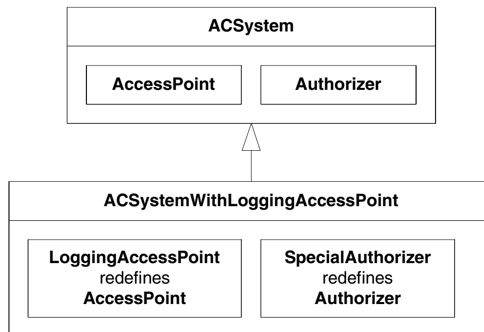


Fig. 7. Using redefinition to achieve the specialized effect

The problem with both solutions is that neither of them can be used to model our specific system described informally at the start of this section. Both solutions can be used to define the particularities of the ground floor, or alternatively the particularities of the top floors. Neither of them can model the particularities of the ground floor *and* the top floors at the same time. We are looking for a mechanism that can model different subsets of a given property set (ap) which is what we think is the only reasonable formalization of the challenge.

Given our specialized definition in Fig. 7 we could add (in ACSystemWithLogging AccessPoint) the necessary information about the top floors. By adding this information these top floors would not be elements of the ap set of AccessPoints. They would represent some completely new property not known before, and that was not the intention. We still want to say that the top floors are indeed a subset of the original set of AccessPoint named ap.

There are means in UML to express that properties of a given set is of different specific subsets. The set ap of AccessPoints may contain objects of Logging AccessPoint and of BlockingAccessPoint or other subclasses, and the way to express it in UML 2 is through special subset constraints.

4.3 The UML Secret of Subsetting Parts

Subset-constraints were introduced in UML 2 to cope with the need to express association specialization [11]. Such subsetting on association ends can be found numerous places in the UML 2 metamodel. This can help us part of the way to express what we want for our particular access control system where the set ap of AccessPoints has been split into one set of BlockingAccessPoints for the top floors and one set of LoggingAccessPoints for the ground floor.

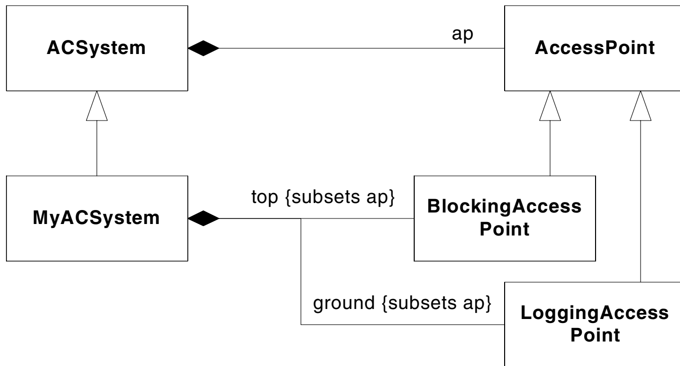


Fig. 8. Subsetting association ends

Fig. 8 shows a legal construction in UML 2 and it does bring us closer to our description goal.

Our answer to the challenge is to apply subsetting to parts. Later we shall use constraints to enhance our approach.

We would like to express that the set ap of AccessPoints should be seen as a number of subsets. These subsets must have a clear relation to the original ap set, but should also describe individual peculiarities.

In fact the notation used in Fig. 8 can also be used directly for all Properties since attributes, parts, and association ends were unified into the Property concept in UML 2. We show how this would look in Fig. 9.

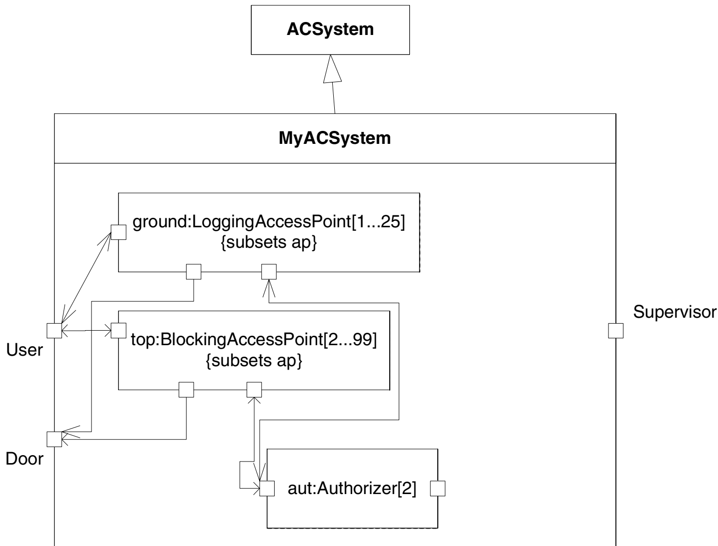


Fig. 9. Naming subsets

4.4 Introducing Notation to Name Subsets

We have in Fig. 9 expressed that ap has two subsets ‘ground’ and ‘top’. The two subsets have the same connectors to other parts as ap had. By applying subsetting also to composite structure parts we have made the model more readable than the model in Fig. 8. Still we find the subset-constraint notation less than satisfactory transparent when it comes to giving new names to subsets of sets already defined in more general notions. Rather we shall use an alternative notation also defined in UML 2: The notation for roles played by part instances of a composite structure is <role-name>’/’<part-name>. It is defined within the notation for StructuredClassifier, but intended to be used when modeling instances (of structured classifiers). We will use this notation separating the subset name from the subsetted name simply by a ‘/’.

In Fig. 10 we illustrate this new notation for subsetting, and in addition we show how the three first points of the challenge can be modeled by adding more constraints to the parts of Fig. 9.

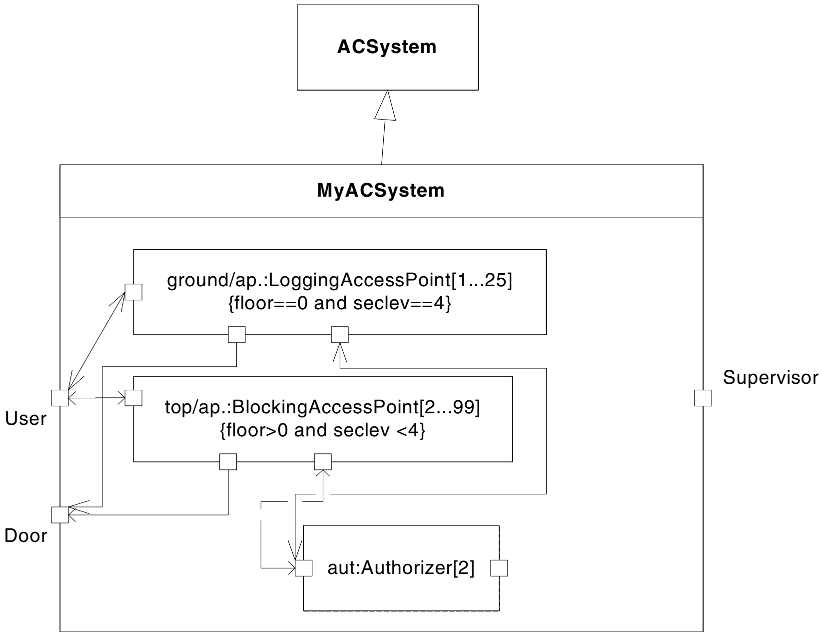


Fig. 10. Named subsets with individual constraints and multiplicities

Note that in this way the notion of role has been generalized: While UML 2 can only have that individual instances play roles, we can have that a subset of instances play the same role. In this example, 'ground' and 'top' are roles played by different subsets of ap.

We have now been able to express that the set ap of AccessPoints is divided in two sets where the ground set is on the ground floor and has high security level. The ground set consists of LoggingAccessPoints (or subclasses thereof) and there are between 1 and 25 of them. The property of the top set is that it contains only BlockingAccessPoints and there are between 2 and 99 of them. The top set access points are not on the ground floor and their security level is not at the very highest.

4.5 Architectural Diversity

We have now shown that the subsets can have different specifications and thereby fulfill the description challenge numbers one to three. The fourth point of the challenge is concerned with the *architecture* of the composite structure. We want to express that the ground set of access points have different connections than the top set of access points. It is actually straight forward to achieve this also since we have two distinct descriptive elements that may have different connections. We must make sure that even though the connections are different, they must still satisfy the general description of the ACSystem (that is how ap is connected in ACSystem).

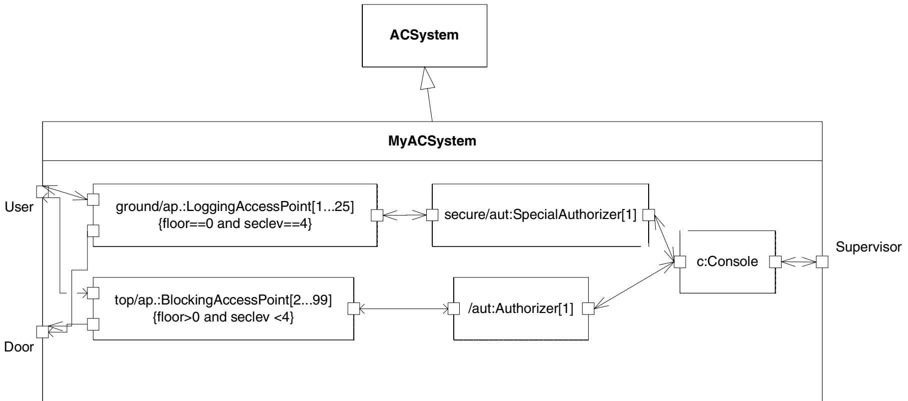


Fig. 11. A specialized composite architecture

We see in Fig. 11 that we have now been able to make a description that is the answer to all four parts of the challenge. We have described configurations based on property subsets combined with a systematic application of constraints.

4.6 Could We Achieve Such Configurations by Other UML Means?

What we have shown in Section 4 is that UML does have mechanisms that can be effectively applied to define configurations and that can be seen as a continuation of the normal UML modeling practices.

Still the diagrams of configurations that we have shown are not at all commonplace in UML models, and therefore we should ask ourselves whether there are other UML mechanisms that could have been applied and that would have done the job equally well.

We have shown that simple specialization cannot cope with both retaining the refinement relation to the original and adding diversity of the description. We need to define subsets. Subsets are constraints and the obvious suggestion is to use constraints all the way (e.g. in a subclass of the original system class). It is simple to declare that everything can be done with OCL, but in fact it is not true.

First, constraints are textual and do not really convey the same kind of message to the readers as does a UML diagram. Adding a bunch of constraints in a subclass is not very readable, especially since this text must also introduce new names. In our case the new names for the subsets must be introduced by the constraints. This is hardly how OCL constraints are normally intended to work integrated with UML. Typically the OCL constraints refer to names defined in the graphical UML model.

Second, even if we accept that new names may be defined in the constraints, the description of new connectors in OCL which will be needed to cope with the last challenge of architectural diversity is clearly far beyond the useful domain for OCL.

Third, such complex use of OCL is far beyond the capacity of the average UML modeler while the need to define configurations is absolutely within the domain of the average UML modeler.

Fourth, the modeler is free to use OCL constraints where graphic means are not sufficient. When new parts are graphically defined (including the introduction of names) the textual constraints will be much simpler, as the constraints then can use these names.

Are there other UML mechanisms that can be applied to define configurations? Some UML modelers may point to the instance model and corresponding graphics. The UML 2 instance model is intended to define an object value where all values of all attributes are described. The UML instance model is described in the Instances Diagram Figure 7.8 in the Kernel package of the OMG recommendation [10] and through the class descriptions of the classes in that diagram. The concepts of the UML instance model are InstanceSpecification, Slot and Valuespecification. InstanceSpecification has a number of Slots, each corresponding to an attribute defined for the class. The concrete syntax is recognized by the underlining of the names of the symbols. For certain examples a configuration may be defined by the instance model, but our example has not described all values and as such cannot be defined by an instance model.

We are tempted to return the argument and will in the following section discuss how our configuration approach can replace the UML instance model.

5 Snapshots, Constructors and Instances

In this Section we shall see that our approach may also naturally bridge the gap between refined structure and refined behavior since behavior is included in the general UML model description.

5.1 Snapshots

Is the system in Fig. 11 a snapshot of the general ACSystem? A snapshot is a term often used to describe a situation in full detail at a specific point in time. The UML specification [10] says "an entity at a point in time (a snapshot)". A strict interpretation of the term will probably conclude that MyACSystem as defined in Fig. 11 is *not* a snapshot since the values are not all fixed. We have not defined how many there are in each of the sets ground and top, and we have not explicitly defined what the floor attribute value should be in all BlockingAccessPoints. Thus it is not a snapshot.

But is this a fruitful definition of a snapshot? We shall argue otherwise. It is obvious that we, through our strategy with subsets and constraints, could get very close to a snapshot.

By using the same simple mechanisms as in Fig. 11 we have in Fig. 12 defined every object (on this level of abstraction) and fixed every value. We notice that we may apply the naming of subsets repeatedly as we have now subsets *zero*, *first* and *second* in MyACSnapshot in Fig. 12. Is this now a snapshot? It is hard to argue against it as the definition seems to contain full information.

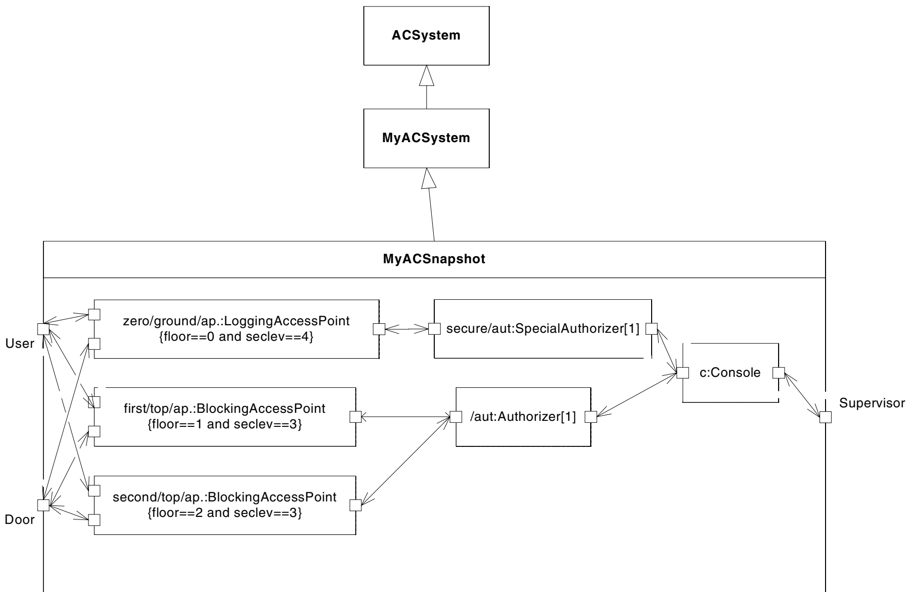


Fig. 12. A snapshot?

Still it is a little unsatisfactory that by showing one single bit of uncertainty the model is no longer a snapshot. If we did not know whether the security level of the second floor was 3 or 2, we could still apply the same mechanisms and state in the constraint that *seclev* of second is either 2 or 3. Is this such a big difference that suddenly the model changes character completely and it is no longer something that can be called a snapshot? For many purposes it is irrelevant whether the security level is 2 or 3 and sometimes the value of something is dependent upon the measuring accuracy. E.g. what is the length of the Norwegian coast?

Is every subclass therefore a snapshot? A snapshot is a description that focuses on the data state of something. The constraints on variables must hold throughout the lifespan of the object. In our snapshot MyACSnapshot it may be argued that the variables will not change, but in the general case a snapshot will instantly transform into another snapshot and the variable values are changed. A traditional snapshot has a lifespan that consists of only one time instant. We believe that whether every value is exact or not is of secondary importance.

But our definition in Fig. 12 of MyACSnapshot may be seen to describe more than only the data. By being based on MyACSystem, it also inherits the behavior specification of the ACSystem. We choose to interpret a snapshot such that the constraints are not constraints of the whole system lifespan, but rather only constraints of the starting time instance. The snapshot shall contain also all necessary behavioral data about the state of the execution. In programs we are talking about the execution stacks and program counter, in modeling we are talking about the current value of the state stack of a nested state machine, or the position within a sequence diagram.

In fact every *generalized snapshot* is the complete description of the *continued behavior* of that system. It is the complete description of all possible continuations of that system from that snapshot.

Since the snapshot has special interpretation of its constraints it is not a simple specialization, and we should rather use a stereotyped dependency than a generalization relation to show the association between MyACSystem and MyACSnapshot. We have shown this in Fig. 13.

5.2 Constructor

A constructor can be seen as a snapshot that describes the initial configuration for a given object of a class. With our generalized notion of a snapshot which includes the behavioral continuation of the system, the constructor becomes an even more expressive concept.

It may be reasonable to restrict constructors such that the behavior status values (program counter, current state and so forth) cannot refer to other symbols than the initial ones.

UML 2 defines a constructor to be any operation of the class that returns (a reference to) a single instance of that class. There may be an instance value of the class associated with the constructor through a create-dependency. That instance value is the default value of the constructor operation. It is not obvious what "default" means here, either the constructor operation returns the instance value associated or it does not. It is reasonable to interpret UML 2 such that constructor operations can only be applied to objects that have just been created by a create action (CreateObjectAction). Given that an instance value is associated with the operation it is reasonable to believe that the newly created object will assume the given instance value.

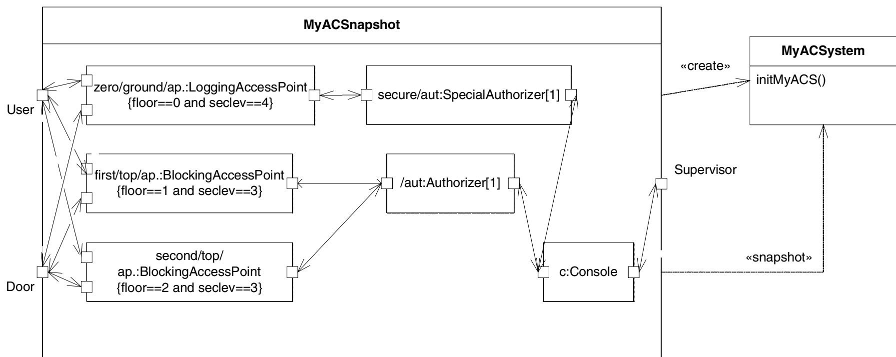


Fig. 13. Constructor similar to UML 2

In Fig. 13 we see that MyACSnapshot is associated with a constructor of MyACSystem, initMyACS(). Application of this would mean that in e.g. an activity diagram there would be a CreateObjectAction referring to MyACSystem and then subsequently an invocation of initMyACS() on the resulting object.

Alternatively UML could take advantage of our more generalized notion of snapshot and use that the constructor MyACSnapshot actually also contains all possible continuations. There is really no need for a separate constructor operation. All we need is to apply CreateObjectAction on MyACSnapshot and then let it run.

5.3 The UML 2 Instance Model and Notation Revisited

As demonstrated above, the notion of configuration (as a combination of property subsetting and constraints) covers the modeling of both snapshots and constructors. The InstanceSpecification part of the UML 2 metamodel was intended for modeling snapshots and constructors, and we therefore reconsider this part of the UML 2 metamodel.

From a metamodel point of view, configuration relies on the existing metamodel elements like Classifier, Property, etc. From a notational point of view we have merely added flexibility to the naming of subsetted properties. We have adopted the notation for subsets from the notation of the instance model, using the slash-notation rather than the more voluminous constraint notation. What are those subsets? Are they still properties? Yes, actually, no new meta-concepts need to be added to the metamodel other than the definition of the «snapshot» dependency.

The UML instance model consists of a set of very general elements: InstanceSpecification, Slot and ValueSpecification, and their main purpose is to refer into the main metamodel of classifiers, properties and associations. Slot has e.g. no name, the idea being that the name is the name of the defining feature.

This can also be seen by the way the slash-notation is described in the UML specification document. The notation section of InstanceSpecification only refers to the section on notation for classifiers, while in the notation section for StructuredClassifier, notation for instance specification is defined, and here the slash-notation is defined as the notation for roles played by subsets of parts:

> *"The namestring of a role in an instance specification obeys the following syntax:*
> *{<name> [‘/’ <rolename>] | ‘/’ <rolename>} [‘:’ <classifiername> [‘,’ <classifiername>]*]*
> *The name of the instance specification may be followed by the name of the part which the instance plays. The name of the part may only be present if the instance plays a role.".*

For the modeling of snapshots and constructors the Instance model adds little information by itself and it is actually superfluous as we have shown earlier in this paper. The main modeling concepts are themselves expressive enough and actually even more flexible than the instance model. The only added feature of the instance as described by the syntax above is that an instance may be typed by multiple classifiers, while a property may only have one classifier as type. It is not obvious what it means to have several classifiers as types.

6 Conclusion

We have investigated how UML 2 may be applied as an architectural description language applying mechanisms that were not in UML 1. We have used a concrete access control example to motivate and illustrate the general notions.

We have shown that UML 2 to a certain extent has the mechanisms needed for the modeling of configurations, by a combination of inheritance with property subsetting and constraints. By applying subsetting and constraints to properties in general and to parts of composite structures in particular, we have described model configurations. The application of these mechanisms ensures a clear architectural refinement relation based on class inheritance.

We have adopted the slash notation for instances playing roles represented by parts and used that for naming subsets of parts: different subsets play different roles.

Through our generalized definitions of snapshot and constructor based on a special «snapshot» dependency we have shown that classes defining configurations may replace the UML instance model. We have also shown that our notions of snapshot and of constructor are more expressive that that of the instance model since also the behavioral continuations are included in our concept.

References

1. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., and Widen, T., *Consolidated Product Line Variability Modeling*, in *Research Issues in Software Product-Lines*, Käkölä, T. and Dueñas, J.C., Editors. 2006, Springer: Berlin Heidelberg New York. 3-540-33252-9
2. Bræk, R. and Haugen, Ø. *Engineering Real Time Systems*. BCS Practitioner Series, ed. Welland, R. 1993, Hemel Hempstead: Prentice Hall International. 398p.
3. Czarnecki, K. and Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*. 2000: Addison-Wesley Professional. 864p.
4. Families, *Families*. 2004, http://www.esi.es/en/Projects/Families/. p. Eureka Σ! 2023 Programme, ITEA project ip02009.
5. Garlan, D., Knapman, J., Møller-Pedersen, B., Selic, B., and Weigert, T. *Modeling of Architecutures with UML*. in *UML2000*. 2000. York, England (panel presentation).
6. Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Object Technology Series, ed. Booch, G., Jacobson, I., and Rumbaugh, J. 2004: Addison-Wesley Professional. 736p.
7. Haugen, O., Møller-Pedersen, B., Oldevik, J., and Solberg, A. *An MDA-based framework for model-driven product derivation*. in *The eighth IASTED International Conference on Software Engineering and Applications*. 2004. Cambridge, USA.: ACTA press 709-714 0-88986-425-X.
8. Haugen, Ø., Møller-Pedersen, B., and Weigert, T., *Structural Modeling with UML 2.0*, in *UML for Real*, Lavagno, L., Martin, G., and Selic, B., Editors. 2003, Kluwer Academic Publishers: Boston. 1-4020-7501-4 p. 53-76.
9. Merriam-Webster's, *Online Dictionary*. 2005: http://www.m-w.com/.
10. OMG. Unified Modeling Language 2.1. 2006, OMG. ptc/06-01-02
11. Rumbaugh, J., Jacobson, I., and Booch, G., *Unified Modeling Language Reference Manual, The (2nd Edition)*. ADDISON-WESLEY OBJECT TECHNOLOGY SERIES. 2004: Pearson Education. 736p.

Modes for Software Architectures*

Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastian Uchitel

Department of Computing, Imperial College London
{dhirsch| jk | j.magee | s.uchitel}@doc.ic.ac.uk

Abstract. Modern systems are heterogeneous, geographically distributed and highly dynamic since the communication topology can vary and the components can, at any moment, connect to or detach from the system. *Service Oriented Computing* (SOC) has emerged as a suitable paradigm for specifying and implementing such global systems. The variety and dynamics in the possible scenarios implies that considering such systems as belonging to a single architectural style is not helpful. This considerations take us to propose the notion of *Mode* as a new element of architectural descriptions. A mode abstracts a specific set of services that must interact for the completion of a specific subsystem task. This paper presents initial ideas regarding the formalization of modes and mode transitions as explicit elements of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. We incorporate the notion of mode to the Darwin architectural language and apply it to illustrate how modes may help on describing systems from the Automotive domain.

1 Introduction

Distributed systems are very complex dealing with a high number of architectures and communicating infrastructures. Modern systems are heterogeneous, geographically distributed and highly dynamic since the communication topology can vary and the components can, at any moment, connect to or detach from the system. As an answer to these requirements, *Service Oriented Computing* (SOC) has emerged as a suitable paradigm for specifying and implementing such global systems. Engineering issues are tackled by exploiting the concept of *services*, which are the building blocks of systems. Services are autonomous, platform-independent, mobile/stationary computational entities. In the deployment phase, services can be independently described, published and categorized. At runtime they are searched/discovered and dynamically assembled for building wide area distributed systems.

All these require, on the one hand, the development of foundational theories to cope with the requirements imposed by the global computing context, and,

* Partially supported by the Project EC FET – Global Computing 2, IST-2005-16004 Sensoria and The Leverhulme Trust.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 113–126, 2006.
© Springer-Verlag Berlin Heidelberg 2006

on the other hand, the application of these theories for their integration in a pragmatic software engineering approach.

At the architectural level, the fundamental features to take into account for the description of components and their interactions include: dynamic reconfiguration, self-organisation, mobility, coordination, complex synchronization mechanisms, multiple communication contexts, and awareness of quality of service. The variety and dynamics in the possible scenarios implies that considering such systems as belonging to a single architectural style is not helpful. These considerations take us to propose the notion of *Mode* as a new element of architectural descriptions. A mode abstracts a specific set of services that must interact for the completion of a specific subsystem task, i.e., a mode will determine the *structural constraints* that rule a (sub)system configuration at runtime. Therefore, passing from one mode to another and interactions among different modes formalize the *evolution constraints* that a system must satisfy: the properties that reconfiguration must satisfy to obtain a valid transition between two modes which determine the structural constraints imposed to the corresponding architectural instances.

This paper presents initial ideas regarding the formalization of modes and mode transitions as explicit elements of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. We consider that the concept of mode helps on closing the gap between requirements and software architectures by using modes as a *scenario-based* abstraction to relate specific use cases with service configurations. Also, we hope that it will permit the verification of reconfiguration correctness (for example, by a predefined set of reconfiguration operations that will carry one subsystem from one mode to another respecting the mode transition specification). It is worth noticing, that the relation between scenarios and modes is not necessarily one-to-one. The idea is that the scenario-based approach can help in understanding how the scenarios and modes can be related providing feedback for the validation of requirements with respect to reconfiguration issues at the architectural level. In particular, we think that modes can help on verifying correctness of coordination policies and deployment issues for self-organising/healing systems.

Our work is funded on the basic ideas from software architecture, as it is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design [16,1]. A fundamental aspect of software architecture is that it is an abstraction that helps manage complexity. To deal with these issues, architecture-based formal modeling notations and analysis and development tools that operate on architectural specifications have been developed (i.e.Architecture Description Languages (ADLs) [15]). Also, notation standards like UML have been proposed to support architectural design [5,6,14]. In this respect, our contribution is the proposal of a (scenario-based) approach that introduces modes as a new first class primitive for languages (with special emphasis on service oriented ones) supporting the

work that has been done in recent years for self-orginising and reconfigurable systems [2,3]. It is worth noticing that modes are already used in other areas such as synchronous programming [13]. In [13], the notion of mode is related to collections of executing states, focusing on behavior. In our case, we are interested in studying modes with respect to system structure. Nevertheless, in future work we plan to study the behavioral side of modes (see Section 4).

In the rest of this paper we present our first ideas on how the notion of mode can be incorporated to an existing ADL (Darwin [9]). The extension is obtained by adding to the language component model an attribute that indicates the component mode in the corresponding architectural instance. We show the usefulness of modes by illustrating how they can help on describing systems from the *Automotive domain*. Also, in Section 4 we will discuss the next steps on possible approaches for the mode based analysis of systems.

Modern automotive systems contain a continuously increasing number of software components that must assist in a big range of operations. These include critical vehicle functions (ABS systems, road repair, etc.) or other less relevant to the vehicle primary function but of importance for nowadays client necessities (for example, road sights, infotainment, etc.). Moreover, these operations are continuously activated and deactivated and component reconfiguration is set up dynamically in a self-organising way. Also, due to advances in mobile technology communication is very complex in automotive software systems, where communication happens within the vehicle (intra vehicle communication), between vehicles (inter vehicle) and between the vehicle and the environment (vehicle-environment). This variety in nature, number, communication and dynamicity makes service-oriented techniques a natural choice for coping with the engineering of automotive systems. We base our work on one of the case studies for the *GC2 EU Project Sensoria (Software Engineering for Service-Oriented Overlay Computers)* [18]. The variety in the possible scenarios in this context provides a interesting testbed for the introduction of modes.

In Section 2 we give a brief introduction of the ADL language Darwin. Then in Section 3 we present modes and apply them for the description of a case study from the automotive domain. In Section 4 we discuss possible approaches for their formalization and how to they can help on the analysis of system properties. Section 5 concludes the paper with final remarks and future work.

2 Darwin

Darwin is a declarative component-based ADL. It supports a hierarchical model, tractable and it is accompanied by a corresponding graphical notation. The overall objective is to provide a soundly based notation for specifying and constructing distributed software architectures [9].

Component Model. The central abstractions managed by Darwin are components and ports. Ports are the means by which components interact. Ports

represent services that components either *provide to* or *require from* other components. A port is associated with a *type*: the interface of the service it provides or requires. Figure 1 shows the textual and graphical representation in Darwin of a filter component which provides an `out` service (filled circle) and requires an `in` service (empty circle), both of type *Stream*.

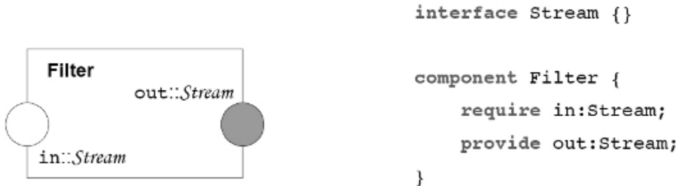


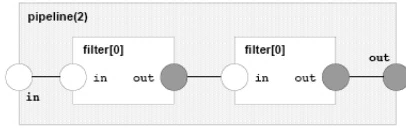
Fig. 1. Filter Component in Darwin

Bindings define a one-to-many mapping relation between provided and required ports. A binding associates the service provided by one component with the service required by another. A provided port may have many required ports bound to it. A required port can be bound to at most one provision. Darwin does not have any special construct to model connectors. Instead connectors, whenever required, are modelled using components and ports.

Components may be nested within composite components to form hierarchical structures. Composite components hide the complexity of the contained structure allowing the specification of the architecture in a varying level of detail. Regular bindings are allowed only between components of the same container. Bindings crossing container boundaries are indirect through port aliases (inward and outward bindings, Figure 2). A port alias is a port in the container component that exports a port from an inner component. Port aliases offer control over the scope of ports in nested constructs while preserving the benefits of nesting. Figure 2 shows a composite component for a pipeline that is obtained from two filters.

Behavioural Specification. The behaviour of components in Darwin is specified both graphically as a Labelled Transition System (LTS) and textually using the Finite State Processes (FSP) notation [12]. The behaviour of an architecture in Darwin is the composition of the behaviours of its individual component constituents, i.e. the parallel composition of the respective LTSs. The resulting LTS can be checked for such properties as the preservation of system invariants or the existence of deadlocks.

Dynamism. Darwins concern in supporting dynamic structures is to capture as much as possible of the structure of the evolving system while maintaining its purely declarative form. Architectural modifications at runtime may cause




```
component Pipeline(int n) {
    require in:Stream;
    provide out:Stream;

    array filter[n]:Filter;
    forall i:0..n-2 {
        inst filter[i];
        bind filter[i].out -- filter[i+1].in;
    }
    bind in -- filter[0].in;
    bind filter[n-1].out -- out;
}
```

Fig. 2. A composite component with an inward (`in` port) and an outward (`out` port) binding

disruption to behavioural aspects of the system such as triggering a deadlock. In [10], it is stated that changes to a systems structure may only be performed when the components involved in the changes are in a quiescent state. A component is considered quiescent if it is not in the process of exchanging application messages with its environment.

Mode Extension. The extension of Darwin with modes is obtained by adding a new attribute to components (boxed names in Figure 3) that indicates the mode in which the component is in the corresponding architectural instance (see Section 3 for the case study details). In the case of basic components, the mode identifies the state of the component. For composite ones, the mode for a composite component is directly related with the modes of its constituents. We assume that each component is in one mode at a time. Ports in gray color mean that the interface port is not "enabled" for binding to another component.

3 Modes for SA: A Scenario for the SENSORIA Automotive Case Study

In this section we introduce an approach to the use of modes at the architectural level by applying it over a case study from the automotive domain. The case study is taken from the *Sensoria Project Automotive Case Study* [18]. The Sensoria case study presents some typical automotive scenarios as they might be available to drivers in the near future. We derive three scenarios which are used to identify the modes and transitions needed to describe the desired evolution of a specific vehicle subsystem. The three scenarios are:

- *Road Sights Scenario:* The driver has subscribed to the dynamic sight service. The vehicles GPS coordinates are automatically sent to the dynamic

sight server. The dynamic sight server searches a sight seeing database for appropriate sights and displays them on the in-car map of the vehicle navigation system.
– *Low Oil Level Scenario:* During a drive, the vehicles oil lamp reports low oil levels. This triggers the in-vehicle diagnostic system. The diagnostic system reports a problem with the pressure in one cylinder head and sends a message with the diagnostic data as well as the vehicles GPS data to the repair server. The service discovery system identifies an adequate repair shop in the area. The repair shop coordinates are sent to the vehicle guiding system to direct the vehicle to the shop.
– *Accident Scenario:* Due to a collision on the route, an automated message is triggered and sent to the accident assistance server that contains the vehicles GPS data. Approaching vehicles are warned about the accident ahead through wireless messages suggesting alternative routes to avoid traffic jams.

In the next section we present a specific vehicle subsystem, the *Route Planning Subsystem*. We will describe the modes for this subsystem and show how they are used to assist modeling architectural instances for the different scenarios we have introduced.

3.1 Route Planning Modes

Our case study is a Route Planning Subsystem (RPS) for a vehicle, which is in charge of providing guiding indications to the driver. The RPS has three possible modes of operation that are specified with Darwin in Figure 3 (described below). To simplify the example we omit port names and port types which are not relevant in this case.

The RPS architecture is composed of three basic components and is shown in Figure 3:

– *Planner (P)*: This component is in charge of determining the solutions for the trip to be done by receiving routing information from the environment and sending it to the User Prompt. Depending on its mode the Planner may send also planning information to the environment for example to guide another vehicle.
– *User Prompt (UP)*: This component receives the information from the Planner and follows and provides the User Interface with the information relative to the real time progression of the trip (i.e. actual position, next turns, etc.).
– *User Interface (UI)*: This component handles the information arriving from the User Prompt and how it is visualized by the Driver. Depending on its mode the User Interface can be reconfigured to connect directly to the environment.

Each diagram in Figure 3 shows the acceptable configurations of the RPS in a specific mode. The interpretation of these diagrams can be dual. In the first place, a composite component mode can be seen as constraining the instances,

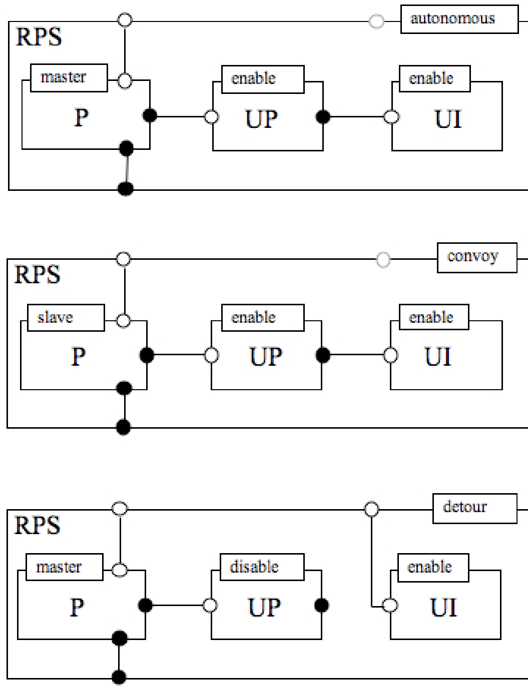


Fig. 3. RPS modes

bindings as well as the modes of its inner components. On the other hand, it can be interpreted as the modes and bindings of the basic components determine the valid mode for the corresponding composite component.

The first RPS mode is the `autonomous mode`. In this mode the driver indicates his destination and the route planner proposes the best route for him. The second mode is the `convoy mode` where the driver has to follow another vehicle from whom is receiving the indications to destination. And finally, the third one is the `detour mode` where the route planner is overrode by an external authority that guides the driver to a detour and in this way avoiding some problem in the route (i.e., an accident or works in the street). In more detail:

– `autonomous`: This mode represents the scenario where the vehicle is planning the trip autonomously (Planner in `master mode`), for example by using the information provided by a *GPS* system in the vehicle or internal information already present in the Planner. As you can see, in this mode the Planner is in `master mode`, and User Prompt and User Interface are in `enable mode`. For User Interface, `enable` is the only mode allowed for this component. In `autonomous mode` we have only intra vehicle communication.
– `convoy`: This mode represents the scenario where the Planner component is guided by the information received by another vehicle in front of him.

The binding to the master vehicle is done through a binding between the required and provided port aliases of the vehicle Planners (see Figure 4). In `convoy mode` the Planner is set to `slave mode` identifying the change in the configuration. In this mode we have inter vehicle communication.

– `detour`: This mode is considered for scenarios like the *Accident* and *Low Oil Level* scenarios. In this mode the User Prompt is in `disable mode` and the User Interface is reconfigured to attend instructions from an external system (Police or Highway Emergency System for example). Also, the Planner maintains its `master mode` as the vehicle may be used by the external system as relaying point for additional planning information to be passed to other vehicles (that switch to `convoy mode`) that are approaching, but are further away and may have more alternatives to choose from. The external system sends instructions directly to the driver redirecting the vehicle to avoid traffic problems or to guide him to the required assistance. In this mode we have vehicle-environment communication.

The point is that in the highly dynamic domain of automotive systems, there are different scenarios that this subsystem must handle that imply reconfigurations changing the architectural style of the system, i.e., the architectural constraints that configurations must satisfy. We use modes to capture these changes of scenarios. A mode is related to a (possible set of) configurations which characterized it. The diagrams in Figure 3 define the RPS component mode types, while in Figures 4, 6 and 7, system configuration instances using the RPS subsystem present different alternatives of these modes. For example, Figure 4 shows a configuration for three RPSs of three cars where the head is in `autonomous mode` and the other two are in `convoy mode`. Figure 6 shows an instance of the `detour mode` where the *Highway Emergency System* is taking control of the planner and sending additional information to the Planner (the ??? mode name indicates that it is not relevant for the example). Finally, Figure 7 shows the `autonomous mode` with its planner connected to an external GPS system.

Figure 5 shows a way of modeling multiple alternatives using $\star$ symbol in the mode attribute (although, other less restrictive alternatives can be proposed) meaning that that component can be in any mode. In this case, Figure 5 identifies the set of possible configuration instances using one RPS in `convoy mode` which can be connected to another RPS in any of the three possible modes. Note that all these configurations follow similar structural constraints.

Modes of a composite component depend on their constituents modes defining mode-based composition. Note that modes not only determine configuration but also coordination and communication mechanisms. In our case study, each one of the RPS modes requires a different type of automotive communication. Figure 8 shows the modes for the RPS and the possible transitions (i.e. reconfigurations) among them. This transition system at the architectural level can help in understanding how the scenarios can be related providing feedback for the validation of requirements.

It is worth noticing, that the relation between scenarios and modes is not one-to-one. For example, a system including the RPS can have a mode for the

Accident scenario that may combine the `detour` with the `convoy modes` of the RPS, where the external system only passes the alarm to the nearest cars approaching the accident zone, which in turn forward the alarm to the other cars behind them by using the `convoy mode`. These may be of help on providing feedback from the architectural level to the requirements by indicating that some scenario may need refinement in more detailed or specific ones.

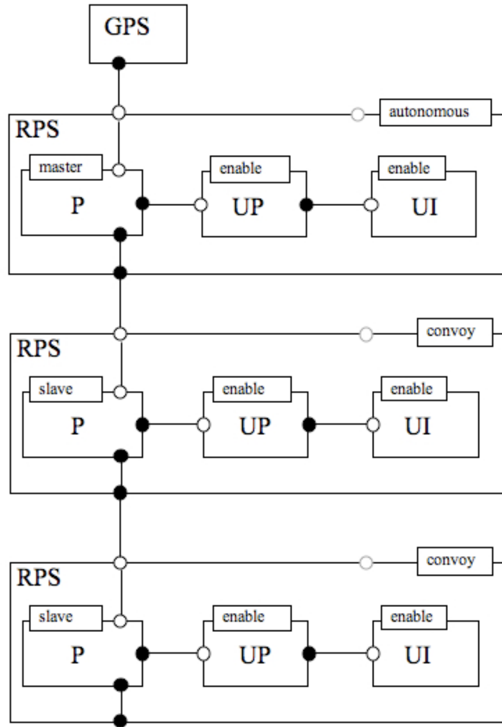


Fig. 4. Convoy mode configuration

With this example we have shown how modes relate scenarios with architectural configurations defining the structure of the system (and type of communication and/or coordination). Also, we can see how modes can help on specifying the possible reconfigurations that are allowed (or not) to handle the relationship among different scenarios (self-organisation), or on guiding the system to take repairing actions in case of some problem (self-healing/reparing). For example, if a system is in a mode where some component fails ending in a non valid configuration (i.e., non valid mode), then depending on the last mode it was, it can determine the resources and operations necessary to reach a valid (maybe the same) mode.

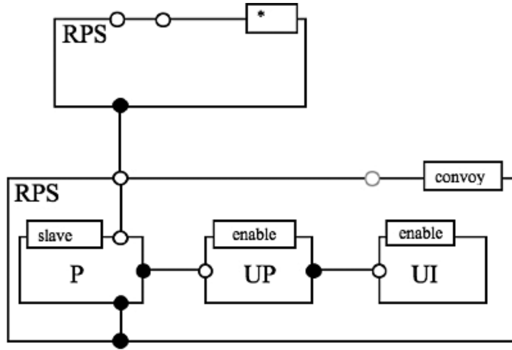


Fig. 5. Unconstrained Leader of a 2-RPS convoy

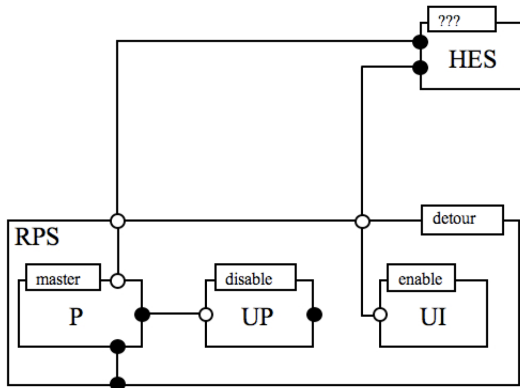


Fig. 6. Detour mode configuration

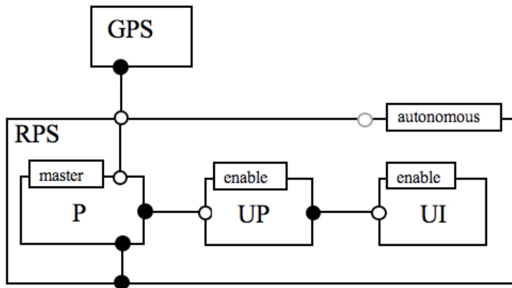


Fig. 7. Autonomous mode configuration

4 Discussing Modes

In Section 3 we have introduced modes over the Darwin language and shows how it is applied to the Automotive domain. We have seen how the mode abstraction

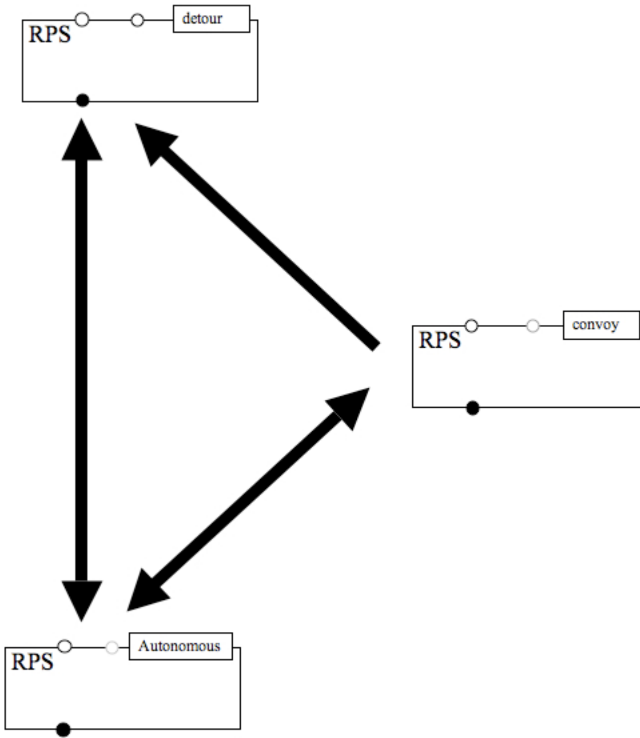


Fig. 8. RPS Modes

can help on closing the gap from requirements to the architectural level by a scenario-based approach which allow to capture the highly dynamics arising in configuration and coordination of service oriented systems. At the same time, this new notion of mode arises interesting questions that our research must try to answer. In this section we will discuss on some alternatives to study in our future work.

Mode Formalization. In Section 3.1 we have incorporated to Darwin, in an *informal* way, the notion of mode as an attribute of component interfaces. Our idea (at least initially) is to obtain a conservative extension of Darwin that respects its original semantics. In fact, in the original language the mode attribute can be modeled by including a dedicated required port to each component. This port is bounded to a special *mode component* whose only function is to indicate the mode in which the component is. This shows that Darwin has enough expressive power to capture the abstraction we needed. Although, incorporating modes as a first class primitive is fundamental to our goals of providing new language abstractions that can cope with the new requirements of global and service oriented systems.

Another future step for the formalization of mode is to study possible extensions of UML. The relation between scenarios and modes, may indicate that extending UML with modes may be useful for relating scenario-based notations with structural ones. One possible way of incorporation modes to UML can be via the definition of specific stereotypes.

Self-* and Reconfiguration Support. Given that self-organisation is fundamental for service oriented computing, another approach we are investigating for the formalization of modes is based on the work done in [4]. The work of [4] presents a software architecture based approach for the specification and verification of self-organising systems. A declarative method is introduced to describe systems underlying software architecture style and then use it to build and maintain system structure during its runtime evolution.

Specifically, based on Darwin component model, system valid structural and evolution constraints are described using the Alloy Language [7]. Alloy specifications consist of definitions of sets, relations among them, and constraints over sets and relations. We plan to extend the Alloy model for Darwin in [4] by adding modes as a new basic element of the model. One benefit of using Alloy is to use the analyser of Alloy models [8] that can be used to generate sample instances that conform to the model or to verify properties of behaviour over a given space of instances. We consider that by incorporating modes to the Alloy constraint model we may be able to identify more clearly configuration issues of self-organising systems.

Another motivation to apply the ideas from [4], is that it provides a method to specify reconfiguration rules over Alloy models as a constraint satisfaction problem. A set of configuration actions is generated when the structure of the system is no longer valid with respect to its architectural description due to either a scheduled change or a failure. Then, the execution of these actions should lead to an architectural instance that remains valid with respect to the style model. In the same way, we can think of using a mode-based approach to reconfiguration and self-organisation which we consider can help to manage the increased complexity is software systems, specially in highly dynamic reconfigurable systems.

Alloy allows us to add structural and evolution constraints to Darwin models obtaining more detailed characterizations of system configurations and their properties. For example, taking the diagram in Figure 5, we may be able to add a structural constraint that only allows `autonomous` and `detour` as the leader mode, exactly identifying the valid configurations with two RPSs. Anyhow, we do not think of Alloy as the final or best approach but as a first step and benchmarking of other languages we plan to study.

Analysis. A main goal for the introduction of modes is in helping on the verification and validation of systems. Our initial approach focuses on the study of techniques for analysing systems structure, but also our future plans will profit from previous experience on scenario-based synthesis of behaviour models from Message Sequence Charts (MSCs) for the identification and validation of modes.

This technique allows the user to specify scenarios in the form of MSCs that capture a desired set of actions, and then to combine them to form one or more state machines (LTS) [17] for deriving the tasks . This allows to use the LTSA tool [11] to analyse models for safety, liveness and temporal logic properties. We consider that the introduction of the mode abstraction in this context can help on reducing complexity and facilitating the validation of correctness between requirements and scenarios derived from modes at the architectural level. It is interesting to think that a component mode may be visible for the state machines that describe the behaviour of the component.

5 Conclusions

In this paper we propose to exploit the notion of modes at the architectural level, where modes are related with specific architectural constraints over the corresponding subsystem configurations. This allows us to assign specific modes to components defining their style and also allows us to specify the interactions and transitions among different modes.

We have introduced the notion of mode by proposing a case study from the Automotive Domain. Our goal is to provide flexible support for the description and verification of complex service oriented systems. We consider that the concept of mode helps on closing the gap between requirements and software architectures by using modes as a *scenario-based* abstraction to relate specific use cases with service configurations.

References

1. Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
2. Garlan,D., Kramer, J., and Wolf, A. Woss '02: Proceedings of the first workshop on self-healing systems. New York, NY, USA, 2002.
3. Garlan,D., Kramer, J., and Wolf, A. Woss '04: Proceedings of the 1st acm sigsoft workshop on self-managed systems. New York, NY, USA, 2004. ACM Press.
4. Georgiadis, I. *Self-Organising Distributed Component Software Architectures*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, January 2002.
5. Hofmeister, C., Nord, R., and Soni, D. *Applied Software Architecture*. Addison-Wesley, 1999.
6. Hofmeister, C., Nord, R., and Soni, D. Describing software architecture with UML. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.
7. D. Jackson. Alloy: A lightweight object modelling notation. Technical report, MIT Lab for Computer Science, July 1999.
8. Jackson, D., Schechter, I., and Shlyakhter, I. Alcoa: The alloy constraint analyzer. In *International Conference on Software Engineering*, pages 730–733, Ireland, June 2000.

9. Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Fifth European Software Engineering Conference (ESEC95)*, Barcelona, September 1995.
10. Magee, J. and Kramer, J. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, ACM Software Engineering Notes, pages 3–14, San Francisco, October 1996.
11. Magee, J. and Kramer, J. *Concurrency: State Models and Java Programs, 2nd Edition*. Wiley, 2006.
12. Magee, J., Kramer, J., and Giannakopoulou, D. Analysing the behaviour of distributed software architectures: a case study. In *5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–245, 1996.
13. Maraninchi, F. and Rémond, Y. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 249–250. Springer Verlag, March 1998.
14. Medvidovic, N. and Rosemblum, D. Assesing the suitability of a standard design method. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.
15. Medvidovic, N. and Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
16. Perry, D. and Wolf, A. Foundations for the study of software architecture. *ACM SIGSOFT*, 17(4):40–52, 1992.
17. Uchitel, S., Chatley, R., Magee, J., and Kramer, J. System architecture: the context for scenario-based model synthesis. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'04)*, ACM Software Engineering Notes, pages 33–42, Newport Beach, CA, 2004.
18. Angelika Zobel. Sensoria deliverable 8.0: Description of scenarios for the automotive case study. 2006.

Integrating Software Architecture into a MDA Framework

Esperanza Marcos, Cesar J. Acuña, and Carlos E. Cuesta

Kybele Research Group, Dept. Computer Languages & Systems
ESCET, Universidad Rey Juan Carlos
Móstoles 28933 Madrid (Spain)
{esperanza.marcos,cesar.acuna,carlos.cuesta}@urjc.es

Abstract. Model Driven Development (MDD) is one of the main trends in Software Engineering nowadays. Its main feature is to consider models as first-class concepts. Model Driven Architecture (MDA), the MDD proposal by the OMG, defines an infrastructure which considers models at three different levels of abstraction, namely Computer-Independent Model (CIM), Platform-Independent Model (PIM) and Platform-Specific Model (PSM). Although it is becoming ever more important, the MDA approach has still some gaps. In our opinion, the lack of an adequate support for architectural design has been, ironically, one of its main drawbacks. MIDAS is an specific Model Driven Architecture for Web Information Systems (WIS) Development. It proposes to model a WIS by considering three different viewpoints, namely Content, Hypertext and Behaviour Viewpoints, which are orthogonal to MDA abstraction levels. In this paper, we propose to extend MIDAS by integrating architectural design aspects. Software architecture is therefore conceived as an crosscutting perspective, which is in turn orthogonal to those three viewpoints. MDA abstraction levels are still considered, and therefore both Platform-Independent Architecture and Platform-Specific Architecture models are defined. This approach, named *Architecture-Centric Model-Driven Architecture* (ACMDA), has several advantages, as it allows architectural design to benefit from the adaptability and flexibility of an MDD process; and on the other hand it extends MDA philosophy by integrating true architectural concerns, effectively turning it into an Architecture-Centric Model-Driven Development (ACMDD) process.

Keywords: Architectural Model, Model Architecture, MDD, Model-Driven Architecture, Architecture-Centric Design, ACMDD.

1 Introduction

In the last few years, Model Driven Architecture (MDA) [20], as proposed by the OMG, has become a leading trend in Software Engineering. MDA is a framework for software development which conceives *models* as first-class elements during system design and implementation; its most important feature is the definition of mappings between these models, which make the automation of *model transformations* possible. Therefore MDA gave rise to a new way of developing software, which is known as

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 127 – 143, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Model Driven Development (MDD) [20, 23]. Among the specific features of the "original" MDA is the concrete grouping of models in three categories, according to their abstraction level; namely, Computation-Independent Models (CIMs), Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs). CIMs are able to model system requirements by defining computer-independent models of the system at hand; these might include domain models, business models, and several others. PIMs are in turn able to model the system's functionality without considering any specific platform, but they are already conceived as computational models. So, PIMs include such models as UML class diagrams, use case models, or statechart diagrams. Finally, systems as described at the PIM level are adapted to a specific platform by means of different PSMs. For instance, the PSM for a database system could either be a relational model or a XML Schema, depending on the chosen technology.

MDA has increasingly become one of the most popular development frameworks in current research, due to the advantages it claims to provide. The supposed benefits of MDA include an improvement in portability, due to the separation of the knowledge of the application from its mapping to a specific implementation technology; an improvement in productivity, due to the automation of this mapping; an improvement in quality, due to the reuse of well-tested patterns and best practices during the mapping of models; and an improvement in maintainability, due to a better separation of concerns and the achievement of a better consistency and traceability between models and the code [8]. However, and unfortunately, the MDA approach has still some gaps that must be filled in as soon as possible. In our opinion, perhaps one of its main drawbacks is that MDA doesn't really take into account the the software architecture design. Although there are already some works which are somehow related both to architectural design and MDA [1, 6, 19], still there is a lot of work to do to obtain a solid and consistent proposal which could achieve general acceptance. In this paper, we present an extended Model Driven Architecture which includes support for software architecture design, in the framework of MIDAS.

MIDAS [17, 25] is a methodological framework for the development of Web Information Systems (WIS). It proposes a Model Driven Architecture supported by two orthogonal dimensions (see Figure 1). In the vertical axis (Y), models are located according to its level of abstraction, using the standard MDA approach: so, they define CIM, PIM and PSM models. In the horizontal axis (X), models are located according to the *aspect* of the system being modeled. In this dimension, we have considered the main aspects of every WIS, namely Content, Hypertext and Behavior. In this paper, we propose a way to integrate a *software architecture* aspect into the MIDAS model architecture. This approach can be considered similar to the one used in [19], where the authors also propose the integration of architectural design into a model driven architecture. However, we disagree in the way this work integrates the architecture into MDA. A deeper analysis of that approach is presented later, after our own proposal is explained (refer to section 3.3).

The integration of a new software architecture perspective into our model-driven architecture has the following advantages:

1. On the one hand, it allows software architectural design to benefit from the same advantages of the MDA approach, as listed above.
2. On the other hand, it causes a change in the development philosophy, turning the MDD approach into an *architecture-centric* MDD, or ACMDD. The use

of a MDA usually implies also that the process follows a MDD philosophy. This means that if the corresponding *model* architecture does not specifically deal with *software* architecture, the development process leaves architectural concerns out. This is, in our opinion, one of the major drawbacks in most of "traditional" MDA approaches. By including software architecture as a part of our model architecture, we are able to change our development approach. Now, the development process is supported by the software architecture. For this reason this approach has been christened as *Architecture-Centric Model-Driven Development* (ACMDD).

The proposal presented here is the result of our previous work on using MDA for WIS development. By applying MIDAS during the development of different WIS's, we identified the need to specifically consider architectural concerns. Different ways to integrate architectural design into our MDA framework have consequently been tested; here, we present the outcome of this work. In this paper we also provide the description of a suitable case study, which applies the ACMDD approach to the development of a specific kind of WIS, namely a web portal which results from the integration of multiple web portals.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the MIDAS Model Driven Architecture as the starting point of this work. Section 3 proposes a Model Driven Architecture that includes the software Architecture Design; this new model architecture becomes the Model Driven Development (MDD) approach in an Architecture Centric Model Driven Development (ACMDD). In section 4, as a case study, we use the previous ACMDD approach to the development of a WIS. Finally, section 5 sums up the main conclusions as well as the future work.

2 MIDAS Model-Driven Architecture

MIDAS is defined as a methodological framework for WIS development. It proposes a model driven architecture supported by two orthogonal dimensions (see Fig. 1):

- Vertical Axis (Y): Models are separated according to their level of abstract-tion. Thus, they are classified as CIM, PIM and PSM models.
- Horizontal Axis (X): Models are separated according to the aspect of the system they model. Considering this, MIDAS distinguises the main aspects of every WIS, namely Content, Hypertext and Behavior. As seen in the Figure, the separation of aspects affects only the PIM and PSM levels. This happens because the CIM level just focuses on domain and business models.

A more detailed description of the MIDAS architecture can be found in [17, 25], about the Content view; [5], about the Hypertext view; [9, 18], about the Behavioral view; [10], about the CIM level; and, finally [24], about our development tool.

The main problem found when applying this architecture was that the software architecture design was not being considered. MIDAS was designed for WIS and then it implies the most common architecture for a WIS, which usually consists of three layers: an user interface layer (which corresponds to the MIDAS hypertext aspect), a persistence layer (which corresponds to the MIDAS content aspect) and a business

layer (which corresponds to the MIDAS behaviour aspect). Therefore, to develop a standard client/server web-based database system is easy applying MIDAS.

However, though the three layer architecture is probably the most popular choice for WIS's, it is not the only option. We found this problem when applying MIDAS to the development of any WIS which uses a platform other than the classic client/server style; a good example could be a WIS built on top of a Service-Oriented Architecture (SOA) support. We have also found the same problem when trying to apply MIDAS to more complex WIS's than the classic Web Database, in particular when the website must include some complex functionality, such as on-line purchases. This is also the case of the example presented in section 4, which consists of a web portal build created by the integration several different web portals.

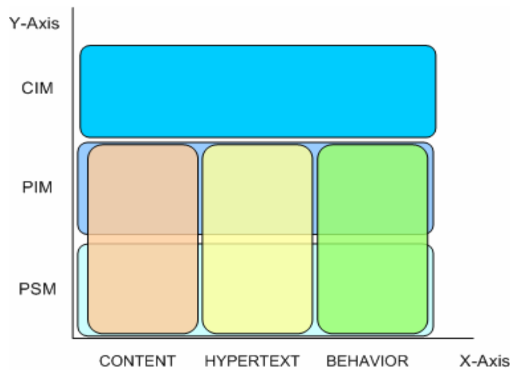


Fig. 1. Simplified MIDAS Architecture

In every real case we have developed with MIDAS which has required a different software architecture than a basic client/server or three-layered architecture, we have always found the same problem: What happens with the architectural design? When should we tackle the software architecture design? How does it affect the rest of the WIS development process? To be able to answer these questions, we have included the architectural design in the MIDAS model architecture, as an aspect located on a new dimension, orthogonal to the existing two we have shown in Fig. 1.

3 Introducing Architecture into the MIDAS MDA

There are, of course, different ways of including the software architecture in a model driven architecture. We'll firstly explain our proposal to later discuss our reasons for choosing this solution among different alternatives.

3.1 Architectural Model vs. Model Architecture

We should perhaps begin by clarifying our choice of terminology. There are several overlapping terms, which are used in more than one sense; in fact, the work brings together the tradition of two different communities (those of software architecture and

model-driven development); though they both share a common Software Engineering background, there are still some differences.

In particular, the central term, *architecture,* is used here to refer to two different concepts. First, when using the isolated term *architecture*, we usually refer to *software architecture*, that is, “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” [14]. From this point of view, architecture is an abstraction of the structure of a system, therefore an abstract entity. But this entity can be made concrete by means of an explicit representation, which is also informally referred to as “the” architecture; but it has also the classic name of *architectural description*, usually provided in terms of an Architecture Description Language (ADL). Of course, an ADL is just a particular kind of domain-specific language which provides a certain *model* of the system. Therefore, the architectural description is also the *architectural model* of the system. Therefore, in this paper, we use the terms *architecture* and *architecture model* interchangeably.

The role of architecture in system development has always been important, though sometimes disminished or even forgotten by methodology definitions. In recent years, however, several methodologies and process definitions have given architecture a central role. These *architecture-centric* [4, 12] processes consider architecture as the main artifact in the software development process. It does not only provide the basic skeleton for the system, but also the basic guide which drives the development itself. Every refinement or extension step during the development process is preceded by a refinement or compositional step in the architecture.

On the other hand, a different tradition is that of *model-driven* development, as described in e.g. [23]. This well-known approach defines *models* as the basic artifacts in software development. The idea is to define the system as a set of models which are progressively refined and composed to get a defined picture of the system, which can ultimately be transformed into some executable form. The approach as a whole is sometimes dubbed as *Model-Driven Engineering* (MDE); meanwhile, the expression *Model-Driven Development* (MDD) is reserved to refer to any software development process in which models play the central role. The best known MDD approach is that of OMG’s Model-Driven Architecture (MDA) [20]. In fact, the term *model-driven architecture* has itself a generic meaning: it refers to the *structure* (or the definition of the structure) of a particular MDD proposal. Specifically, the list of models to create, where should they be located, and the relationships between them. However, OMG’s MDA has acquired such relevance that it has taken over the most generic meaning, and now the expression is mostly used to refer to this particular proposal.

In the context of MDD, we can use also the term *model architecture* (not to be confused with the architectural model described above). It refers to *the structure of the set of models* we are dealing with during the MDD process, and the term can also be used to refer to the set of models itself. The model architecture is not the generic definition of the structure provided by the MDD definition (in fact, the model-driven architecture we just defined, which provides the distinction between PIM and PSM models, for instance), but the *concrete* set of models we are dealing with at every step and every *moment* during the MDSD process.

Finally, recent work has brought together these two approaches, by including an architecture model into the model architecture, and giving it a central role in the software development process. These mixed approaches are known as *architecture-centric model-driven development* (ACMDD) processes [16]. There can be several different kinds of ACMDD, depending on the way the architectural model is defined, and how it is used in the model-driven process.

The role of the architectural model is described as a part of the generic structure of the ACMDD process, which is referred to as the *architecture-centric model-driven architecture* (ACMDA). Here we recover the generic meaning of the "MDA" part of the acronym, which it is not used to refer to OMG's approach (even when our own proposal uses some MDA terminology, like the PIM-PSM distinction). We also use the ACMDA acronym to refer to the proposal we describe in this paper, for the reasons we expose in section 3.4.

The work by Manset *et al* [16] is among the first ACMDD process proposals. The work described in this paper defines the ACMDA extension of our previously existing MIDAS development process for WIS [17]. They describe different approaches to the ACMDD concept, as detailed in section 3.4.

The architecture-centric MIDAS model is a ACMDA in the sense that it describes a certain architecture which implements an ACMDD process. But our use of the term "ACMDA" is also justified by the fact that, in our particular proposal, *architecture* is used in both senses of the word, as the acronym suggest. We explain this in more detail again in section 3.4, when discussing its meaning.

3.2 Software Architecture as a New Dimension in a Model Architecture

After different attempts to include the software architecture design in the MIDAS model architecture, finally, we have decided that the software architecture has to be a different aspect, but in a new dimension. This new dimension will be orthogonal to the level of abstraction and to the aspect ones (see Fig. 2). This is because the software architecture design is related with each of the other system aspects (content, hypertext and behaviour). Moreover, the software architecture will determine which models of each aspect we'll need for modeling a specific WIS. This way, our MDD process gets transformed, as already explained above, into an *Architecture-Centric, Model-Driven* development process, under the ACMDD umbrella. This way, we obtain one of the benefits outlined in the introduction, which is provided by the inclusion of software architecture into model driven architecture (refer to section 1).

For example, suppose a very simple Web site that only has some static Web pages. The architecture of this WIS is very simple; it has just one layer corresponding with the user interface. This simple architecture will indicate to the developer that he just needs to model the hypertext; so, he'll use only the models required for the hypertext aspect. This is a very simple example, but it can illustrate the implications of the software architecture in the model driven development. A more complex architecture, of course, will imply using models of different aspect. For instance, a WIS for electronic purchase, with dynamic pages, implemented with SOA, will require models of the hypertext, content and behaviour aspects. The specific models needed for each WIS will be determined by the software architecture design. In section 3.4 we'll better illustrate this idea by means of a case study.

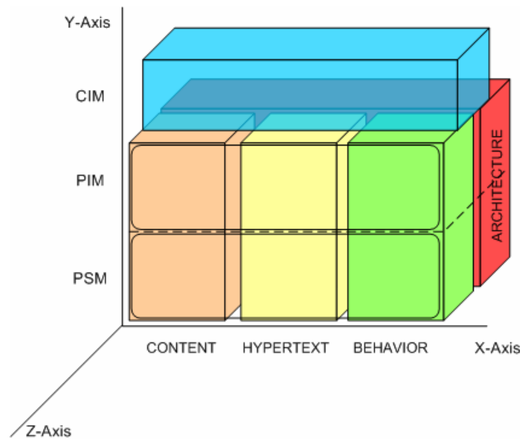


Fig. 2. MIDAS model architecture, including the software architecture model

As you can see in 0 the software architecture aspect, as well as the other aspects, affects only to the PIM and PSM levels. This is because, modeling the architecture, as well as modeling the content, hypertext or behaviour, fall down into the solution space. At CIM level we model the application domain and the architecture doesn't depend on the domain; it's part of the solution.

Finally, we can appreciate how the software architecture can be designed at PIM and at PSM levels. In fact, it is possible, and recommendable in a MDA framework, to model the architecture independently of the implementation platform. In this way, the architecture model, as well as any other model of the system, will be transformed into different PSM models, depending on the chosen specific platform. In this way, we get the other benefit posed in the introduction derived of the inclusion of the software architecture in a model driven architecture (see again section 1).

3.3 Discussion: Alternate Approaches

We have studied, of course, others ways of including the software architecture into a model architecture, but for different reasons why we have been rejecting them.

- **Software architecture as a new aspect in the same dimension that the content, hypertext and behavior ones**
  As we have argued above, including the software architecture as a new aspect, but in the same dimension (see Fig. 3), places the architecture in the same position to the other aspects with regard to the development process. However, in our opinion, the architecture is not just one aspect more of the system but the aspect that has to drive the development process. Being an orthogonal aspect, it allows managing the process, indicating which aspects and models are needed according to the software architecture design.

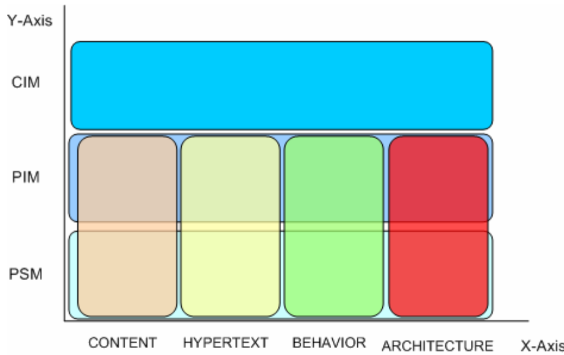


Fig. 3. Including the software architecture as a new aspect

- **Software architecture as a new level between the CIM and the PIM ones**
  We've found two reasons for not accepting this solution. On the one hand, because the software architecture design falls down into the solution space, and the solution space starts at the PIM level; the CIM level describes the problem space instead. On the other hand, because an architecture design defined between the CIM and PIM levels would be a *high level* architecture design, leaving in this way no pace for the design of a more specific architecture model.

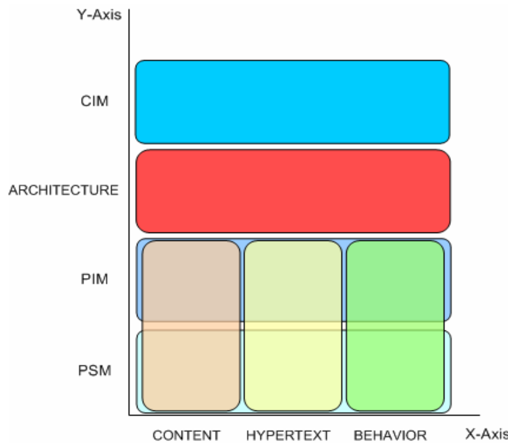


Fig. 4. Including the architecture between CIM and PIM levels

Apart from the already provided process-oriented reasoning, there is another reason which is obvious in the light of the above figure (Fig. 4). Consider that our MDD proposal, MIDAS, has consciously chosen a concern-based approach; the different *aspects* in a WIS development have been explicitly separated, following again the well-known principle of SoC [7]. Though this choice was made due to the specific features of a WIS, this is an established and useful practice, which is not necessarily linked to systems of this nature.

But if we had chosen the approach in 0, the architecture model would be the only one which is not concern-oriented. This defines an intrinsic asymmetry, which perhaps makes some sense in an architecture-centric approach, but it does not seem to be an reasonable solution in our context.

- **Software architecture as a new level between the PIM and the PSM ones**
  This proposal is the option chosen by [19], and in fact there are some reasons to support it. But we have also found two reasons for not accepting this solution in our context. On the one hand, if we have not the architecture design until completing the PIM models, the software architecture couldn't drive the development process. So, we'll have a MDD approach instead of an ACMDD one. Designing the architecture after the PIMs of the rest of the aspects, prevent the architecture to be the guide that allows choosing the aspects and models required for a specific application. On the other hand, what will be the level of abstraction of the architecture designed at this level? Will it be an independent platform architecture? Or should it be already a platform dependent one? Note, that this approach doesn't allow modeling the architecture at different levels of abstraction (PIM and PSM), missing in this way the benefices derived of a MDA approach for the software architecture.

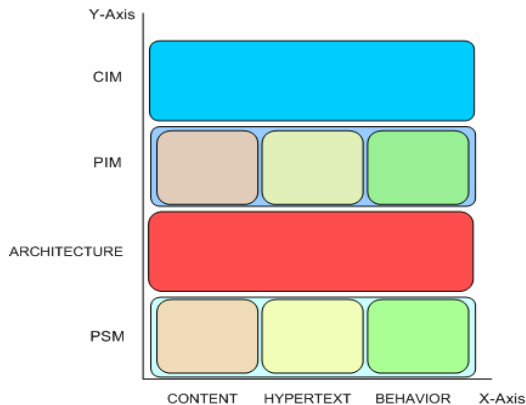


Fig. 5. Including the architecture between PIM and CIM levels

3.4 The Role of the Architectural Aspect: MIDAS as an ACMDA

So far, we have discussed the question in terms of Model-Driven Engineering and its influence over software development processes; but a detailed discussion in the light of existing Software Architecture description theory is still missing. Therefore, the purpose of this section is to fill in this gap, at least in part, by providing some initial reflections about our proposal from the architectural point of view.

As we'll see, the discussion is perhaps deceiving, as it appears to be more complex than it actually is. The reason is the persistent tension between what the models *seem* to be, and what they actually *are*. To decide this, we must consider the two meanings of the word *architecture* (first as architectural model, then as model architecture), as

already exposed in section 3.1. And then, we must define the role the former one plays in the context of the latter one.

In our two-dimensional model[1], the levels of abstraction from MDA (CIM, PIM and PSM levels) define the dimension which represents *evolution* in the development process, and how models are refined into more detailed stages. As already exposed, architecture is a model itself, and therefore it is also inside the MDD process. As it belongs to the solution space, we don't consider a CIM-level for architecture, but there are indeed PIM- and PSM-level instances of architecture. These models, namely *platform-independent architecture* and *platform-specific architecture* (see Fig. 6 and Fig. 7 to get an example) represent different stages in the evolution of the architecture model, and therefore the role of architecture is *orthogonal* to this dimension.

The three basic models in MIDAS (content, navigation, behaviour) are indeed *architectural views*, in the sense of [14], and in fact they can also be adequately conceived as *architectural aspects*, in the sense of [7]. This provides a comfortable framework to deal with them, and we could consider if the architectural model can be considered as another aspect. In fact, it also provides a *view* of the system, a complete description of it which uses a different viewpoint. But that is not a good approach. As we have already explained in the first option in section 3.3, this contradicts the fact that the architecture is the driver of the MDD process, making it architecture-centric. Also, architecture is strongly related to every other view.

The latter sentence hints towards another idea. Architecture can be seen as a *crosscutting* aspect, not just simply a symmetric view. This is quite similar to the distinction between *views* and *perspectives* in the work by Rozanski and Woods [22]. In this sense, architectural model is a *perspective*. There are some more perspectives in the MIDAS model, as it is the case with the semantic view included in 0. We will not focus on it here, as it is better described in previous work [1, 2].

Architecture is therefore a *perspective*, not an view. But it does not play the role of *just* a perspective, as it is in charge of controlling the process. That is the reason why our approach is an ACMDD process. Then, apart from providing a description of the system (which is inherent on it being a model), it is controlling the way in which *the elements from the other views are distributed*. In fact, it has to decide and explicitly state *which views are instantiated* during the ACMDD process, and which of them are not. Therefore it serves also as a meta-model for the ACMDD process itself. And this meta-model quality is lacked in what the rest of the models. So, architecture is still a perspective, but it is also something else.

We might compare to approach to the one in some other ACMDD processes, in particular the already mentioned proposal by Manset *et al* [16]. This approach is architecture-centric because it defines the model-driven development process on top of a powerful architecture description language and platform, Archware. The whole development process is supported by a formal model of development, which has an architectural nature: everything is conceived on top of the architecture.

Then, the approach by Manset is an ACMDD, because every model in the model architecture is an architectural model (or it is supported by an architectural model); therefore it is an architecture-centric process because the model driving the process is

[1] It does not need to be 2-dimensional, and in fact there is a 3-dimensional variant of this model; but this is not relevant for the discussion at hand, and it is not commented here.

an architecture. Our approach is similar but different. In our proposal, architecture is a high-level description of the system which guides the rest of the development process, and plays the central role in our model architecture. In fact, our architecture viewpoint is the “map” to provide the structure of the model architecture, deciding which views are instantiated and which of them are not. Then our approach in also an ACMDD, but it is quite different from Manset’s.

In summary, both Manset’s [16] proposal and ours are ACMDDs, though using a different perspective, and they are even compatible.

In our proposal, the architectural model (the first A in the ACMDA acronym) is a model in the model-driven process, which takes the form of a perspective which is, also, located in an orthogonal dimension. It is indeed the model which drives the development process, and that’s why it is architecture-centric. But it is also the one which defines the structure of our concrete model architecture, and decides which models are considered and which are not. So it is also the “metamodel” for the MDD process, in summary the MDA for it. And therefore, the *same* model plays also the role of the second A in the ACMDA acronym. This is the reason why we justify our use of the term ACMDA to refer to this specific extension of MIDAS.

4 Case Study: Applying the ACMDA Approach

This section elucidates the use of the proposed ACMDD by means of a case study. The case study deals with the development of integration web a portal, that is a web portal which integrates information and data from different underlying web portals. In order to develop integration web portals, we have defined software architecture which was previously presented in [1,2].

Section 4.1 introduces the platform-independent architecture and Section 4.2 introduces the platform-specific deployment of such architecture using Semantic Web Services implemented by WSMO (Web Services Modeling Ontology) as the specific platforms. Finally section 4.3 discusses how the advantages of ACMDD are applied to this case study.

4.1 Architectural Modelling at PIM Level

The proposed platform-independent architecture shown in Figure 6 aims to offer a service-based platform-independent architecture for web portal integration.

The services involved in the architecture can be split in two groups; *core services* group, depicted with a dashed line in figure 6, and the *access services* group. Each service has a service description, which is used to advertise its capabilities, interfaces, behaviour, and quality. Publication of such information about available services provides the necessary means for discovery, selection, binding, and composition of services.

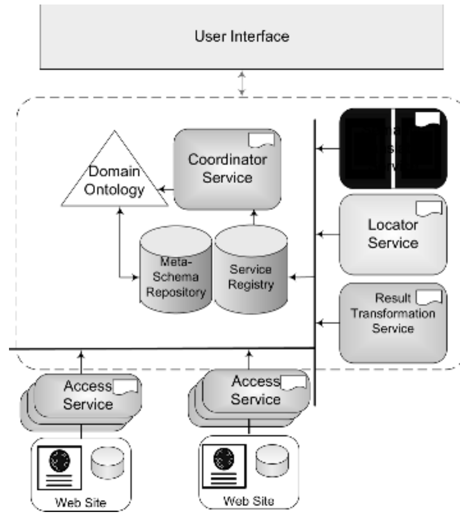


Fig. 6. Platform-Independent Architecture (Architectural PIM)

The Core Services are the mandatory services which provide all the tasks necessary to achieve integration. The Access Service Group is composed of the services that give access to the data or services provided for Web portals. They are a special type of Web services that must implement data retrieval and service accessing operations. There are two main tasks that have to be performed by the access services. First, they provide the needed capabilities for data extraction from the sources. These capabilities have to deal with two main tasks: to automatically perform navigational sequences to access the pages containing the required data and to extract the desired information from the retrieved pages. By accessing the information sources by means of access services, the information sources retain their autonomy (the owner of the sources is different from the owner of the integration system). The data source owner retains control over the shared data, and it decides which data are shared. Second, they must allow access to the specific services offered by the Web portals.

4.2 Architectural Modelling at PSM Level

We have decided to implement the above described architecture by means of semantic web services (SWS). There are several proposals for SWS description; among the best known we can find WSMO (*Web Services Modeling Ontology*) [12], OWL-S[21] and WSDL-S [3], but there are some others. WSMO and OWL-S are the most prominent proposals. Nevertheless, we have chosen WSMO in this case, mainly because it was specifically developed to be used on integration, because it includes an execution environment called WSMX (Web Services Execution Environment) and because WSMO is now the SWS initiative with the most intensive research activity.

WSMX enacts as a middleware to ease the execution of SWS described using WSMO, that way WSMX could be considered as black box component in architecture of the PSM level. 0 depicts the architecture at PSM level; this is the platform-specific

architectural model, or *platform-specific architecture* for short. This approach to architecture is different from, but somehow similar to, what many people in the software development community understands as "architecture", namely the choice of technology, components and services. But this related notion of technical architecture, or *tarchitecture*, is conceived from a different point of view [13], as opposed to non-technical viewpoints; instead of that, the *architectural PSM* in our proposal is just the last stage in the system's refinement, and consequently the one which has more specific and technical details.

Therefore our model-driven approach to architectural refinement provides a way to distinguish between differently detailed architectural descriptions, sometimes even models of the same system. Now we are able to define relationships between these architectural models, in an unprecedented way. It has been known for long that two different systems can share the same architecture; but now the level of abstraction can be taken into account, so that we are now able to say that two systems share the same *PIM-level* architecture, but have different *PSM-level* architectures. Of course, this is implying a smaller distance than the existing between two systems with two different architectural PIMs, even when they finally use the same low-level platform.

WSML descriptions of Web Services, ontologies, mediators and goals are sent to WSMX for compilation. The user interface, that is, the integration web portal creates a service requirement in the form of a WSML message consisting of a goal that describes what WSMX should execute. The goal is then sent to WSMX for execution. When WSMX receives the WSML message with a specific goal, it discovers the WS that best matches that goal, mediates the service requirement data following mapping rules between the source format ontology and the ontology of the discovered WS, and finally invokes it, providing the data to it in the concepts and formats it expects.

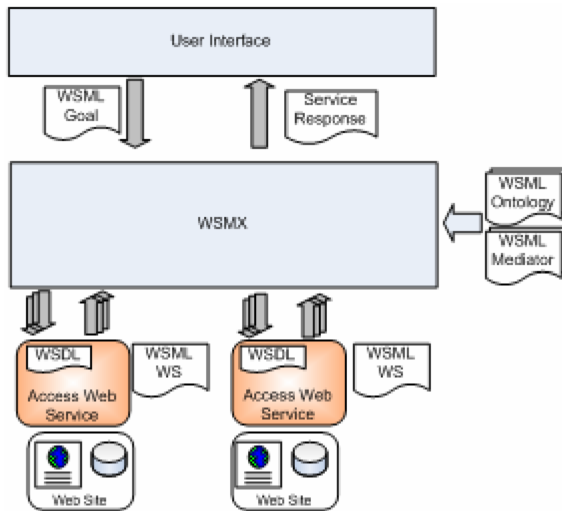


Fig. 7. Platform Specific Architecture (Architectural PSM)

Note that, in this PSM level architecture, the access services are transformed into SWS, the services description used at PIM level are split in two different descriptions, a syntactic one materialized by a WSDL document and the semantic one, that is a WSML document. It is supposed that these Web Services are already developed by the web portals that are going to be integrated. Nonetheless the semantic descriptions should be developed.

4.3 Improving the Development Process Using ACMDD

As was previously stated, the inclusion of the software architecture in the model driven architecture yield two main advantages. On the one hand, it allows software architecture design to profit from the same advantages, explained above, of the MDA approach. This fact is clearly noticed in the case study proposed. To get a web portal integration architecture that can be used in different scenarios, it is important to address the architectural design following a MDA fashion. At PIM level the design of a platform-independent architecture is addressed. That way it is possible to define all the abstract components needed in every integration system from a conceptual point of view, allowing an architectural design free from the technological constraints existing at design time. Next, after some design decisions, all the required components for web portal integration could be implemented by different specific technologies depending on specific needs, available technologies, etc. at PSM level.

Even though the case study tries to propose an integration framework for web portals, note that the architecture depicted in Figure 6 is based on a service- oriented paradigm, so it can be implemented in a web environment just as well as any other one. For that reason we consider this architecture as platform independent one.

On the other hand, to include the software architecture in the model driven architecture will determine which models should be boarded in each particular WIS development. To explain this advantage on our case study, first, Fig. 8 depicts an extended model architecture for MIDAS, where just the aspects of Content, Hypertext, Behaviour, and Semantics are provided for reasons of clarity.

The Semantics view (already briefly mentioned in section 3.4), is an additional view in the MIDAS model, which has not been explained before due to its particular nature, and to the fact that it does not affect the current discussion. In fact, it plays the role of a *crosscuting* view and therefore it is easier to conceive as a perspective; but we will not discuss this view in the rest of the paper, and here it is just provided to have a complete example. Reader is referred to [2] for more details.

Taking up again the platform independent architecture depicted in figure 6, from this picture we can clearly choose which models of the PIM level of the other aspect should be used to model each component. For example, The user interface component is addressed at PIM level of the Hypertext aspect, using the Extended Slice Model and the Extended Navigation Model. The Service Registry and the Meta-Schemas Repositories modeling is boarded by the Conceptual Data Model at PIM level of the Content aspect. The functionality of the Core Service Group is analyzed by means of the models proposes by MIDAS at the PIM level of the Behavior Level.

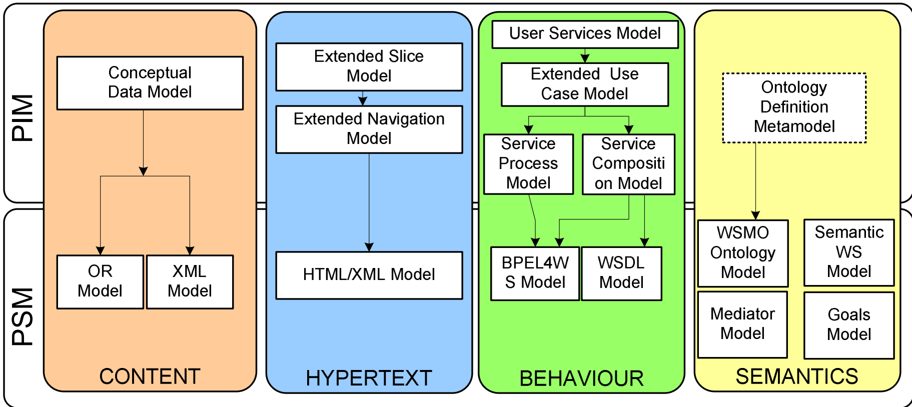


Fig. 8. Model Architecture in MIDAS

The same occurs with the platform specific architecture depicted in figure 7. To model the User Interface component, we need to transform the Extended Navigation Model obtained at PIM level into an HTML or XML model. The ontologies, goals, web services, and mediators descriptions in WSML are modeled by the models at PSM level of the semantic aspect of MIDAS. However, note that the we do not use the models at the PSM level of the content aspect, that is because at PSM level we use WSMX, and WSMX already provide their own structure for the ontology, services, and meta-schema storage.

5 Conclusions and Future Work

Currently, one of the most important trends in software development is related to model driven architectures. However, the basic MDA approach has still some gaps, as it mostly ignores the architectural design. This gives rise to two main problems:

- On the one hand, the software architecture design is unable to be supported by a MDA approach itself.
- On the other hand, architecture is left out of the development process.

With the aim to solve these problems, we have proposed to include the software architecture design as a new aspect in a MDA architecture. This aspect is orthogonal to other aspects of the system (such as Content, Hypertext or Behaviour) and also considers the standard PIM and PSM levels of abstraction. Then, we have discussed the way in which this solves the two problems mentioned above, namely:

- Software architecture will be considered both at PIM and PSM levels, then achieving the benefits of a MDA approach.
- Software architecture can now guide the development process, consequently turning the original MDD approach into a ACMDD approach.

This proposal is the result of our previous work in MDA for WIS development and we have illustrated it by means of a real case study. In the future we plan to apply the

MDA-based framework proposed here to other WIS's with different architectural styles. Currently, we are applying it in the context of SOA-based plaftorms and on top of a Grid architectured. These case studies will allow us to test the real applicability of the extended MIDAS model architecture, and also to refine it, if necessary.

Acknowledgments

This research has been partially funded by the Spanish Ministry of Education of Science in the framework of the national Research Projects GOLD (MEC-TIN2005-00010) and DYNAMICA, subproject PRISMA (MCYT-TIC2003-07804-C05-1).

References

1. Acuña C., Gómez J.M., Marcos E., and Bussler C. A Web Portal Integration Architecture based on Semantic Web Services. *Proc. of 7th Intl. Conf. on Information Integration and Web based Applications and Services* (IIWAS 2005), pp. 174-185, Malaysia, 2005.
2. Acuña C., Gómez J.M., Marcos E. and Bussler C. Towards Web Portal Integration through Semantic Web Services. *Proceedings of 1st Intl. Conf. on Next Generation Web Services Practices*, Seoul (Korea), IEEE Computer Society, 2005.
3. Akkiraju, R., Farell, J., Miller, J.A., Nagarajan, M., Sheth, A. and Verma, K. *Web Service Semantics: WSDL-S*. http://w3.org/2005/04/FSWS/Submissions/17/WSDL-S.htm, 2005.
4. Broy, M. Model Driven, Architecture-Centric Modeling in Software Development. In *Proceedings of 9th Intl. Conf. in Engineering Complex Computer Systems* (ICECCS'04), pp. 3-12, IEEE Computer Society, April 2004.
5. Cáceres, P., Marcos, E., and De Castro, V. Navigation Modeling from a User Service Oriented Approach, *Proc. of 3rd Biennial Intl. Conf. in Advanced in Information Systems*, Lecture Notes in Computer Science 3271, pp. 150-160, 2004
6. Colombo, P., Pradella, M., and Rossi, M. A UML 2-compatible language and tool for formal modeling real-time systems architectures. *Proc. of 21st Annual ACM Symposium on Applied Computing (SAC'06)*. April 2006. ACM, Press.
7. Cuesta, C., Romay, P., de la Fuente, P. and Barrio-Solórzano, M. Architectural Aspects of Architectural Aspects. In Morrison, R. and Oquendo, F. (eds.), *Proc. 2nd European Workshop on Software Architecture (EWSA'04)*, Lecture Notes in Computer Science 3527, pp. 247-262, Springer Verlag, 2004.
8. Czarnecki K and Helsen S. Classification of model transformation approaches. In Bettin, J., van Emde Boas, G., Agrawal, A., Willink, E. and Bezivin, J. (eds), *Proc. 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture,* Anaheim (CA), October 2003.
9. De Castro, V., Marcos, E. and López Sanz, M. A Model Driven Method for Service Composition Modeling: A Case Study. *International Journal of Web Engineering and Technology*. Accepted for publication, 2005.
10. De Castro, V., Marcos, E., and Wieringa, R. *From Business Modeling to Web Services Composition: A Web Engineering Approach*. Submitted to 25th Intl. Conf. on Conceptual Modeling (ER2006). Tucson, Nov. 2006.
11. Graw, G., and Herrmann, P. Generation and Enactment of Controllers for Business Architectures Usign MDA. In Oquendo, F., Warboys, B. and Morrison, R. (eds.), *Proc. 1st European Workshop on Software Architecture (EWSA 2004)*, pp. 148-166, Lecture Notes in Computer Science, 3047. Springer Verlag, May 2004.

12. Gomaa, H. Architecture-Centric Evolution in Software Product Lines. *ECOOP'2005 Workshop on Architecture-Centric Evolution (ACE'2005)*, Glasgow, July 2005.
13. Hohmann, L. The Difference between Marketecture and Tarchitecture. *IEEE Software*, vol. 20(4), pp. 51-53, 2003.
14. IEEE AWG. *IEEE RP-1471-2000: Recommended Practice for Architectural Description for Software-Intensive Systems*. IEEE Computer Society Press, 2000.
15. Lausen H, Polleres A. and Roman D. (Eds). *Web Service Modeling Ontology Submission*. Retrieved From: http://www.w3.org/Submission/WSMO/, 2005
16. Manset, D., Verjus, H., McClatchey, R. and Oquendo, F. *A Formal Architecture-Centric, Model-Driven Approach for the Automatic Generation of Grid Applications*. Proc. 8th Intl. Conf. on Enterprise Information Systems (ICEIS'06). Paphos, May 2006.
17. Marcos, E., Vela, B. and Cavero J.M. Methodological Approach for Object-Relational Database Design using UML. In R. France and B. Rumpe, (eds.), *Journal on Software and Systems Modeling (SoSyM),* Vol. 2, pp. 59-72, Springer-Verlag, 2003.
18. Marcos, E., De Castro, V. and Vela, B. Representing Web Services with UML: A Case Study, *Proc. of 1st Intl. Conf. on Service Oriented Computing*, Lecture Notes in Computer Science 2910, pp. 17-27, 2003.
19. Mikkonen, T., Pitkänen, R., and Pussinen, M. On the Role of Architectural Style in Model Driven Development. In Oquendo, F., Warboys, B. and Morrison, R. (eds.), *Proc. 1st European Workshop on Software Architecture (EWSA 2004)*, pp. 74-87, Lecture Notes in Computer Science, 3047. Springer Verlag, May 2004.
20. Miller, J., and Mujerki, J., editors. *MDA Guide, Version 1.0.* OMG Technical Report, Document omg/200-05-01. http://www.omg.com/mda, 2003.
21. OWL Services Coalition. *OWL-S: Semantic markup for Web services*. Retrieved from http://www.daml.org/services/owls/1.0/owl-s.html, 2003.
22. Rozanski, N. and Woods, E. *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*. Addison-Wesley, 2005.
23. Selic, B. *The pragmatics of Model-Driven developmen*t, IEEE Software, Vol. 20(5), Sep.-Oct. 2003, pp. 19-25, 2003.
24. Vara, J.M., De Castro, V., and Marcos, E. WSDL Automatic Generation from UML Models in a MDA Framework. In *International Journal of Web Services Practices* (IJWSP), Vol. 1 (1-2), pp. 1-12. CS Press, November 2005.
25. Vela, B., Acuña, C. and Marcos, E. A Model Driven Approach for XML Database Development. In *Proc. of 23rd Intl. Conf. on Conceptual Modelling (ER2004)*. Springer Verlag, Lecture Notes in Computer Science, 3288, pp. 780-794, 2004.

Layered Patterns in Modelling and Transformation of Service-Based Software Architectures

Claus Pahl and Ronan Barrett

Dublin City University
School of Computing
Dublin 9, Ireland
{cpahl|rbarrett}@computing.dcu.ie

Abstract. Service-oriented architecture is a recent paradigm for architectural design. The software engineering aspects in this context, that have not been sufficiently addressed, are software evolution and software migration. Architectures are of great importance if large software systems change. Architectural transformations can guide and make this change controllable. In this paper, we present a modelling and transformation method for service-based software systems. Architectural configurations, expressed through architectural patterns, form the core of an underlying specification and transformation calculus. Patterns on different levels of abstraction form transformation invariants that structure and constrain the transformation process. We explore the role layered patterns can play in modelling and as invariants for transformation techniques.

Keywords: Service-oriented Architecture, Service Processes, Architecture Specification, Architecture Transformation, Web Services.

1 Introduction

The development of distributed software systems based on service architectures is rapidly gaining momentum. *Service-oriented architecture* (SOA) is emerging as a new paradigm for the architectural design of widely distributed software systems, supported by platforms such as the *Web Services Framework* (WSF) [1]. Due to the ubiquity of the Web, the WSF platform and SOA paradigm can be expected to play a major role in the future of software development.

Architectural design is about separating computation from communication. In service-based, distributed environments such as the WSF, a notion of processes is central to capture service composition and interaction between services. We present an architectural model and engineering techniques to support, firstly, modelling and specification of services and service-based processes and, secondly, property-preserving transformations of service-based architectures.

Our solution is an approach to the *architectural configuration* of services, based on formal modelling of service communication and interaction processes. One of the distinguishing features of our approach is a *three-layered architecture model*

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 144–158, 2006.
© Springer-Verlag Berlin Heidelberg 2006

addressing different architectural levels of abstraction. Each layer is supported through a *pattern-based modelling approach.* A service-based architectural configuration calculus that combines patterns and process behaviour in architectures forms the backbone of this approach. The exploration of the role of layered process-oriented patterns is the central objective here.

Formality is required in this framework to obtain precise and unambiguous specifications of process-based service architectures and to complement specification by analysis and reasoning facilities. In particular architectural change and evolution requires a technique for process-oriented property-preserving transformations. Various formal approaches to the representation of processes have been suggested in the past. *Process calculi* such as the π-calculus [2] are suitable frameworks for architectural configurations due to their abstraction from service.

A number of different modelling approaches exist, using different formalisms, e.g. [5,4] using Petri nets. We use the π-calculus as the basis, which helps us to define a notation for service-based architectural configuration. The π-calculus, a calculus for mobile processes, is particularly useful due to a similarity between mobility and evolution – both are about changes of a service in relation to its neighbourhood – which helps us to support architectural transformations.

We give some background and an introduction to our layered architecture model, called SAM, and our transformation calculus, called SACC, in Section 2. Pattern-based architecture modelling and specification, supported by the architecture configuration calculus SACC, is addressed in Section 3. Architectural transformations are defined in Section 4. Finally, we discuss related work in Section 5 and end with some conclusions in Section 6. A Web-based, service-oriented learning technology system serves as a case study throughout the paper.

2 Architecture Model and Specification Calculus

The objective of *software architecture* [3] is the separation of communication from computation. Architectures are about *components* (i.e. loci of computation) and *connectors* (i.e. loci of communication). This allows a developer to focus on structures and the dynamics between components separately from component implementation. Various *architecture description languages* (ADL) and modelling and development techniques have been proposed [6,7,8]. An architectural model captures common concepts found in a variety of architectural description languages: components provide computation, interfaces provide access to blackbox components, and connectors provide connections between components. In service-based architectures, the focus shifts towards the composition of services to processes and the overall *configuration of services and service processes*. Process and interaction behaviour is an essential part of modelling service architectures [3].

A *service* is usually defined as a coherent set of operations provided at a certain location [1]. A service provider makes an abstract service interface description available, which can be used by potential service users to locate and invoke this service. Services are often used 'as is' in single request-response interactions. More recently, research has focussed on the *composition of services* to

processes [1]. Existing services can be reused to form business or workflow processes. The *principle of architectural composition* that we look at here is *process assembly.* The discovery and invocation infrastructure – a registry or marketplace, where potential users can search for suitable services, and an invocation protocol – with the services and their clients form a service-oriented platform.

At the core of our architecture modelling and transformation technique is a conceptual architecture model. The objective of this architecture model is to capture the characteristics of service-based architectures. A layered conceptual *service architecture model* (SAM), that is tailored towards the needs of service- and process-oriented platforms, shall address the different levels of abstraction in service-based architectures:

- *Reference architectures* are high-level specifications representing common structures of architectures specific to a particular domain or platform.
- *Architectural design patterns* are medium-scale patterns – usually referred to as design patterns or architectural frameworks.
- *Workflow patterns* are process-oriented patterns that represent common business or workflow processes in an application domain.

Based on the architecture model SAM, we define a calculus for architectural specification and transformation – the *service-based architectural configuration calculus* (SACC) – that has features of an abstract architectural description language (ADL) at its core[1]. Its main aim is to support the architectural configuration of services. Two elements define our calculus:

- a *description notation* to capture architectural properties,
- *rules and techniques* for transformation.

The calculus is directly based on the π-calculus [2]. However, it adds a few combinators to express workflow and design patterns. A simulation notion from the π-calculus helps us to capture the idea of property-preservation and permitted structure and behaviour variations during transformation.

Our architectural process specification notation consists of basic process activities, activity combinators, and process abstractions. The basic element describing process activity is an *action.* Actions π are combined to *service process expressions.* Actions of a service can be divided into

- *invocations* **inv** x(y) of other services via channel x, which connects to the remote service, passing y as a parameter,
- *activations* receive $\mathbf{rcv}_x$(a) from other services and the dual reply $\mathbf{rep}_x$(b), with channel x and parameters a and b.

The *process combinators* are basic forms of *workflow patterns*:

- *Actions* π are primitive processes.
- *Sequences* are represented as $P_1;P_2$, meaning that process P_1 is executed and the system transfers to P_2, where the next action is executed.

[1] This calculus does not qualify as an ADL since our focus is on processes and architectural configuration, neglecting interfaces and connector specifications.

- *Exclusive Choice* means that one P_i $(i = 1, 2)$ from **choice** P_1,P_2 is chosen.
- *Multi-Choice* **mchoice** P_1,P_2 allows any number of the processes P_i $(i = 1, 2)$ to be chosen and executed in parallel.
- *Iteration* **repeat** P executes process P an arbitrary number of times.
- *Parallel composition* **par** (P_1,P_2) executes processes P_1 and P_2 concurrently.

Additionally, *restriction* **restr** m.P means that m is only visible in P. $A(a_1, \ldots, a_n) = P_A$ is a *process abstraction*, where P is a process expression and the a_i are free variables in P. A local variable is introduced using **let** $x = \pi$ **in** P. Inaction is denoted by 0.

The *semantics* can be defined in terms of the π-calculus [2]. The language constructs can be directly mapped to π-calculus constructs. The basic actions are defined in terms of send $\overline{x}\langle y \rangle$ (for invocation **inv** and reply **rep**) and receive $x(y)$ (for receive **rcv**) of the π-calculus. The combinators are defined directly through their π-calculus counterparts, except the multichoice **mchoice** P_1,P_2, which is defined in terms of π-calculus-supported combinators as **choice** (A, B, **par** (A,B)) – essentially a parallel composition of all elements of the powerset of the **mchoice** argument list. The abstraction is the π-calculus abstraction.

3 Pattern-Based Service Architecture Modelling

The *service-based architectural configuration calculus* SACC enables modelling and specification of pattern-based service architecture configurations. We will use an e-learning system called IDLE – the Interactive Database Learning Environment – to illustrate our approach [9]. IDLE is based on a Web software architecture that provides a range of educational services:

- It is a multimedia system that uses different mechanisms to provide access to learning content, e.g. Web server and a (synchronised) audio server.
- It is a composite, interactive system that integrates components of a database development environment (a design editor, a programming interface, and an analysis tool) into a teaching and learning context.
- It is a constructive environment in which learners can develop their database applications, supported by shared storage and workspace.

In this section, we introduce a pattern-based modelling method that is suitable for modelling architectural configuration and processes of service architectures at different levels of abstraction, using IDLE for illustration. Our hypothesis is that the presented service process calculus SACC provides a suitable specification technique for modelling service architectures for all pattern types.

3.1 Patterns and Abstraction Levels

Architecture and *design patterns* are recurring solutions to software design problems [10]. These patterns are about the design and interaction of objects, as well as providing a communication platform concerning reusable solutions

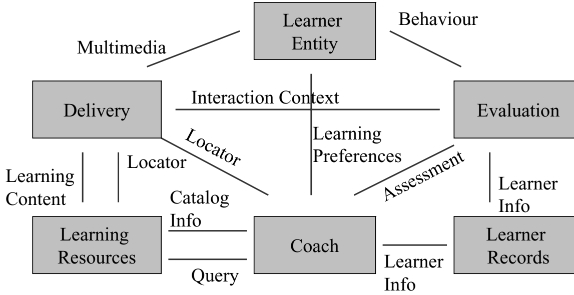


Fig. 1. Reference Architecture – Overview of the Reference Architecture LTSA

to commonly encountered design problems. Patterns at different levels of abstraction – reference architectures, architectural design patterns, and workflow patterns – form an essential part of our service-specific architectural transformation approach. We cover the three layers of the architecture model SAM with our notation. Workflow operators for service processes are directly integrated as operators. An architectural design pattern expressing service interaction patterns can be formulated as an expression of a number of concurrently executing processes. Reference architectures can be modelled at the level of abstractions.

Reference Architectures. *Reference architectures*, if they exist for a platform or a domain, can play an essential role in the architectural definition of a software system. They often emerge in an abstracted and standardised form from successful architectural assemblies. Reference architectures define accepted structures that help us to built maintainable and interoperable systems.

In the context of educational software systems, our case study domain, the IEEE-defined *Learning Technology Standard Architecture* (LTSA) provides a service-oriented reference architecture [11], see the UML-style class diagram in Fig. 1. Six central components such as Delivery or Coach are identified. These components provide services to other components, e.g. the Delivery component provides a Multimedia delivery service to the LearnerEntity. These services are usually related to processing data in different types of media.

Besides domain-specific architectures, platform-specific reference architecture are important. Examples of classical Web-based architectures are *client-server* architectures or *three-tiered architectures*.

Design Patterns. *Design patterns* are recognised as important building blocks in the development of software systems [10]. Their purpose is the identification of common structural and behavioural patterns. A rich set of design patterns has been described, which can be used to structure a software design at an intermediate level of abstraction. Usually, *architectural patterns* (such as model-view-controller) are distinguished from *design patterns* (such as factory, composite, or iterator). We see both forms of patterns as intermediate-level constraints on a system architecture, i.e. on services and on their interaction patterns.

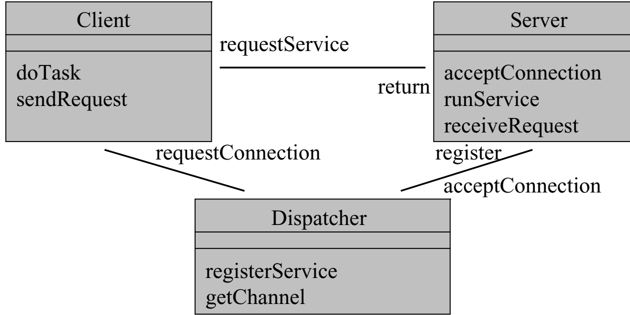


Fig. 2. Pattern – The Client-Dispatcher-Server Architectural Design Pattern

Learner = **repeat** (**inv** requestEducServ = requestConnection();
inv res = requestEducServ(resId))
Delivery = **inv** registerEducServ(id);
repeat (**rcv**$_{\text{acceptConnection}}$(c); **rcv**$_{\text{requestEducServ}}$(s);
rep$_{\text{requestEducServ}}$(run(s)))
Coach = **choice** (
choice (**rcv**$_{\text{registerEducServ}}$(id); **rcv**$_{\text{unregisterEducServ}}$(id))
repeat (**rcv**$_{\text{requestConnection}}$();
let c = getChannel()
in par (**inv** acceptConnection(c); **rep**$_{\text{requestConnection}}$(c))))

Fig. 3. Specification – Educational Service (EducServ) Registration and Provision in IDLE based on the Client-Dispatcher-Server Design Pattern

Design patterns also play a role in the design of Web services architectures [12]. An example of an architectural design pattern is the *client-dispatcher-server pattern* [12]. The pattern architecture with its interactions is visualised in Fig. 2 in UML-style representation. The client requests a service in the pattern. The server is the provider of the service. The dispatcher is the mediator between client and server. Servers register their services with the dispatcher and clients request connection channels to servers in order to use the services.

Example **1.** *In IDLE, a learner requests content from a resources server. The IDLE specification in SACC, Fig. 3, is based on the client-dispatcher-server pattern, Fig. 2, with the learner (as client), a coach (as dispatcher), and the resources and delivery subsystem (as server). The learner is a client invoking services of the delivery (request a connection and an educational service). The coach handles the service registration (from the delivery) and forwards the delivery channel (provides by the delivery component) to the learner. Passing channel*

$$\begin{aligned}\text{WorkSpace} = \textbf{choice}\ (\ & \\ & \textbf{repeat}\ (\ \textbf{rcv}_{\text{retrieve}}(\text{resId});\ \textbf{inv}\ \text{provide}(\text{res})\ )\ , \\ & \textbf{repeat}\ (\ \textbf{rcv}_{\text{store}}(\text{resId, res})\ )\ \ )\end{aligned}$$

Fig. 4. Specification – Specification of the IDLE Storage and Workspace Service

names over channels is a typical example of the π-calculus ability to model dynamic infrastructures. The learner then uses the provided channel to use the delivery component's educational service.

Abstracted pattern definitions such as *client-dispatcher-server* can act as building blocks in higher-level architectural specifications. Patterns are defined as process expressions and made available as process abstractions. These macro-style building blocks also form a pattern repository. A detailed discussion of pattern-based specification of IDLE can be found in [13].

Workflow patterns. *Workflow patterns* are small-scale process patterns [14]. Workflow patterns relate to connector types that are used in the composition of services – we provide them as built-in operators. An example of a workflow pattern is the Unix-style pipe, which is similar to a sequencing workflow pattern. Workflow patterns are small compositions of basic activities. Workflow patterns and their implementation in Web services architectures are described in [15].

Example 2. The multichoice operator is an example for process compositions [15]:

$$\textbf{mchoice}(\text{Lecture}, \text{Tutorial}, \text{Lab})$$

expresses that any selection of the IDLE services Lecture*,* Tutorial*, and* Lab *can be used concurrently, e.g. a user can use lecture and lab services in parallel.*

To identify these workflow patterns in the architecture specification is important since often not all pattern are supported by the implementation languages. Then, workarounds based on architectural transformations have to be found.

$$\textbf{choice}(A, B, C, \textbf{par}(A, B), \textbf{par}(A, C), \textbf{par}(B, C), \textbf{par}(A, B, C))$$

is an equivalent workaround to the multichoice workflow, needed if the implementation language does not support the multichoice pattern **mchoice**(A, B, C) – which is the case with some WS-BPEL implementations [15].

3.2 Modelling Service Architectures

Modelling service-oriented architectures starts with the identification of services. Two cases can be distinguished:

- Some of the components of a system will clearly exhibit service character – an SQL execution element, which is part of the IDLE lab resources and delivery subsystem, is an example.

– Some components might not be implemented as services, but could easily be wrapped up if required. An example of this category is a storage and workspace feature.

In our case study, the problem is re-engineering of a legacy system into a service-based system. The existing architecture – even though not adequately designed and documented – provides a starting point for service identification.

Example **3.** *We use the LTSA reference architecture as the starting point for the service-based modelling of IDLE due to the LTSA's SOA character. We, however, realise the storage and workspace function, which could have been integrated into either learning resources or learner records in terms of the LTSA, as a separate service. This IDLE feature can be specified as a service process, see Fig. 4. The workspace service either deals with incoming retrieval or storage requests.*

Once all services have been identified, the connections and interactions between services have to be modelled. We propose a top-down method starting with reference architectures, followed by architecture and design patterns and finally workflow patterns. Subsystems and composite components of high-level architectures are refined down to the workflow level. For instance, top-level LTSA services can be internally composed of small-scale interacting services. The presented modelling technique allows us to adequately address the modelling aspects of a service-based educational software system. We have presented this technique within a method for top-down, pattern-based layered modelling.

4 Transformation

Software architecture addresses more than the high-level system design. Software change resulting from maintenance and evolution is equally important. We focus on architecture transformations – a central software change technique. Often, architectural transformations are a necessity. Interoperability can be a transformation objective. For instance, a new reference architecture might need to be adopted. Another objective can be to accommodate changes in the interface and interaction processes of individual services. Workflow pattern are often transformed if implementation restrictions have to be dealt with.

A central objective of architecture transformation is to implement the planned changes, but also to preserve existing properties. Here, the existing service processes shall be preserved, i.e. process expressions act as invariants of the transformation. These processes are expressed as patterns at different levels of abstraction. While the idea of preserving patterns at all layers is obvious, a verifiable transformation technique is needed. A notion of simulation shall capture the ideas of equivalence and refinement of services and service processes – an essential element of the modelling aspect.

A prerequisite for the transformation is the explicit architecture specification of the existing system. A complete specification is not necessary; accuracy and level of preservation of the transformation, however, depend on the degree of

detail and number of patterns identified. In IDLE, we have for instance analysed an inadequately documented system to extract structures and patterns.

4.1 Simulation and Transformation Rules

Our transformation technique is based on a notion of simulation and on simulation-based transformation rules. It has to address the needs of layered pattern-based models and the focus on patterns as transformation invariants.

- Reference architectures. Each service abstraction is mapped to a service abstraction in the new architecture. The transformation objective determines whether the service process definition will have to be changed. The transformation is subject to invariants, i.e. pattern preservation.
- Architectural design patterns. Often, interaction processes needs to be changed to accommodate new or modified service functionality. Ideally, newly emerging patterns the service participates in will simulate the original patterns.
- Workflow patterns. Workflow pattern transformations can often be handled automatically in architecture implementations.

Property preservation is a central goal of our architecture transformations. A simulation notion shall capture service process pattern preservation in the transformation technique. A *simulation* definition, adopted from the π-calculus, satisfies the pattern preservation requirement for the processes that we envisage:

> A process Q *simulates* a process P if there exists a binary relation $\mathcal{S}$ over the set of processes such that if whenever $P\mathcal{S}Q$ and $P \xrightarrow{m} P'$ then there exists Q' such that $Q \xrightarrow{n} Q'$ and $P'\mathcal{S}Q'$ for service processes n and m.

This simulation definition expresses when a process Q based on service expressions n preserves, or simulates, the behaviour of a process P based on service expressions m. The services n and m can be unrelated, as this definition is about observable behaviour.

In order to automate transformation support based on this simulation definition, a constructive theorem supporting this definition is required. This will be the basis of a transformation rule which allows the verification of preservation and the automation of the transformation. In [16], we have developed a constructive simulation test based on the construction of transition graphs for the process expressions of the SACC calculus.

Since usually not the entire specified behaviour should be preserved, we have introduced the notion of patterns to capture common behavioural aspects that need to be preserved. Patterns at different levels of abstraction identify reliable and maintainable interaction patterns between services. These are ideally preserved. Central in our transformation technique is, therefore, the following *transformation rule*, which associates patterns and simulation:

> Given an architecture specification S in SACC, create an architecture specification S' as follows. For each abstraction A in S (apply this rule

recursively from top to bottom), *map* A *to* A' where A' is another abstraction such that for any pattern P, which A participates in, A' *simulates* P' with $P' = P[A/A']$, i.e. A' substitutes A in P. P is replaced by P' to cater for renaming of abstractions.

The determination of an invariant, the pattern P, is a common, but often non-trivial problem. This problem can be alleviated through domain-specific patterns. We will address this methodological aspect below.

4.2 Applying Pattern-Preserving Transformations

We will demonstrate the adoption of a new reference architecture, the LTSA, on the highest level of abstraction for the IDLE system. The transformation aim is interoperability of IDLE services and components with other LTSA-specified components and reuse. This interoperability objective, however, can have an impact on all levels of abstraction. For instance, the SCORM Run Time Environment standard prescribes interfaces for learning technology objects, which would have to be reflected in service interfaces here.

Example 4. The starting point for the transformation is the architecture specification of an existing system – in our case IDLE in its current form. IDLE on the highest level of abstraction is a parallel composition of composite processes:

$$\begin{aligned} \text{IDLE} = \textbf{par}\ (\ & \text{Learner}, \text{Delivery}, \text{StudentModel} \\ & \text{PedagogyModel}, \text{Workspace}, \text{Evaluation}, \ldots\ ) \end{aligned}$$

where each top-level service is an abstraction of a process expression based on other, more basic services. Some of these are already similar to LTSA components – we have indicated this fact by using the similar names – others such as StudentModel *and* PedagogyModel *have no direct counterpart in the LTSA. Several different combinations of individual services can form patterns; these might actually overlap. We will discuss an example later on.*

The first transformation step is to describe IDLE's architectural characteristics – ideally in terms of LTSA to simplify the transformation, see Fig. 1.

Example 5. The client-server-dispatcher pattern, see Fig. 2, is not identical to the structure that can be found in the IDLE system, see Fig. 3. We have added the interaction with the resources server. The pattern itself as an identifiable pattern is nonetheless worth preserving and is, thus, one of the invariants. In our case, the client-dispatcher-server *pattern:*

$$\textbf{par}\ (\text{Client, Dispatcher, Server})$$

is simulated *by the composite IDLE process:*

$$\textbf{par}\ (\text{LearnerEntity, Coach, Delivery})$$

resulting from the composition of learner, coach, and resources and delivery subsystems of the IDLE reformulation in LTSA terminology. This means that the pattern is a good abstraction of IDLE functionality that needs to be preserved.

LTSA is a high-level pattern. In IDLE, we add functionality. This architectural change arises from the workspace service integration into IDLE.

Example ***6.*** *The explicit storage and workspace service, see Fig. 4, requires the services* LearnerEntity *and* Delivery *to be modified in their interaction patterns. Again, the pattern shall be the invariant of the transformation, but some refinements – constrained by the simulation definition – need to be made to accommodate the added service within the system.*

In order to identify workflow patterns that need to be preserved, these can easily be identified due to their implementation as operators in the notation.

Example ***7.*** *The specification of the IDLE educational service system based on the client-dispatcher-server architectural design patterns in Fig. 3 based on Fig. 2 is defined in terms of workflow patterns. The* Learner *is based on a sequence of activities. The* Coach *is based on choice in the first part, and a concurrent split and merge in the second part. These are candidates for invariants.*

The transformation task is to transform IDLE into LTSA-IDLE – an architectural variant of IDLE with LTSA-conform interfaces and interaction processes.

Example ***8.*** *In the transformation, we need to consider the source, the invariant, the target construction, and the preservation proof:*

– Source*: The starting point of the transformation is the original IDLE specification. Since in our case a full specification did not exist, we analysed the system and extracted central features. The high-level architecture is given in Example 4 and some detailed excerpts are presented in Figs. 3 and 4.*
– Invariant*: The invarant is determined by patterns on different levels of abstraction. The LTSA determines the high-level architecture. We focus here on the client-dispatcher-server pattern as the architectural pattern invariant as explained in Example 5.*
– Target Construction*: The LTSA-based architecture specification of some IDLE services – which is the transformation result – can be found in Fig. 5. It is constructed based on our transformation rule as follows. At the reference architecture level, IDLE is mapped to LTSA-IDLE where the merger of* StudentModel *and* PedagogyModel *simulates the* Coach*. At the architectural design pattern level, the parallel composition of the individual components is changed at the subcomponent level (*Coach*) to reflect the merger.*
– Simulation and Preservation*: The invariants – LTSA and client-dispatcher-server – are two patterns that have to be simulated by the new architecture:*
  - *We have adapted our original terminology to the LTSA terminology. For instance,* Learner *becomes* LearnerEntity*. The two components* StudentModel *and* PedagogyModel *are merged into* Coach*, i.e. the model components were abstracted by a single* Coach *interface, which results in the LTSA pattern being simulated.*

LearnerEntity	= **inv** preferencesInfo = getPreferences(); **inv** setPreferences(alter(preferencesInfo)); **inv** learnResource = multimedia()
Coach'	= **repeat** (**choice** ($\mathbf{rcv}_{\text{getPreferences}}$(); $\mathbf{rep}_{\text{getPreferences}}$(prefInfo), $\mathbf{rcv}_{\text{setPreferences}}$(preferencesInfo), $\mathbf{rcv}_{\text{getLearnerInfo}}$(id); $\mathbf{rep}_{\text{getLearnerInfo}}$(info), **inv** uri = locator(resource)))
Delivery'	= $\mathbf{rcv}_{\text{locator}}$(uri); **inv** learnResource = retrieveResource(uri); $\mathbf{rep}_{\text{multimedia}}$(learnResource)
LearningResources	= $\mathbf{rcv}_{\text{retrieveResource}}$(uri); $\mathbf{rep}_{\text{retrieveResource}}$(retrieve(uri))
LearnerRecords	= $\mathbf{rcv}_{\text{getLearnerInfo}}$(id); $\mathbf{rep}_{\text{getLearnerInfo}}$(info(id))

Fig. 5. Transformation – Resulting Adaptive Delivery in IDLE Architecture (selected components and services) based on the LTSA

- *The new* Coach' *service handles the interaction with the learner and pedagogy model components internally. The original* Coach *specification from Fig. 3 has been adapted to reflect this fact. The structural and behavioural properties of the client-dispatcher-server pattern* P := **par**(Client, Dispatcher, Server) *are still intact, i.e. the pattern is preserved accroding to the transformation with pattern* P *and the original* Coach *adapted to* Coach'. *The three pattern components are still present and the externally visible interaction behaviour is the same*[2].

The specification in Fig. 5 describes the adaptive delivery of resources. After updating preferences by interacting with the coach, the learner entity requests and receives learning resources via a multimedia channel from the delivery service. The learning resources service retrieves the actual content for the delivery service, which in turn delivers it to the learner entity.

In our method, *design patterns* that can be identified in an existing system, such as the original IDLE, should be *invariants of the architectural transformation.* In [13], we have shown that design patterns that were identified for object-based systems also occur in service-oriented architectures. This method can be supported by transformation tools. The architect provides the source system model and identifies preservable patterns from the model patterns and, if necessary, renamings and non-standard transformations. The tool would then carry out the transformation by applying the transformation rule subsitutions to patterns and discharging the preservation proof obligations.

The combination of the most frequent patterns seem to be domain-specific, as our investigation indicates [13]. Examples of frequently occurring design patterns that we have identified in IDLE, other learning technology systems, and also the

[2] The formal proof is based on a constructive simulation test developed in [16], which is beyond the scope of this paper.

LTSA are the factory, proxy, observer, composite, and serialiser patterns. Other, less frequent patterns that we have found include the iterator and the strategy pattern. These common patterns could result in a domain-specific formulation of patterns such as LearnerEntity-Coach-Delivery (see Example 5) and a repository of domain-specific patterns, which would help software architects in the difficult task of identifying invariants of the transformation.

5 Related Work

Some ADLs are similar to our approach in terms of formality and their focus on processes. Darwin [17] is a π-calculus based ADL. Darwin focuses on component-oriented development approach, addressing behaviour and interfaces. Restrictions based on the declarative nature of Darwin make it rather unsuitable for the design of service-based architectures, where both binding and unbinding on demand are required features. Wright [18] is an ADL based on CSP as the process calculus. Wright supports compatibility and deadlock checks through formalised specifications, based on explicit connector types. This is an aspect that we have neglected here, but that could enable further analysis techniques, if we introduced typed channels. In [19], the formal foundations of a notion of behaviour conformance are explored, based on the π-calculus bisimilarity relation. We chose the π-calculus as our basis, since it caters for mobility, and, consequently, allows us to address architecture evolution and transformation [8]. Mobility allows us to deal with changes in the interaction infrastructure. The client-dispatcher-server pattern is an example where a new channel is dynamically formed. On the metalevel, architecture transformation also means controlled changing of architectural structures. The impact of observational semantics based on states denoting a family of bisimilar configurations has yet to be investigated in detail.

Patterns have recently been discussed in the context of Web service architectures [12,15]. In [15], a collection of workflow patterns is compiled. We have based our operator calculus on this collection, aiming at a support for most of the patterns described. The client-dispatcher-server pattern that we have identified in our IDLE system is also discussed in [12]. Other patterns that we have mentioned mainly originate from [10].

A recent software architecture approach for service-based systems is Model-Driven Architecture (MDA) [20]. MDA emphasises the importance of modelling and transformations. The latter are, in contrast to our framework, part of the modelling process between modelling layers. Our framework addresses the transformation of multi-layered architecture specifications. While MDA is vertically oriented, i.e. mapping from abstract domain models to more concrete platform and implementation models, we follow a more horizontal transformation approach on the level of architecture models.

6 Conclusions

A new architectural design paradigm such as service-oriented architecture (SOA) requires adequate methodological support for design, maintenance, and evolution.

While an underlying deployment platform exists in the form of the Web Services Framework (WSF), an engineering methodology and techniques are still largely missing. We have presented a layered architecture model (SAM) that captures architectural structures at different levels of abstraction through patterns. A calculus (SACC) allows the process behaviour in architectures and architectural configurations to be captured. Interaction behaviour and composite processes within the architecture have turned out to be an essential aspect for the development and maintenance of service-based systems.

The importance of modelling for SOA has been recognised – and has resulted in the development of Model-Driven Architecture (MDA) as an approach to support the design of service-based software systems. We have focussed on layered pattern-based process modelling and architectural configuration – two aspects that can complement the MDA approach. The formality of our approach satisfies the automation requirements of MDA and even adds reasoning aspects. We are currently working on an architectural configuration tool for Web services that supports workflow and architectural patterns in the specification and that automatically translates these platform-independent specifications into Web service-specific notations such as WS-BPEL. Our emphasis here was on the applicability of the method by demonstrating the usefulness for a service-based learning technology system. We have investigated the role that layered pattern modelling can play for service-oriented architecture. The purpose of the SAM model and the SACC calculus is to provide a support technique for this modelling.

We have applied the presented framework in the ongoing design, maintenance, and evolution of the IDLE environment. The transformation technique was only outlined in its principles – our objective was the motivation of the method. In general, some of the architectural engineering activities can be better supported. The pattern framework could be extended to include distribution patterns, which would complement the existing layered functional patterns. A critical aspect of the approach is the reliance on the quality of the architectural description of the original system and the adequacy of the identified patterns. Transformations depend on the detail of the input architecture and the patterns that define the transformation invariant. The extraction of a system's architecture and the correct identification of intended patterns for undocumented systems is a difficult aspect that, although essential for the success has been addressed only through the idea of domain-specific patterns. Re-engineering approaches for the architectural level can provide further solutions here.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer-Verlag, 2004.
2. D. Sangiorgi and D. Walker. *The π-calculus – A Theory of Mobile Processes*. Cambridge University Press, 2001.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. SEI Series in Software Engineering. Addison-Wesley, 2003.

4. B.-H. Schlingloff and A. Martens and K. Schmidt. Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3-26. 2005.
5. R. Dijkman and M. Dumas. Service-oriented Design: A Multi-viewpoint Approach. *Intl. Journal of Cooperative Information Systems*, 13(4):337-368. 2004.
6. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In *Proceedings European Conference on Software Engineering / International Symposium on Foundations of Software Engineering ESEC/FSE'97*, pages 60–76. Springer-Verlag, 1997.
7. C. E. Cuesta, M. del Pilar Romay, P. de la Fuente, and Manuel Barrio-Solorzano. Architectural Aspects of Architectural Aspects. In R. Morrison, B.C. Warboys, and F. Oquendo, editors, *2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047, 2005.
8. F. Oquendo, B.C. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWARE: Architecting Evolvable Software. In R. Morrison, B.C. Warboys, and F. Oquendo, editors, *2nd European Workshop on Software Architecture EWSA 2005*. Springer LNCS 3047, 2005.
9. C. Pahl, R. Barrett, and C. Kenny. Supporting Active Database Learning and Training through Interactive Multimedia. In *Proc. Intl. Conf. on Innovation and Technology in Computer Science Education ITiCSE'04*. ACM, 2004.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Design*. Addison Wesley, 1995.
11. IEEE Learning Technology Standards Committee LTSC. *IEEE P1484.1/D8. Draft Standard for Learning Technology - Learning Technology Systems Architecture LTSA*. IEEE Computer Society, 2001.
12. N.Y. Topaloglu and R. Capilla. Modeling the Variability of Web Services from a Pattern Point of View. In L.J. Zhang and M. Jeckle, editors, *Proc. European Conf. on Web Services ECOWS'04*, pages 128–138. Springer-Verlag, LNCS 3250, 2004.
13. C. Pahl and R. Barrett. Towards a Re-engineering Method for Web Services Architectures. In *Proc. 3rd Nordic Conference on Web Services NCWS'04*. 2004.
14. W.M.P. van der Aalst, B. Kiepuszewski A.H.M. ter Hofstede, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.
15. M. Vasko and S. Duskar. An Analysis of Web Services Flow Patterns in Collaxa. In L.J. Zhang and M. Jeckle, editors, *Proc. European Conf. on Web Services ECOWS'04*, pages 1–14. Springer LNCS 3250, 2004.
16. C. Pahl. An Ontology for Software Component Matching. In M. Pezzè, editor, *Proc. Fundamental Approaches to Software Engineering FASE'2003*, pages 6–21. Springer-Verlag, LNCS 2621, 2003.
17. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, Springer LNCS 989, pages 137–153. 1995.
18. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
19. C. Canal, E. Pimentel, and J.M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
20. Object Management Group. *MDA Model-Driven Architecture Guide V1.0.1*. OMG, 2003.

Towards MDD Transformations from AO Requirements into AO Architecture*

Pablo Sánchez[1], José Magno[2], Lidia Fuentes[1], Ana Moreira[3], and João Araújo[3]

[1] Dpto. Lenguajes y Ciencias de la Computación
ETSI Informática, Universidad de Málaga
Málaga (Spain)
{pablo, lff}@lcc.uma.es
[2] Dpto. Engenharia Informática,
Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Leiria,
Leiria (Portugal)
magno@estg.ipleiria.pt
[3] CITI/Dpto. Informática Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa
Lisboa (Portugal)
{amm,ja}@di.fct.unl

Abstract. Aspect-Oriented (AO) Software Development has been created to offer improved *separation of concerns* mechanisms. AO concepts first appeared at the programming level and are now being addressed at the early stages of the software development life cycle. Currently, there are several AO approaches available for the various software development phases, but each one usually encompasses a single phase of the software process. This results in a wide gap between proposals at different levels of abstractions, raising several problems when trying to map artifacts between proposals from adjoining levels. This gap is clearly noticeable when an AO architecture design is intended to be derived from an AO requirements specification, since some requirements artifacts in AO approaches cannot be easily mapped to architectural artifacts. This paper explains how to reduce this gap by using model transformations between AO requirements engineering models and AO architecture design models. The goal is to automate part of the process of deriving an AO software architecture from an AO requirements specification.

1 Introduction

The *Software Architecture* of a computing system usually has to satisfy a set of requirements (functional and extra- or non-functional) specified during the requirements analysis phase. It is the responsibility of the software architect to design an architecture for the system that meets the functional requirements and

* This work has been supported in part by the project HP2004-0015 and EC Grant IST-2-004349-NOE AOSD-Europe.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 159–174, 2006.
© Springer-Verlag Berlin Heidelberg 2006

satisfies the quality concerns, or non-functional requirements, such as robustness, security, distribution, etc.

Such concerns normally affect various artifacts of the application architecture. However, in traditional architectural representations these concerns, known as *crosscutting concerns*, cannot be properly modularised. The tyranny of the dominant decomposition is acknowledged as the main cause that hinders an effective separation of concerns [1]. Crosscutting concerns normally appear *scattered* across different components of the application architecture, resulting, as a side effect, in *tangled* base architectural artifacts, which makes software architecture definition, adaptation, maintenance and evolution more difficult.

Aspect-Oriented Software Development (AOSD)[1] is an emerging discipline that promotes the separation of crosscutting concerns, by encapsulating them in special modules, the *aspects*.

AOSD initial research focused mainly on the implementation level. Recently, the main concepts of AOSD are slowly being considered at the earlier stages of the software development lifecycle. Several Early Aspects approaches have been proposed[2] [2] recently, including Aspect-Oriented Requirements Engineering (AORE) approaches as well as Aspect-Oriented Architecture Design (AOAD) approaches. These proposals normally focus only on requirements or on the architecture, but do not address how to map aspects identified at requirements level into architectural artefacts, in an integrated fashion. Therefore, the benefits of identifying aspects at requirements level may be lost at architectural level, or the step of specifying application architecture is simply skipped [3].

There are some initial approaches that try to map proposals from different authors [4,5,6], providing a set of guidelines to relate concerns gathered during the requirement analysis to architectural concerns. Even when a clear mapping process is available, a manual application of the process can be a repetitive, laborious and error-prone task. So our goal is to automate the process of deriving an AO architecture from an AO requirements specification. The resulting architecture descriptions will benefit from AORE processes, since these already identify aspects that will be reflected, and also managed, in the architecture. We will use Model-Driven Development(MDD) to support the automation of this process.

Since many of the aspects identified at the requirements level are typical of distributed systems (e.g. security, fault tolerance, response time) their transformation to the architecture level can be reused in many systems. The application of repetitive common solutions is encapsulated in a model transformation, which, in this case, takes a requirements model and produces an architectural model.

By combining AOSD and MDD technologies to derive software architecture descriptions, the following benefits are obtained:

– Repetitive, laborious and error-prone tasks, required to create a model from another model, are automated using MDD transformations.
– Transformations from requirements analysis models to architectural models are performed in a consistent manner, since they are automated.

[1] http://www.aosd.net
[2] http://www.early-aspects.net

– Best practices and recurring scenarios can be encapsulated in automatic transformations.
– AOSD improves modularisation, therefore improving software development, maintenance and evolution. New requirements can be incorporated into the AORE model and consistently and automatically propagated to the architecture level.
– AOSD promotes the reuse of architectural (and requirements analysis) artefacts. As base artefacts do not contain crosscutting concerns, they are highly reusable. Also, as crosscutting concerns are well-encapsulated in *aspectual* modules, these are also potentially reusable.
– As the obtained architecture is better modularised, opportunities for parallel development are also increased.

After this introduction the paper is structured as follows: Section 2 provides some background on AOSD. Section 3 describes our proposal, with its main tasks, model transformations and tool support. Section 4 discusses some related work. Finally, Section 5 outlines our conclusions and gives directions for future work.

2 Aspect-Oriented Software Development

When decomposing a system at a specific development level, some concerns may not fit the selected decomposition criteria. In Figure 1.a, the concerns participating in a specific system are shown in the upper level. A system decomposition for a particular stage (requirements, architecture, implementation, etc.) is shown below. In classic decomposition techniques (object-oriented, component-based, etc.) some concerns, named crosscutting concerns do not align well with the decomposition criteria and therefore cannot be adequately modularised, appearing tangled and scattered in several decomposition artifacts. For instance, the concern AccessControl is tangled with the concerns Encryption and Registration in the artefact B, which should ideally only encapsulate the Registration concern. AccessControl is also scattered along with artifacts A and B. Crosscutting concerns hinder software development, maintainability and evolution. AOSD improves

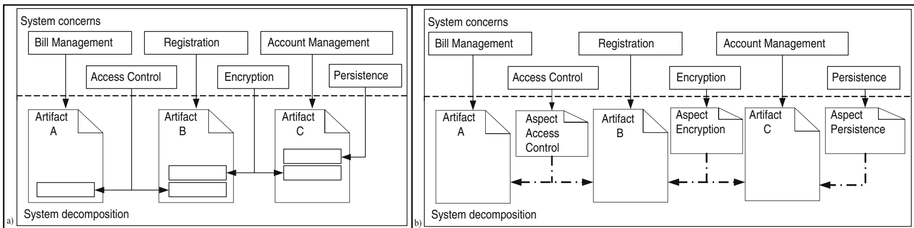


Fig. 1. Modularisation of crosscutting concerns

the system modularisation providing mechanisms to encapsulate crosscutting concerns, in special modules called aspects (see Figure 1.b).

Special composition rules (dotted-dashed lines) indicate how the aspects are woven into the base units (in this case artifacts A, B and C), avoiding the tangling and the scattering of the crosscutting concerns (AccessControl, Encryption and Persistence).

When applied to requirements engineering, base artifacts usually are use cases, viewpoints, goals, etc. Aspect-Oriented Requirements Engineering provides a systematic means for the identification, modularization, representation and composition of crosscutting properties, both functional and nonfunctional ones. These crosscutting concerns are encapsulated in separate modules, known as aspects, and special composition mechanisms are offered to support influence analysis and trade-offs before the architecture design is derived.

When applied to software architecture, base units are components and aspects are a special kind of component which encapsulate crosscutting behaviours. Aspectual components are composed (woven, in AO terminology) with base components in order to obtain the whole application. Following the component technology principles, those points in which aspect behaviour can be added (join points, in AO terminology) to a base component behaviour can only be those that appear as part of the component public interface (i.e. component creation/destruction, message sending/receiving, etc.).

3 From AO Requirements Engineering to AO Architecture Using MDD

3.1 Our Approach

Our aim is to derive an AO architecture from an AO requirement specification, preserving, whenever possible, the information contained in the requirements specification. This is achieved through the process depicted in Figure 2.

AORE. The first task is to collect system requirements through interviews with stakeholders, analysis of business rules, etc. Using an AORE approach, system concerns are identified, captured, composed and analysed. The output of this process is a requirements textual specification describing the relevant information about the system concerns (either functional, such as Billing or

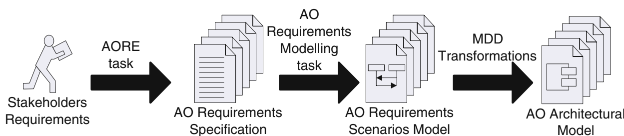


Fig. 2. Process for automatic transformation of an AO requirements model into an AO architectural model

non-functional such as Integrity). According to [7], functional requirements can be described at different levels of granularity: *sky* (broad system goals), *kite* (more detailed goals containing several subgoals), *sea* (a single system interaction), or *mud* (low-level and highly detailed requirement) levels. Our approach is independent of the AORE approach selected, but a constraint is imposed on the output of this process: functional requirements should be modelled at the sea- level to facilitate model transformations. Since there will be fewer elements in each scenario, they are affected by fewer non-functional requirements, and architectures can be constructed by small increments of simple scenario transformations.

AO Requirements Analysis Modelling. A UML model is constructed for the textual AO requirements specification obtained in the previous step. We have developed a UML Profile for modelling AO requirements. It is the starting point of our approach. Using this Profile, AO requirements are modelled as a set of scenarios, and each one may have several non-functional concerns as explained in following subsections. Therefore, the output of this process is an AO requirements scenario model.

MDD Transformations. The AO requirements scenario model is transformed into an AO architectural model using predefined MDD transformations. The AO architectural model is constructed incrementally by transforming each scenario individually. To transform each scenario, each functional requirement is transformed first, and then, the non-functional requirements are injected into the transformed functional requirement to ensure all the information present in the requirements model is used at architectural level. The architectural model is expressed using the UML 2.0 Profile for CAM [8], one of the few AO architectural approaches currently available, and model transformations are expressed using the QVT (Query, View, Transformations) standard.

Since there may be different strategies to design an architectural solution, it is possible to define several transformations for a single non-functional requirement. It is the responsibility of the software architect to select the appropriate set of transformations in order to generate an architecture that satisfies the required quality attributes. On the other hand, architects may first generate several candidate architectures, each one attending to different architectural design decisions, and then analyse them and select the best one. In general, some non-functional requirements might be lost during the transformation, not appearing explicitly in the resulting architectural model. To handle this, an auxiliary traceability file would be required.

Each step is detailed in the following subsections.

3.2 AORE

The fist step in our approach is to apply an Aspect-Oriented Requirements Engineering (AORE) approach in order to gather the system requirements, functional and non-functional (quality attributes), which are going to be used to

drive the architecture design. Our approach is independent of the AORE approach selected. The only restriction imposed, as commented before, is related to the sea-level granularity of the description of the functional requirements [7], to guarantee that each one contains a single interaction with the system, plus a set of associated non-functional requirements.

In order to explain some of the concepts introduced in this paper we use the Portuguese Automatic Toll Collection System case study described as:

In a road traffic pricing system, drivers of authorised vehicles are automatically charged at toll gates. The gates are placed in special lanes called green lanes. A driver has to install a device (a gizmo) in his/her vehicle. The registration of authorised vehicles includes the owners personal data, bank account number and vehicle details. The gizmo is sent to the client to be activated using an ATM that informs the system upon gizmo activation. The toll gate sensors read a gizmo. The information read is stored by the system, and used to debit the respective account. When an authorised vehicle passes through a green lane, a green light lights up, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the number (used to find the owner of the vehicle). There are three types of toll gates: single toll, where the same type of vehicles pays a fixed amount, entry toll to enter a motorway and exit toll to leave it. The amount paid on motorways depends on the type of the vehicle and the distance travelled.

For illustration purposes, we have adopted the Aspect-Oriented Requirements Analysis (AORA) [9,10,11] approach for AORE, since we have previous experience handling it. However, as we said, other approaches could also be selected. The application of this approach to our case study is briefly described below. This approach, as are most of AORE approaches [12]. is composed of three main tasks: Identify Concerns, Specify Concerns, and Compose Concerns. The identified kite-level [7] concerns are:

- Register Vehicle: It registers vehicle and owners data. It includes the return payment information from the bank and the gizmo activation.
- Pass Single Toll: This handles usage in a single point tollgate. If the vehicle is registered a light turns green, the amount to be debited in the owner's account is displayed and the passage is stored. Otherwise, the light turns yellow and the vehicle plate number is photographed.
- Enter Motorway: It handles vehicles joining a motorway. If the vehicle is registered, a light turns green and the entry data is stored. Otherwise, the light turns yellow and the vehicle registration number is photographed.
- Exit Motorway: This handles vehicles leaving the motorway. If the vehicle is registered and entered correctly, a light turns green and the amount to debit is displayed. Otherwise, the light is turns yellow and the vehicle registration number photographed. Data about the usage is stored in the system.
- Pay Monthly Bill: This bills the system users on a monthly basis. Payments are effected through bank transfers from the vehicle owner's account.

We now need to refine these concerns to the sea-level [7]. For example, the Authorise Vehicle scenario is identified as part of the Pass Single Toll, Enter Motorway and Exit Motorway broader scenarios. This scenario is responsible for identifying a vehicle (by its gizmo) each time it passes through a toll gate. Additionally, it has to satisfy a set of quality attributes (or non-functional requirements). In this paper, only a subset of all non-functional concerns associated with this scenario will be considered. In particular, the Authorise Vehicle scenario needs to guarantee that only *authenticated* drivers use the system and it needs to be executed within a low time frame (*response-time*). Therefore the Authorise Vehicle scenario will be defined by the functional *Identification* and the non-functional *Authenticity* and *Response Time* scenarios. The AORA process identifies *Authenticity* and *Response Time* as crosscutting concerns at the requirements level. We will use the Authorise Vehicle scenario to illustrate our approach.

3.3 AO Requirements Modelling with Scenarios

Once the system has been decomposed in simple scenarios, the next step is to model them in UML. We have developed a UML Profile, similar to [13], to address this goal.

Figure 3 shows how the Authorised Vehicle scenario is modelled according to that Profile. For each scenario identified, a package is created, which contains a UML sequence diagram describing the functional requirements of that scenario. In Figure 3 (left) the simple functional concern Identification is modelled. The non-functional concerns are also modelled as UML diagrams. To improve separation of concerns, non-functional concerns are described independently of functional concerns as parameterised UML diagrams, following an approach similar to Theme [3]. Therefore, non-functional concerns do not contain any reference to specific base concerns. For instance, the Response Time between two messages can be described abstractly using a UML sequence diagram plus a time constraint as shown in Figure 3 (central part). Authenticity could be described by means of another diagram or textually (see Figure 3, right). To guarantee Authenticity between messageA, sent by A to B, and messageB, sent by B to A, some

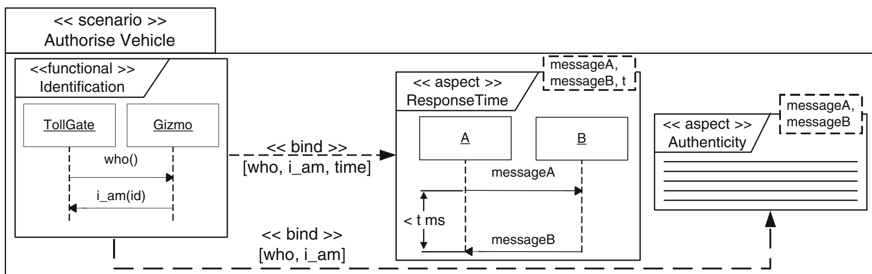


Fig. 3. UML representation of Authorised Vehicle scenario

extra mechanisms have to be provided. At the requirements level, this textual description would be enough. How non-functional requirements are modelled does not affect their transformations (as will be mentioned in the next section).

Although these diagrams are shown inside the packages representing the scenarios, they are really defined outside these packages, and later imported by them. This avoids scattered representations of non-functional requirements.

Functional and non-functional requirements are composed by means of bind relationships, also adopted from the Theme approach [3]. These *binds* relationships specify how and where non-functional concerns affects the functional concerns. For instance, the bind(who, i_am, time) provides specific elements, coming from functional concerns, to the *Response Time* template parameters. With these actual values, the template can be instantiated, specifying how *Response Time* is applied on the Identification scenario.

The model of Figure 3 illustrates how AOSD improves modularisation. Crosscutting concerns, like Response Time (between a request and its answer) and Authenticity (of the answer to a request), are encapsulated in diagrams, which are described independently and kept separate from functional concerns.

3.4 Transformations

These transformations are specified using the QVT (Query, View, Transformations) standard [14]. QVT is a standard model transformation language proposed by OMG[3] and comprises three sublanguages: QVT-Relational, QVT-Core and QVT-Operational, which offer different styles and abstraction levels for specifying transformation specifications.

We have opted for specifying transformations using QVT-Relational, since it is more user-friendly. A transformation expressed in this language can be viewed as a graph transformation, called *relation*, which contains two patterns, described based on instances of the metaclasses of the corresponding metamodel and their relationships. Thus, to specify transformations in QVT requires in depth knowledge of the source and target metamodels. Transformations are executed in one direction, which determines which are the source and the target models. The semantics of a model transformation is: whenever the source pattern is found in the source model, the target pattern has also to appear in the target metamodel. To satisfy the relation, it is allowed to create, update and delete objects of the target metamodel.

Transformations of requirements models in architectural models is achieved in three main steps:

1. Components, ports and the interfaces of the architectural models are created based on information of functional concerns. For each kind of lifeline in the sequence diagram representing the functional concern, a component is created with a port. If a lifeline A receives a message, an interface IA is created. A provided relationship is established at the architectural level between the

[3] Object Management Group, http://www.omg.org

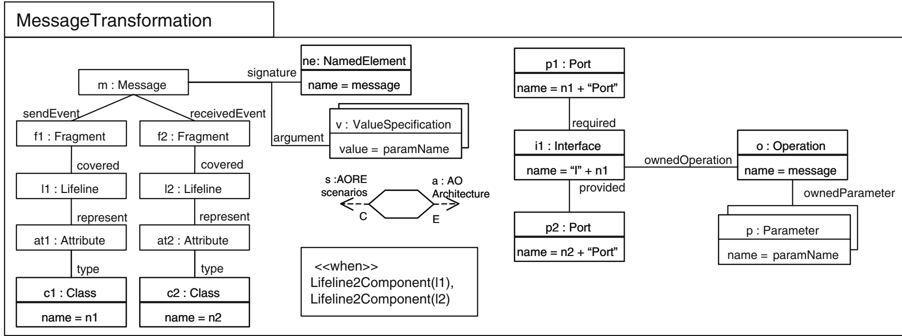


Fig. 4. Message transformation specified in QVT

component A (resulting from transforming the lifeline A) and the newly created interface IA. For each lifeline B that sends messages to the lifeline A, a required relationship is added to the architecture from the corresponding component B (created previously by transformation of lifeline B) and the newly created interface IA.

2. A partial sequence diagram for each functional scenario is created. This diagram contains the transformation of the interactions which do not appear as parameters in bind relationships, i.e. are not affected by aspects.
3. Interactions affected by aspects are transformed into interactions of the architectural model using *pattern-based transformations*. These kinds of transformations, a certain pattern encapsulates the design of a solution for the aspectual requirement. This pattern is instantiated using the parameters of the bind relationships that compose aspects and base functional concerns. This transformation completes the sequence diagram created in the previous step. For instance, the message who() sent from the TollGate lifeline to the Gizmo lifeline would not be transformed in the previous step because it is affected by an Authentication aspect. Then, in this third step, it is transformed into an augmented interaction with new elements (added by the pattern) which provides the required authentication support.

Figure 4 shows one of the transformations relate to functional concerns, specifically the transformation for creating an interface, provided/required relationships, and an operation in the interface for each message exchanged between lifelines at requirements level. The left pattern specifies, in terms of the UML metamodel, that each time a message between two lifelines is found in the source model (left pattern), an interface with a specific name should exist. This interface is provided by the port associated with the component resulting from transforming the lifeline which receives the message(right pattern). This interface is required by the port associated with the component resulting from transforming the lifeline which sends the message(right pattern). The when clause, in a QVT relation, indicates that before satisfying this relation, the Life2line2Component relation has to be satisfied by the lifelines l1 and l2. It ensures that components

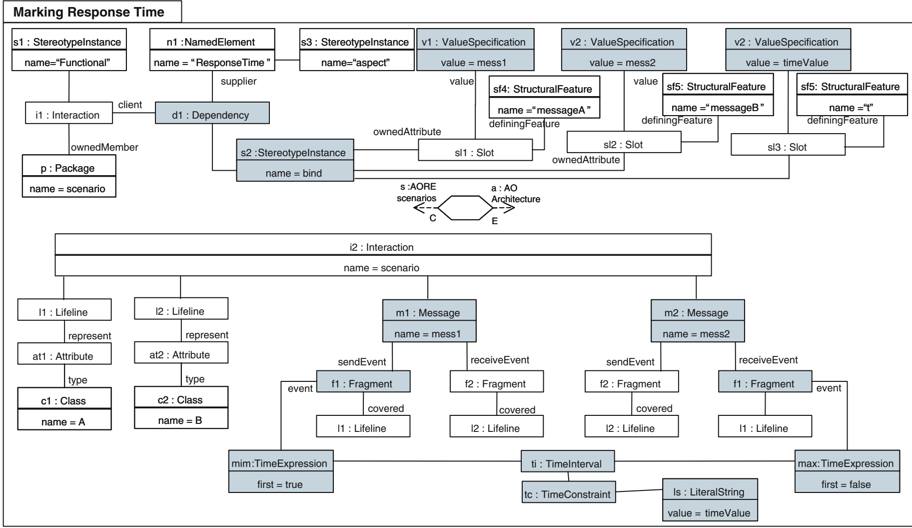


Fig. 5. Response Time Pattern Transformation in QVT

and associated ports have already been created when the MessageTransformation relation is trying to be satisfied.

Figure 5 depicts a pattern transformation for the aspectual requirement Response Time. The main elements for performing this transformation are shown in gray. Basically, each time, inside a scenario, a bind relationship is found associating a functional concern with a ResponseTime concern (top pattern), a UML temporal constraint for the messages influenced by the ResponseTime aspect is injected at the architectural level (bottom pattern). For the injection of the temporal constraint, actual values of the bind relationship parameters mess1, mess2 and timeValue are used.

3.5 Architectural Description Obtained

After applying the transformations to the Authorise Vehicle scenario the architectural solution of Figure 6 is obtained. This architecture is expressed by means of the UML 2.0 Profile for CAM [8][4].

In the UML 2.0 Profile for CAM, components are represented as common UML 2.0 components. Aspects are depicted as a special kind of component, stereotyped as ≪aspect≫. Provided/required interfaces are represented in the usual UML 2.0 notation. CAM components never interchange messages directly, instead, they communicate through their ports. Messages sent to a port from

[4] This reference contains only the CAM description and the UML 1.x Profile for CAM. A complete description of the UML 2.0 CAM Profile will be published soon by authors as a Technical Report.

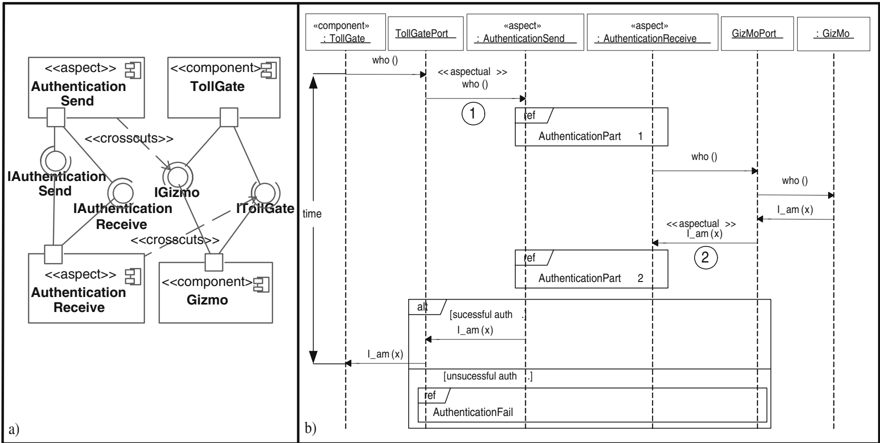


Fig. 6. Architectural representation of the Authorise Vehicle scenario

outside a component are forwarded to component internals, while messages sent to the port from the inside are forwarded to the connected external components. This approach enables the sender to declare required interfaces, and to send messages to its own ports when communicating with the environment, rather than identifying an external target component directly. Components defined in this way are assembled by wiring them together by means of provided/required interfaces. CAM models contain two different views, a structural view, detailing components, ports, interfaces and connections, and a behavioural view detailing how components interact exchanging messages. Aspects are executed on component interactions.

In the structural view (Figure 6.a), that an aspect crosscuts an interface can be indicated by means of a dependency stereotyped as ≪crosscuts≫ from the aspect to the crosscut interface. This relationship is optional and it only serves for the purposes of drawing attention to relevant crosscuttings in structural views. To indicate all the existing crosscuttings in structural views can lead to cluttered diagrams. How aspects are composed with base components is indicated by means of sequence diagramas (Figure 6.b), placing a message, stereotyped as ≪aspectual≫, from a port to an aspect, when an aspect has to be executed on a component interaction. In Figure 6.b, the who() message (label 1) sent from the TollGatePort to the aspect AuthenticationSend, and stereotyped as ≪aspectual≫, represents the interception by this aspect of the who() message, originally sent from the TollGate to the GizMo. The same idea applies to the message I_am(x) sent from the GizMo port and intercepted by the aspect AuthtenticationReceive. Although an aspect execution is modelled as an explicit method call, what it actually means is that between the time when a port receives a message and when this message is dispatched, the crosscutting behaviour (an advice in AspectJ [15] terminology) is executed obliviously to the component. It should be

noticed that aspects are applied outside the components and interactions between a component and its ports are aspect free. Therefore, components are not aware of aspects applied to them, which improves components reusability and composition.

When the transformations are applied to the scenario in Figure 3, the TollGate and Gizmo lifelines of the functional requirement Identification are transformed into components, and the messages who() and i_am(x) are transformed into operations of the interfaces ITollGate and IGizmo respectively. These interfaces are associated with the previously created components as shown in Figure 6.a. The aspectual components AuthenticationSend and AuthenticationReceive are also introduced in the architecture to satisfy the non-functional requirement Authenticity.

The behavioural view of this architecture is depicted in Figure 6.b. These diagrams show the interaction between components that fulfils the functional requirement Identification (messages who() and I_am(x) outgoing and incoming from/to components). In addition, this functional requirement is performed achieving Authenticity (who() and I_am(x) messages intercepted by the aspects AuthenticationSend and AuthenticationReceive; dialogues between these aspects). It should also be noted that all of the identification process, including the authentication task, has to be performed within a specific time constraint.

The main contributions of this architectural description are: (1) Interactions between component internals and ports are not affected by non-functional requirements, which are satisfied outside the components by means of aspects. It promotes parallel development and component reuse. (2) Trade-offs between non-functional requirements are preserved and reflected at architectural level. The AORE process has to able to detect a trade-off between Authenticity and ResponseTime. This trade-off is reflected in the architectural description since the time constraint includes the authentication dialogues. It means that when designers select specific algorithms to perform the authentication process, they should take the time constraint into account.

3.6 Tool Support

In the development of this approach, we have tried to use standard tools and languages, whenever possible.

As our approach is independent of the AORE process selected, the tool support for this stage depends on the toll support provided by the specific AORE process chosen. However, as any AORE process is allowed, the requirements engineer can select the process that s/he prefers thereby decreasing the adoption effort required for this approach.

Both the AO scenario model and the architectural model are based on UML 2.0 Profiles. The AO scenario model serves as input for transformations, which returns AO architectural models. Transformation tools manage UML models using their XMI representation (XMI is a standard to serialize models according to a machine-readable XML format). Therefore, to produce input models

and/or visualize output models, any UML 2.0 tool with an XMI import/export facility can be selected. Currently, there is a sufficient number of modelling tools supporting this[5], so development teams can choose their preferred one.

Although QVT is a standard supported by OMG, there is, as yet, a lack of stable tools supporting any of its languages. We have specified transformations in QVT for two reasons:

1. When stable tools are released, QVT-relations might be implemented directly on them;
2. In the meanwhile, QVT-relations serve as a great guide to specify how to implement transformations in other languages.

Since AO scenario models and AO architectural models can be serialized in XML, the problem of implementing a model transformation (QVT-relation) can be considered as a problem of creating an XML document (target model) from another XML document (source model). XSLT can be adopted for implementing QVT-relations, because it offers a standard, vendor-independent and high-level solution for transforming an XML document into another XML document. Other solutions for parsing and creating XML documents, such as SAXP[6] or DOM[7], could be used.

As model transformations become more and more complex, implementing them by dealing with XMI representations of the models becomes more difficult and tedious to maintain. If the transformation complexity is quite high, the use of non-standard model transformation languages, such as ATL[8] or MTF[9] should be considered. These languages hide the complexity of dealing with the XMI representation of the models, providing high level constructs to manipulate them. In this sense, ATL seems to be the most suitable choice, since it is more or less similar to the QVT standard. However, if the complexity of the transformations is manageable, the effort of learning a non-standard language which will probably disappear in the next few years, would not be justified.

4 Related Work

To the best of our knowledge, this is one of the initial approaches combining MDD and AOSD to produce aspect-oriented architectures from aspect-oriented requirements models. However, there are several works combining AOSD and MDD at different developmental phases. As clearly stated in [16,17,18], the main motivation for applying AOSD to MDD/MDA is to solve the problem that MDD presents regarding the lack of specific mechanisms for the separation of crosscutting concerns at each modelling level. If all concerns can be modelled

[5] http://www.uml.org/#Links-UML2Tools
[6] JSR 173 - http://www.jcp.org/en/jsr/detail?id=173
[7] http://www.w3.org/DOM/
[8] http://www.sciences.univ-nantes.fr/lina/atl/
[9] http://www.alphaworks.ibm.com/tech/mtf

separately at a certain level of abstraction, models of that level will become more manageable, extensible and maintainable. An alternative approach, close to ours, consists of applying the MDD philosophy to the development of aspect-oriented applications, by proposing MDD/MDA generation processes to transform AO models at different phases of the software life cycle. The AOMDF (Aspect-Oriented Model Driven Framework) proposal [19] combines both approaches, but it focus on the detailed design development phase.

Deriving an aspect-oriented requirements specification from aspect-oriented architectural models has been addressed in [4,20], where the authors establish some guidelines and heuristics about how this mapping should be performed. However, these mapping processes are manual, since they do not cope with new MDD techniques.

5 Conclusions and Future Work

This paper presents an initial step towards automating the generation of aspect-oriented architectures from aspect-oriented requirements specifications. The process discussed uses MDD and it is (semi)automatic, as the architects have to select a specific set of transformations from among several possible choices, each one corresponding to a candidate architecture which satisfies the functional and the non-functional requirements. When architects have doubts regarding what would be the best software architecture, they can generate several architecture candidates with less effort, because architecture generation is performed automatically. They can then apply an aspect-oriented software architecture analysis approach [21].

A case study has been used to illustrate our approach. After producing an AO requirement specification using the AORA [9,10,11] process, we identify a set of AO scenarios, each one containing a simple (sea-level) functional requirement and a set of non-functional requirements. Then, this model is transformed into an AO architecture model. Transformations are implemented using the QVT-relational notation. Finally, an architecture for a simple scenario is obtained. This architecture satisfies functional and non-functional requirements, preserves trade-offs, and it is better modularized because crosscutting concerns are well encapsulated, avoiding scattering and tangling, by using AO techniques.

As future work, we will investigate the conflicts, dependencies and interactions between scenarios, how to deal with requirements information which cannot be naturally mapped into an architecture and more powerful techniques for trade-off analysis. As part of our ongoing work, we are testing different ways of implementing transformations. Currently, simple transformations have been implemented by means of XSLT sheets, but we are investigating the adoption of specific transformation languages, such as ATL, which decrease the complexity of dealing with XMI.

References

1. Tarr, P., Ossher, H., Sutton Jr., S.M., Harrison, W.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Aspect-Oriented Software Development. Adisson-Wesley (2005) 37–61
2. AOSD-NoE: Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. AOSD-Europe Deliverable D11 (2005)
3. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design : The Theme Approach. 1 edn. Addison-Wesley Professional (2005)
4. Chitchyan, R., Pinto, M., Fuentes, L., Rashid, A.: Relating AO Requirements to AO Architecture. In: Workshop on Early Aspects, 20th Int. Conference on Object-Oriented Programming, Tools and Applications (OOPSLA), San Diego, California (USA) (2005)
5. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Adisson-Wesley (2004)
6. Araujo, I., Weiss, M.: Linking patterns and non-functional requirements. In: 9th Conference on Pattern Language of Programs (PLoP), Monticello Illinois (USA) (2002)
7. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2001)
8. Pinto, M., Fuentes, L., Troya, J.M.: A Dynamic Component and Aspect-Oriented Platform. The Computer Journal **48**(4) (2005) 401–420
9. Brito, I., Moreira, A.: Advanced Separation of Concerns for Requirements Engineering. In: In Proc. of the 8th Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Alicante (Spain) (2003) 47–56
10. Brito, I., Moreira, A.: Integrating the NFR framework in a RE model. In: Workshop on Early-Aspects, 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, (UK) (2004)
11. Soeiro, E., Brito, I., Moreira, A.: An XML-Based Language for Specification and Composition of Aspectual Concerns. In: 8th International Conference on Enterprise Information Systems (ICEIS), Paphos (Cyprus) (2006)
12. Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software **23**(1) (2006) 61–70
13. Araújo, J., Wittle, J., Kim, D.: Modeling and Composing Scenario-Based Requirements with Aspects. In: Proc. of the 12th International Requirements Engineering Conference (RE), Kyoto (Japan), IEEE CS Press (2004) 58–67
14. Object Management Group (OMG): MOF QVT Final Adopted Specification (ptc/05-11-01). http://www.omg.org/docs/ptc/05-11-01.pdf (2005)
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proc. of the European Conference on Object-Oriented Programming (ECOOP). Volume 2072 of Lecture Notes in Computer Science., Budapest (Hungary), Springer (2001) 327–355
16. Aksit, M.: Systematic analysis of crosscutting concerns in the model-driven architecture design approach. In: Symposium on how Adaptable is MDA?, Twente (The Netherlands) (2005)
17. Amaya, P., Gonzalez, C., Murillo, J.: Towards a subject-oriented model-driven framework. In: 1st Int. Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (AB-MB-SoC), 1st European Conference on MDA - Foundations and Applications (ECMDA-FA)), Nuremberg (Germany) (2005)
18. Kulkarni, V., Reddy, S.: Integrating aspects with model driven software development. In Al-Ani, B., Arabnia, H., Mun, Y., eds.: Software Engineering Research and Practice, Las Vegas, Nevada (USA), CSREA Press (2003) 186–197

19. Simmonds, D., Solberg, A., Reddy, R., France, R., Ghosh, S.: An aspect oriented model driven framework. In: Proc. of the 9th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Twente, (The Netherlands), IEEE CS (2005) 119–130
20. Grundy, J.: Multi-perspective specification, design and implementation of software components using aspects. International Journal of Software Engineering and Knowledge Engineering **10**(6) (2000) 713–734
21. Tekinerdogan, B.: ASAAM: Aspectual Software Architecture Analysis Method. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), Oslo (Norway), IEEE CS Press (2004).

Modeling and Analyzing Mobile Software Architectures

Clemens Schäfer

Chair for Applied Telematics / e-Business*
University of Leipzig, Germany
schaefer@ebus.informatik.uni-leipzig.de

Abstract. The emerging behavior of a mobile system is determined by its software architecture (structure, dynamics, deployment), the underlying communication networks (topology, properties like bandwidth etc.) and interactions undertaken by the users of the system. In order to assess whether a mobile system fulfills its non-functional requirements like response times or availability already at design time, the emergent behavior of such a system can be simulated by using an architectural model of the system and applying an simulation approach where a network model and a user interaction model are used for providing the contextual information.

In this paper we show how such an architectural model can expressed in our ADL Con Moto, how functional and non-functional properties of an architecture can be modeled and how simulation of the mobile system can be used to yield the desired properties.

1 Motivation

Modeling the architecture of mobile distributed systems using a domain-specific architecture description language (ADL) is considered as an useful approach [3], since the influence of mobility emphasizes the necessity to examine functional properties of software architectures as well as non-functional properties. This corresponds to the fact that "mobility represents a total meltdown of all stability assumptions ... associated with distributed computing" [15], which subsumes the problems software engineers have to face in practice when they build mobile distributed systems. Examples for these problems are network structures, which are no longer fixed and where nodes may come and go, communication failures due to lost links over wireless networks, or restricted connectivity due to low bandwidth of mobile communications links. These all have in common that they affect the emergent non-functional properties of a system like performance, robustness, security or quality of service. Besides non-functional properties, these intrinsic challenges of mobile systems may also affect the functional aspects of a system, since a mobile system may have to provide extra functionality like replication facilities or caching mechanisms in order to ensure usability in situations where the aforementioned problems occur. With our ADL *Con Moto* (Italian

* The chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 175–188, 2006.
© Springer-Verlag Berlin Heidelberg 2006

for "with motion") we propose a language which enables system developers to address these issues during the early stages of system development in order to allow them to make appropriate design choices for the mobile system.

2 Introduction

Mobile systems show complex emergent behavior due to the combination of software aspects with telecommunication issues and the therefore eroding stability assumptions. In order to determine whether a mobile system fulfills non-functional requirements like response time or availability of service, a quite complex model of the system is needed.

1. The model must reflect the system's physical structure, comprising physical components (devices) and physical connectors (communication links, network topology) as well as the properties of these items like bandwidth or bandwidth distribution and computational resources, since for example a mobile component might take more time being executed on a mobile client compared to the execution on a server.
2. The logical structure of the system must be modeled in detail, comprising information about software components, their dependencies and deployment on the physical components and the possible changes in the deployment structure.
3. The model has to reflect the dynamics of the system, i.e. the behavior of the logical components, their interactions and the exchanged information.
4. Finally, user interaction with the system must be expressed, specifying how many users are existing and how these users interact with the system.

These aspects show that the challenge in modeling mobile system lies in the need to find an appropriate level of abstraction, since over-simplification will cause meaningless analysis results; however, too detailed models are not practical during the design process. Any modeling approach should remain as abstract and as free from technological implementations of real mobile systems as possible; nevertheless, realistic assumptions about the technological implementation of a mobile system are sometimes necessary to yield feasible simulation results.

The remainder of this paper is structured as follows. First, an overview about related work is given. Next, our approach for modeling mobile systems using Con Moto is presented. After depicting an example system and simulation results for this system, results are discussed.

3 Related Work

ADLs in general have been a topic of research in previous years. The necessity for modeling non-functional properties in architecture description has been recognized by Shaw and Garlan [16]. The classification work of Medvidovic and Taylor [8] presents a sound compilation of properties of existing ADLs. From their work it becomes obvious, that none of the ADLs presented there is suitable

for modeling dynamic aspects of mobile systems. In the past, this fact lead to the development of mobile ADLs which have recently been presented. The ArchWare project with its π-ADL [12] is one result of these efforts. Another mobile ADL can be found in the works of Issarny et al. [5]. Both present an ADL for mobile systems based on Milner's π-calculus [9]. These two ADLs have in common that they are able to model the dynamics of mobile systems, which is due to their theoretical foundation in the π-calculus. Although they vary in terms of elaboration and tool support, the fundamental difference—from the perspective of this paper—is the treatment of non-functional properties, which is absent in the π-calculus ADL approach. Issarny et al. address non-functional properties in their work, but the treatment of non-functional properties is bound to a global conformance condition, which must hold for a predefined set of non-functional properties assigned to components and connectors, and does not allow the composition of non-functional properties, which is novel in our approach. Besides the design of mobile ADLs there is other research in the area of non-functional properties of software systems. This work is mainly based on the Lamport's TLA+ language [6], which is a logic for specifying and reasoning about concurrent and reactive systems. Zschaler [17] presents a specification of timeliness properties of component based systems, but these as well as the underlying work of Aagedal [1], where the integration of TLA+ approach into architectural description is proposed, are not regarded further in our context, since the models in TLA+ lack the support for mobility. Other approaches based on Markov Chains and process algebras (e.g. the work of Hermanns and Katoen [4]) are not promising for out purposes, since fall short for the support for mobility.

4 Approach

In the following we describe the constituents of the Con Moto approach. All elements in the following are necessary to derive properties like bandwidth utilization, network congestion, dynamic evolvement of software deployment, transaction times or service availability for a system under analysis. Retrieving these properties during simulation is quite straight-forward if an appropriate representation of the mobile system and its usage is chosen.

Figure 1 shows an overview about the different elements of a Con Moto model and the simulation environment: The core architectural model is made from a behavioral and a structural specification of the system. This is due to the fact that in addition to the obviously existing structural model of mobile systems their behavior influences evolvment of the architecture and thus has to be modeled as well. Together with instantiation information, the simulator can create instances of the architectural model for simulation purposes. During simulation, communication network structures will be provided for the system as they are modeled in the network model. By applying user interactions by instantiating the Usage Patterns, the modeled system can evolve in the simulator and the evaluation results can be calculated. In the following, we will present the different aspects of this model and exemplify their use by showing an example.

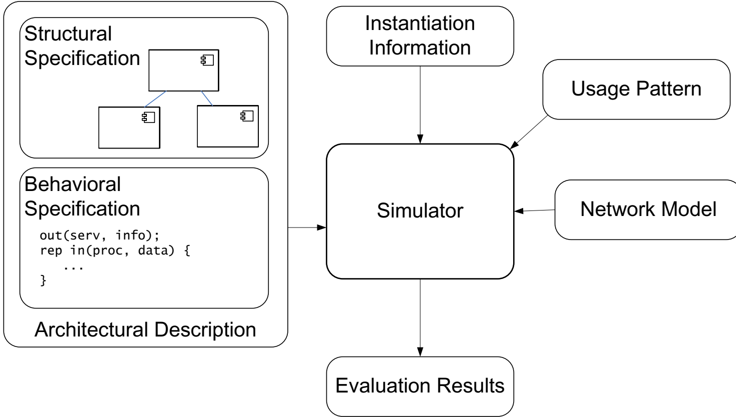


Fig. 1. Con Moto constituents

4.1 Behavioral Model

Mobile systems have to react to external conditions; the dynamically changing configuration is inherent to mobile systems. Therefore it makes sense to base architectural modeling on a behavioral model, assuming that any structural aspects like components or connectors can be seen as constraints for the behavioral model of the system.

Like other ADLs for mobile systems [13], we build our behavioral model on π-calculus. π-Calculus [10] is a process algebra with explicit support for mobility. It is based on communication primitives which allow the exchange of processes or communication nodes among processes. However, π-calculus in its full beauty offers features which are not necessary for our approach. Since we build a simulation environment, only constructs which reflect typical programming situations are used; others are discarded for the sake of simplicity. Such a restriction has also been done in the work of Pierce and Turner: with Pict [14] they present a π-calculus-based programming language, where they also omit some features of core-π-calculus, slightly reducing expressive power, but removing nondeterminism and making it appropriate for programmers.

As shown in Table 1, Con Moto provides different constructs for modeling processes: The output action allows the communication of an object over a so-called *Pin* in Con Moto (in π-calculus, the pins are called *names*). Other than in Pict, we only allow the synchronous output like in π-calculus, since we decouple input and output by means of the connectors.

Similar to Pict, we restrict π-calculus's replication prefix to input statements. Hence we do not allow the replication of processes; nevertheless, new processes can be created together with input operations, which is a quite realistic assumption, as it allows easily the creation of processes which respond to input data. The choice operator as a source for nondeterminism is omitted, but a if/then/else construct is added.

Table 1. Notation

π-Calculus	Con Moto	
$\overline{x}y$	`out(x,y)`	synchronous output
$x(y)$	`in(x,y)`	input
$e_1 \mid e_2$	`par e1, e2`	parallel composition
$(\nu x)e$	`new x; e`	channel creation
$!x(y).e$	`rep in(x,y) e`	replicated input

Modeling behavior includes messages that will be exchanged by processes will be implemented in Con Moto. Usually, abstractions of real-world messages are used in such situations; only that portion of a message is modeled, which is absolutely necessary to reflect the message's impact on control flow and behavior of the system. In Con Moto, we also specify meta-information about the size of messages, because in simulation situation the real-world size of such objects is necessary for simulation, hence supporting non-functional properties, since these meta information can be used e.g. by the network part of the model to calculate transmission times etc..

4.2 Structural Model

Having identified the processes as basis for the model of a mobile system, structural information has to be added since a solely behavioral view of the system would be unappropriate. Therefore, a structural model of the mobile system is set up. The challenges are twofold: On the one hand we now need an abstraction which allows us to set up a decomposition of a mobile system and on the other hand we need some decision on what the smallest entity of mobile code is.

Structural aspects have been considered in all ADLs so far. It is commonly accepted that an structural model comprises components, connectors and configurations. The components are the *locus computandi*: calculations are preformed on the components, whereas connectors model the communication relationships among components. Configuration can be seen as the state of a system and represents all interconnections between components by means of connectors.

Components. For modeling mobile systems we have to clarify the notion of components and connectors. In Con Moto, we distinguish between *physical components* and *logical components*. Physical components are devices like PDAs or servers, are constrained in their resources (memory size, CPU power etc.) and act as execution environment for logical components. Logical components model software components. They do not have resource constraints in our understanding and can occur as components and component instances. Instances of logical components have a state. In order to allow communication, physical as well as logical components have ports, which are aggregations of ports and pins, which finally allow the interconnected processes to communicate.

Connectors. In Con Moto there are two different kinds of connectors, namely *physical connectors* and *logical connectors*. Logical connectors are used for communication between logical components and are ideal: they have an unbounded bandwidth and null latency. In contrast, physical connectors connect physical components and these are not ideal, having a limited bandwidth and a latency time greater than zero.

Logical connectors can be embedded in physical connectors. This is necessary, if logical components on different physical components shall communicate. The logical connector between the two connected logical components is embedded in the physical connector between the two physical components, which act as the execution environments for the two logical components.

Mobility. Components are the smallest entity of mobile code in Con Moto. We assume that the *component* should be the element which is mobile. We do not take the extreme view of Mascolo et al. that every line of code is potentially mobile [7], because we want to model systems where this assumption would be unrealistic. We allow logical components as well as logical component instances to be communicated among processes. The same is true for logical connectors. This allows us to cover all kinds of mobility which are shown in the work of Fuggetta et al. [2]:

- *Client-server*, where a data file f is transferred from a node n_u to a node n_p. A program p executes on node n_p and the results are transferred to node n_u. The client on node n_u controls the operation. This is the situation as shown in our example below.
- *Remote evaluation*, where a program p is transferred from node n_u to node n_p, and executed there. Results are returned to n_u. The client controls the operation. Using Con Moto, this can be expressed by sending a logical component (which is the program p) to the computing node.
- *Code-on-demand*. Data file f and program p are transferred to n_u and executes there. The user demanding the code controls the operation.
- *Mobile agents*. Program p is transferred to n_f and executes there. Results are transferred to n_u. The agent itself controls the operation.

Configuration. It is obvious that configuration of mobile systems evolve over time, since components can connect and disconnect to other components due to their behavior. For mobile systems, however, developers usually express constraints on the possible configurations which might occur. By means of deployment diagrams like in UML 2.0 [11], developers of systems can express where components are deployed, hence which logical components are placed on which physical components. However, to be able to express constraints for configuration evolvement, this is not sufficient. Besides expressing an initial state of the deployment, there should be the possibility of expressing where components may be deployed during runtime, because then and only then runtime checks are possible whether the configuration of an mobile system evolves correctly.

Architectural Connection. Architectural connection, hence the way how components are connected to each other by means of connectors, is a crucial aspect for mobile systems, since here all imponderabilities of mobility arise. For realistic systems, there may be many and complex dependencies among logical components leading to many logical connections. Physical connections are fewer: usually only a small number of physical connectors from among physical components.

In our system, logical connections must be embedded in physical connections; logical connections hence cannot be ideal–there is no synchronicity or parallelism.

In order to allow different communication protocols like synchronous calls (e.g. Remote Procedure Calls, Service Invocation) and asynchronous communication (events), our approach using pins where processes can exchange information is sufficient. Nevertheless, when a system is modeled on a quite high-level basis, there is the requirement for provides– and uses–interfaces and for services.

In order to provide a general basis, we introduce in Con Moto the possibility of ports which can consist of other ports and pins. By expressing bind rules, high-level ports can be connected, and by resolving the port hierarchy and subsequent application of binding rules various pins will be connected.

5 Example System

For illustration purposes we will use a simple example system. This example system is a mobile client/server system. The users of the mobile system carry mobile devices, which are connected to a server via mobile communication links; in our example, we provide either an GPRS link, which has a rather low bandwidth, and an UMTS (3G) link, which has a higher bandwidth. There are three software components in the example system: a user interface component (UI) is deployed on the mobile devices; a database component (DATA) is deployed on the server. The actual business logic of our system is captured in the component BUSINESS, which is a mobile component and thus can be either deployed on the server or on the mobile devices. When the user invokes a service of the UI component, a request is sent to the BUSINESS component (either on the mobile device or on the server). This component itself invokes a service of the DATA component before it returns its calculation results to the UI component. The structure of the example system is shown in Figure 2.

5.1 Modeling in Con Moto

At the end of this paper, the Con Moto code, which is actually a document in an XML dialect, of the described example is shown. The two hardware components `MOBILE` and `SERVER` are declared in the section `<physical-components>`. For both, their CPU power is set and the possible connections to the network, which ends up in physical connectors during simulation. The `<network-access>` for `MOBILE` allows connection either to UMTS or GPRS network, the `SERVER` can only connect to the WAN.

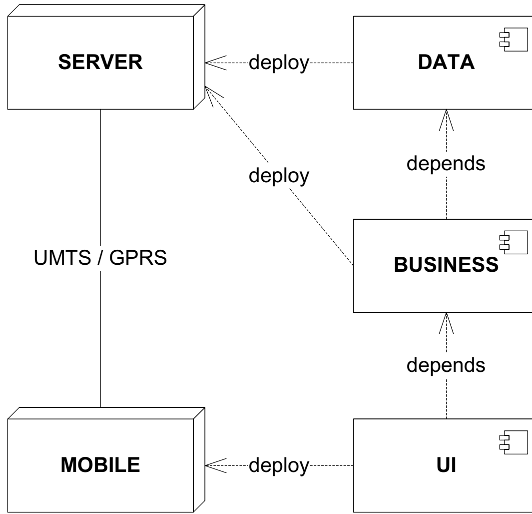


Fig. 2. Example system

The actual network model is given in the section `<network-config>`. Here, the network types UMTS, GPRS and WAN are defined. For all these network types, the bandwidth is specified (10.0, 2.0 and 1000.0 kBit/s). Latency times are not given. An additional network node named `backbone` is also given. All network connections via UMTS, GPRS and WAN automatically connect to this backbone, allowing to address any device from any other device which is connected to the network, i.e. physical components can communicate when they have connected to the network–which is a model similar to the internet. For UMTS and GPRS nodes in the network, we define that these nodes are equally distributed, which is necessary information if during simulation the number of network nodes is increased.

By introducing ports and port hierarchies in the section `<connection>` it is possible to have complex ports which act as an method provider interface or method invoker interface. By specifying macros for ports a certain behavior can be implemented in the port definition and easily reused in the actual process definition. In the example, the invokation of a service is modeled as a macro in port `methodInvoker`. Since port `methodsProvider` has an extendable process which provides the counterpart for this macro, method invocation, waiting for execution and returning of a method result can be specified in π-calculus using `in` and `out` command on pins. In the processes in definition of the logical components, however, these macros and processes can be reused, yielding a code which is structurally equivalent to code in an imperative programming environment.

The logical components `DATA`, `BUSINESS` and `UI` are specified in the section `logical-components`. For the components `BUSINESS` and `UI` startup processes are defined, which execute when the components are deployed. During these processes, lookups of the components (`BUSINESS` in case of `UI` and `DATA` in case

of BUSINESS) are performed and the logical connections to the components are established.

The processes of the methodProvider ports are extended for implementing services on components, such that the action which is to be undertaken after a service has been called is implemented in the processes on the logical components. On DATA the service getData sets a size of the return package of 100 bytes and blocks the CPU for 100ms. This return package size is used by the simulator to calculate the transmission time through the network. On BUSINESS the service getInfo makes a call to getData before a return package size of 5000 bytes is set and the CPU is blocked for 500ms.

5.2 Simulation

We have simulated the example system described here using our Con Moto simulator and have varied the users (and respectively, the MOBILE devices) from 10 to 150. The users use the system as modeled by a Poisson-process with an arrival rate of 10 per hour. The simulations have been performed for an time resolution of 1ms, and each simulation took not more than approximately 10 seconds on a 2 GHz Pentium PC, using a prototypical implementation of the simulator written in Java. Figure 3 shows the simulations results, meaning that starting with 90 users, the system gets increasingly congested and the response times of the services at the UI component increase drastically. Differences can be seen in the response times of GPRS and UMTS, which is due to the higher bandwidth of UMTS compared to GPRS.

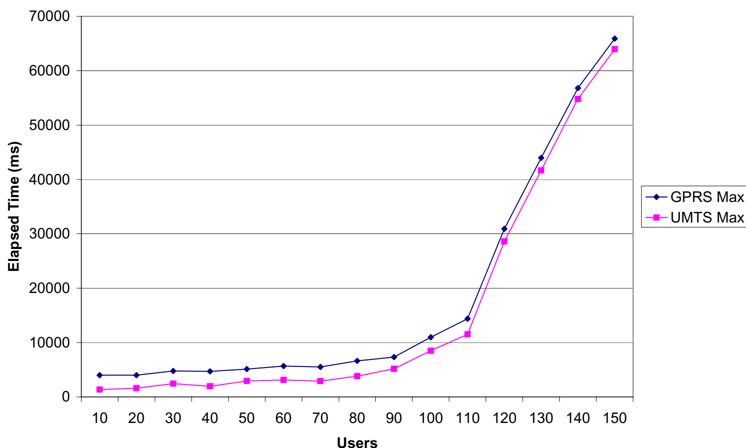


Fig. 3. Simulation result

6 Discussion

In this paper we have described how mobile systems can be modeled using the Con Moto approach with the goal of determining quality-of-service parameters

during design time by means of an simulation approach. By basing an architectural description on π-calculus and making a clear distinction between logical and physical components and connectors, modeling of mobile systems on a quite high level is possible with feasible effort. First simulation results on an toy example system show that the general approach is promising. Nevertheless, further formalization of the approach is necessary and subject to ongoing work.

Areas of further work are the discussion of models for physical communication channels. So far, we assume just constant bandwidth and latency time, but more complex models of modeling transmission characteristics of communication channels and–especially–availability characteristics of these channels are necessary for realistic simulation results. The area of user interaction with a mobile system is also part of further investigation, since not only the stochastic processes for user behavior need careful consideration–also the question how to derive user interaction patterns suitable from simulation from business process models is interesting. Finally, evaluation of the approach by comparing simulation results to real-world measurements is a future task.

References

1. J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems.* PhD thesis, University of Oslo, 2001.
2. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
3. V. Gruhn and C. Schäfer. Architecture Description for Mobile Distributed Systems. In *Proceedings of the Second European Workshop on Software Architecture (EWSA 2005)*, pages 239–246. Springer-Verlag Berlin Heidelberg, 2005.
4. H. Hermanns and J.-P. Katoen. Performance Evaluation := (Process Algebra + Model Checking) × Markov Chains. In *Proceedings of CONCUR 2001*, LNCS 2154, pages 59–81. Springer-Verlag Berlin Heidelberg, 2001.
5. V. Issarny, F. Tartanoglu, J. Liu, and F. Sailhan. Software Architecture for Mobile Distributed Computing. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 201–210. IEEE, 2004.
6. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.
7. C. Mascolo, G. P. Picco, and G.-C. Roman. A fine-grained model for code mobility. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 39–56, London, UK, 1999. Springer-Verlag.
8. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Januar 2000.
9. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
10. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
11. OMG. Unified Modeling Language (UML) Specification: Superstructure, Version 2.0 (formal/05-07-04).

12. F. Oquendo. π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM Software Engineering Notes*, 29, Mai 2004.
13. F. Oquendo. π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM Software Engineering Notes*, 29, May 2004.
14. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
15. G.-C. Roman, G. P. Picco, and A. L. Murphy. Software Engineering for Mobility: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 241–258. ACM Press, 2000.
16. M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1995.
17. S. Zschaler. Formal specification of non-functional properties of component-based software. In J.-M. Bruel, G. Georg, H. Hussmann, I. Ober, C. Pohl, J. Whittle, and S. Zschaler, editors, *Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004*, September 2004.

Example Code

```
<system>
  <connection>
    <port-role name="in" />
    <port-role name="out" />

    <port-role name="methodsInvoker" extends-role="out" >
      <ports name="methodInvoker" />
    </port-role>

    <port-role name="methodInvoker" >
      <pin name="call" />
      <pin name="return" />

      <macro>
        <parameter name="argument" />
        <result name="result" />
        <pi>
          out(call, argument);
          in(return, result);
        </pi>
      </macro>
    </port-role>

    <port-role name="methodsProvider" extends-role="in" >
      <ports name="methodProvider" />
    </port-role>

    <port-role name="methodProvider" >
      <pin name="invoke" />
      <pin name="response" />

      <process>
        <pi>
          object arg, result;
          rep in(invoke, arg) {
            <extension-point />
            out(response, result);
```

```
      }
    </pi>
  </process>
</port-role>

<bind-rule>
  <scope>
    <from>methodsInvoker</from>
    <to>methodsProvider</to>
  </scope>
  <bind>
    <from>methodsInvoker.methodInvoker</from>
    <to>methodsProvider.methodProvider</to>
  </bind>
</bind-rule>

<bind-rule>
  <scope>
    <from>methodInvoker</from>
    <to>methodProvider</to>
  </scope>
  <bind>
    <from>methodInvoker.call</from>
    <to>methodProvider.invoke</to>
  </bind>
  <bind>
    <from>methodInvoker.response</from>
    <to>methodProvider.return</to>
  </bind>
</bind-rule>
</connection>


<network-config>
  <passive-node name="backbone" />

  <active-node name="UMTS">
     <multiplicity>0.5</multiplicity>
     <auto-link>
       <node>backbone</node>
       <bandwidth>10.0</bandwidth>
     </auto-link>
  </active-node>

  <active-node name="GPRS">
     <multiplicity>0.5</multiplicity>
     <auto-link>
       <node>backbone</node>
       <bandwidth>2.0</bandwidth>
     </auto-link>
  </active-node>

  <active-node name="WAN">
     <multiplicity>unbounded</multiplicity>
     <auto-link>
       <node>backbone</node>
       <bandwidth>1000.0</bandwidth>
     </auto-link>
  </active-node>

</network-config>


<logical-components>
  <component name="DATA">

    <port type="methodProvider" name="getData">
      <extend-process>
```

```
        <pi>
          result.size = 100;
          useCpu(100);
        </pi>
      </extend-process>
    </port>
  </component>

  <component name="BUSINESS">
    <size>200</size>

    <start-process>
      <pi>
        PhysComp remoteHW = lookupPhysComp("SERVER");
        LogComp remoteSW = remoteHW.lookupLogComp("DATA");
        connect(this.getData, remoteSW.getData);
      </pi>
    </start-process>

    <port type="methodInvoker" name="getData" />

    <port type="methodProvider" name="getInfo" >
      <extend-process>
        <pi>
          object res;
          object par;
          res = getData(par);
          result.size = 5000;
          useCpu(500);
        </pi>
      </extend-process>
    </port>
  </component>

  <component name="UI">
    <port type="methodInvoker" name="getInfo" />

    <start-process>
      <pi>
        PhysComp remoteHW = lookupPhysComp("SERVER");
        LogComp remoteSW = remoteHW.lookupLogComp("BUSINESS");
        connect(this.getInfo, remoteSW.getInfo);
      </pi>
    </start-process>

    <pin name="action">
      <process>
        <pi>
          object dummy;
          rep in(action, dummy) { getInfo(dummy); }
        </pi>
      </process>
    </pin>
  </component>
</logical-components>

<physical-components>
  <component name="MOBILE">
    <memory>unbounded</memory>
    <cpu>10</cpu>

    <network-access>
      <xor>
        <type>UMTS</type>
        <type>GPRS</type>
      </xor>
    </network-access>
```

```
      <logical-component-deployment>
        <name>UI</name>
        <instance>on-start</instance>
      </logical-component-deployment>

      <logical-component-deployment>
        <name>BUSINESS</name>
        <instance>client-controlled</instance>
      </logical-component-deployment>

    </component>

    <component name="SERVER">
      <memory>unbounded</memory>
      <cpu>1000</cpu>

      <network-access>
         <type>WAN</type>
      </network-access>

      <logical-component-deployment>
        <name>BUSINESS</name>
        <instance>on-start</instance>
      </logical-component-deployment>

      <logical-component-deployment>
        <name>DATA</name>
        <instance>on-start</instance>
      </logical-component-deployment>
    </component>

  </physical-components>

</system>
```

Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design

Holger Schmidt and Ina Wentzlaff

University Duisburg-Essen, Faculty of Engineering, Department of Computer Science, Workgroup Software Engineering, Germany
{holger.schmidt, ina.wentzlaff}@uni-duisburg-essen.de

Abstract. In this paper, we present a pattern-based software development method that preserves usability and security quality characteristics using a role-driven mapping of requirements analysis documents to architectural design artifacts. The quality characteristics usability and security are captured using specialized problem frames, which are patterns that serve to structure, characterize, and analyze a given software development problem. Each problem frame is equipped with a set of appropriate architectural styles and design patterns reflecting usability and security aspects. Instances of these architectural patterns constitute solutions of the initially given software development problem. We illustrate our approach by the example of a chat system.

1 Introduction

Besides the functional aspects of a software system, a software engineer must face *quality characteristics* such as security and usability. In general, all software systems have quality requirements, even if they are often acquired insufficiently and less considered compared to functional aspects during the software development life cycle. Causing serious damage to the economy (e.g., a stock market system, market share, and sales market of software product), endangering personal privacy or threatening people's life (e.g., a medical chip card system, traffic accidents, or airplane disasters) can be possible consequences if software neglects usability needs or security demands. Many security-critical software systems fail because their designers protected the wrong things, or protect the right things but in the wrong way. Inadequate usability is a reason for user activities causing undesired and dangerous software system effects. Thus, adequate security and usability engineering requires to have an explicit understanding of the security and usability requirements and to provide effective techniques to accomplish them.

Knowing that building systems with security and usability demands is a highly sensitive process, it is important to reuse the experience of commonly encountered challenges in these fields. This idea of using *patterns* has proved to be of value in software engineering for years, and it is also a promising approach in security and usability engineering. Patterns are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure or behavior. Patterns are defined for different activities at different stages of the software development lifecycle. *Problem frames* [9] are patterns that classify software development *problems*. *Architectural styles* are patterns that

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 189–203, 2006.
© Springer-Verlag Berlin Heidelberg 2006

characterize software architectures [2]. In Software Engineering *design patterns* [6] are commonly used for finer-grained software design and they are as well used for coarse-grained architectural design.

Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, instead of starting from scratch each time. The problem frames defined by Jackson [9] cover a large number of software development problems, because they are quite general in nature. To support software development in more specific areas such as security and usability engineering, however, *HCI-oriented problem frames (HCIFrames)* [18] have been developed for usability engineering, while *security problem frames* and *concretized security problem frames* [8] have been developed for security engineering.

In this paper, we show how to use the problem frames approach in the area of security and usability engineering to develop architectures. We propose a pattern-based method developed for *preserving* security and usability characteristics from requirements analysis to architectural design. Section 2 gives an overview of this method. Initially, a software engineer must understand the context of a software development problem and decompose the overall problem situation into smaller subproblems (Sections 2.1 and 2.2). For this purpose, we apply problem frames defined by Jackson [9] and specialized problem frames for security and usability demands (Sect. 3). To preserve the quality characteristics identified and collected using specialized problem frames, we equip each problem frame with corresponding architectural patterns. Then, entities, facets, and their interactions in the problem description represented within the instantiated problem frames are mapped by a role-driven process to corresponding components and classes of the solution description (Sect. 4).

Additionally, security and usability problem frames can be systematically transformed into notations of the Unified Modeling Language (UML) [17]. This increases their value in later software development phases. Thus, we obtain a software design based on commonly known architectural patterns and achieve a seamless transition from requirements analysis to software design, preserving quality characteristics. We illustrate our approach by developing a chat system, and conclude our in Sect. 5.

2 A Pattern-Based Software Development Approach

We propose a pattern-based software development process consisting of four steps, which will be described in detail in the following sections:

1. Understand the problem situation (Sect. 2.1)
2. Decompose overall problem into simple subproblems (Sect. 2.2)
3. Fit subproblems to problem frames (Sect. 3.1)
   (a) Identify quality characteristics (Sect. 3.2)
   (b) Classify subproblems according to quality demands (Sect. 3.3)
4. Instantiate corresponding architectural and design patterns (Sect. 4)

We illustrate our approach by the pattern-based development of a chat application, starting from the requirements analysis and leading to the derivation of software design artifacts.

The starting point for the analysis of our software development project (the *system mission*) can be outlined in one simple sentence:

"A text-message-based communication platform shall be developed which allows multi-user communication via private I/O-devices."

Requirements describe the application environment when our developed software is in operation. They represent desired properties of the problem domain. In contrast, *domain knowledge* describes given properties of the environment (*facts*) and important environmental conditions (*assumptions*). Desired and given properties of the problem domain are summarized by a context diagram. It describes the overall problem situation, which we want to improve through our software product.

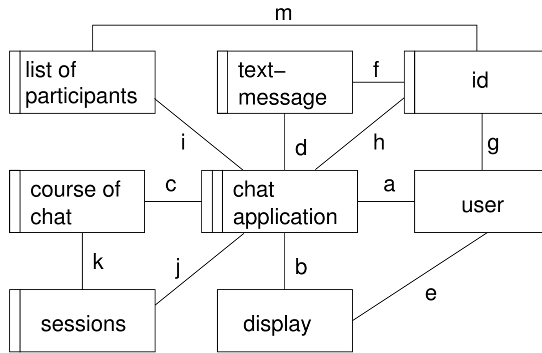
In Tab. 1 requirements `R1` - `R10` and domain knowledge (consisting of some facts `F1` - `F2` and assumptions `A1` - `A2`) are collected to elaborate our system mission.

Table 1. Requirements and domain knowledge for the chat application

R1	Users can phrase text-messages, which are shown on their private graphical displays.
R2	Users send their phrased text-messages to the chat, which are stored in the public course of the chat in their correct temporal order.
R3	The course of the chat is shown to the users on their private graphical displays.
R4	Sending text-messages changes the presentation of the course of the chat on the user's graphical displays.
R5	Each text-message is related to its respective user, so that the originator of a message can be identified.
R6	All users are stored in a list of participants, which is visible to every chat user.
R7	To each course of the chat a corresponding list of participants is shown.
R8	Various chat sessions considering different subjects of discussion are offered to the users.
R9	All available chat sessions are shown to the users.
R10	Users can switch among different chat sessions.
F1	Users can only understand the course of the chat, if the text-messages are presented in their correct temporal order (First In - First Out (*FIFO*)).
F2	If more than two users participate in the chat, it is required to relate messages to their originators in order to maintain a comprehensible chat communication.
A1	Users will follow the course of the chat on their private graphical display.
A2	Several users will participate in the chat.

2.1 Understand the Problem Situation

In the terminology of Jackson [9], the software development goal is to build a *machine* that changes the environment in a specified way. Thus, an intensive investigation of the given and desired properties of the problem environment is mandatory (cf. Tab. 1). This requirements and domain analysis process is accompanied using a *context diagram* that represents the interactions between the machine (software to be developed) and its application environment (see Fig. 1). It shows where in the environment the software development problem is located.



a: {phraseTextMessage, sendTextMessage, login, signUpChatSession, signOffChatSession}
b: {showTM, showTMMeta, showTMDefault, showUserID, showCourseOfChat, showTransformationInProcess, showListOfParticipants}
c: {registerTM, recordUserID, CourseOfChatContent}
d: {editTextmessage,TMContent, TMMeta, TMDefault}
e: {followCourseOfChat}
f: {IdentifiesTMOriginator}
g: {Correspond2OneAnother}
h: {relateUser2ID, UserID}
i: {UserList, registerUserID}
j: {AvailableSessions}
k: {collectOfferedChatSessions}
m: {enterUsersIntoListOfParticipants}

Fig. 1. Context diagram of the chat application

A context diagram consists of *domains* (rectangles) and sets of *shared phenomena* (labeled links between rectangles), which are derived from the requirements and domain knowledge (cf. Tab. 1). Domains correspond to entities or facets of the real world, whereas the *machine domain* (rectangle with two vertical lines) represents the software product which ought to be developed, in our case: the chat application itself. Hence, there is exactly one machine domain contained in a context diagram (chat application in the center of Fig. 1). Any arbitrary number of additional domains can be part of the overall problem situation. They can be further classified. Data types, data structures, database schemata, or other representations of information, that need to be built and introduced by the software developer, are denoted by *designed domains* (rectangle with only one vertical line), for instance text-message. *Given domains* (simple rectangles) are concepts of the real world, which already exist and do not need to be constructed. They are relevant for the problem description and its solution, and need to be considered in the context diagram, too, for instance display.

Shared phenomena are operations, actions, events, or states which are common to two domains. For instance, the machine domain chat application shares the phenomenon AvailableSessions (which is derived from requirement `R8`) with the designed domain sessions, cf. interface j in Fig. 1. Context diagram, requirements, and domain knowledge are created and collected iteratively to cover the overall problem situation and help to understand the given and desired interactions of environment and machine. Requirements and domain knowledge that are found through software analysis are expressed by domains and shared phenomena in the context diagram (cf. Tab. 1 and Fig. 1).

2.2 Decompose Overall Problem into Simple Subproblems

As the context diagram in Fig. 1 illustrates, it would become difficult to start a structured software development process based on such a complex problem situation. The problem needs to be split into simple subproblems for which known solution methods are available. This is achieved by decomposing the context diagram with the help of the requirements by means of *knowledge-based projection* into smaller subproblems. For each subproblem, all other subproblems are assumed as already solved (*separation of concerns*). The subproblems are derived from the context diagram using operators for problem decomposition (e.g., by combining domains or omitting shared phenomena). Those simple and independent subproblems are represented by instantiated problem frames in the following.

3 Patterns for Software Development Problems

Problem frames [9] are patterns to structure and classify software development problems. Each problem frame is represented by a frame diagram, which relates a set of requirements via several problem domains to the machine domain, using shared phenomena. The outcome is a fixed and abstract problem structure.

A problem frame needs to be instantiated by concrete problem content, taken out of the context diagram with the help of requirements. An instance of a problem frame shows the relation of the respective requirements to the particular domains of the corresponding problem context, which are relevant to reflect the requirements.

Requirements describe desired properties of the environment after the machine is in action. In contrast, shared phenomena at the interface of machine and environment are used to formulate the *specifications*, which are descriptions that are sufficient to develop the machine.

Problem frames support the creation of adequate software specifications (e.g., represented using UML sequence diagrams) by elaborating the essential interactions of environment and machine. The specification constitutes the basis for the development of the machine. It is used for software design, coding, testing, and acceptance of the final software product.

Sometimes it is necessary to compose and create new problem frames to be able to detail and classify a certain software development problem more precisely. Therefore, we merged and extended the basic problem frames introduced by Jackson where applicable. Some selected frame instances of the chat application example are presented to exemplify our pattern-based software development approach. In the following, we show the abstract frame diagram elements (in italic style) together with the concrete problem pattern instance (content of a domain and corresponding shared phenomena).

3.1 Fit Subproblems to Problem Frames

Figure 2 shows the instance of the problem frame "*commanded model display*". Its frame diagram is a variant of the "*commanded display*" frame developed by Jackson [9] and an enhancement of the "*query*" frame developed by Choppy and Heisel [3]. The

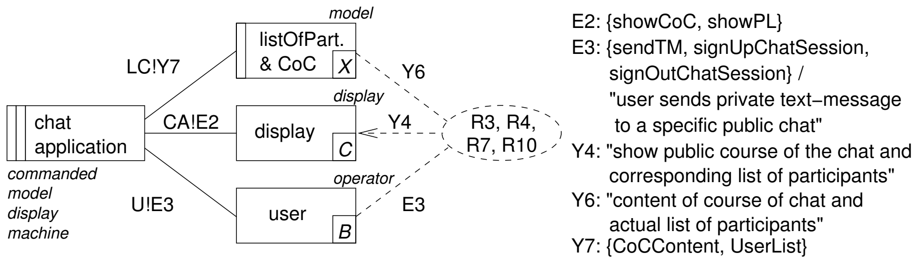


Fig. 2. Instance of the problem frame "commanded model display"

problem frame "*commanded model display*" can be composed out of the basic problem frames "*commanded behaviour*" and "*model display*", too.

The problem frame *"commanded model display"* in Fig. 2 describes the following problem situation: If the *operator* (user) sends a command (phenomenon signUpChatSession of interface E3) to the machine (chat application), it will be executed by the machine and yields some according effects. Here, the state of the *model* (list of participants) is shown on a *display* (display) using the phenomenon showPL of the interface E2. In addition, the actual state of the *model* (public course of the chat) is shown, Fig. 2.

To extract this subproblem from the overall problem, we applied two decomposition operators. We *combine domains*, e.g., list of participants (listOfPart.) and course of the chat (CoC). Additionally, we *omit shared phenomena*, e.g., registerUserID at the interface Y7, because it is not relevant for this subproblem.

Compared to the context diagram in Fig. 1, this frame instance contains only those domains and shared phenomena, which are relevant to fulfill a subset of all requirements namely R3, R4, R7, and R10 (see Tab. 1). The oval on the right-hand side of the frame diagram contains the requirements which are mapped to according parts of the problem context (dashed lines to the domains). The left-hand side of the frame diagram relates the corresponding problem context to the machine. Thus, an instantiated problem frame that is read from right to left indicates the translation of natural-language requirements (in the oval) via the problem context (domains) into technical descriptions which are sufficient to build the machine. Phenomena at the interface of environment and machine can be used to derive *specifications*. The arrowhead pointing to a domain constrains the domain's behavior or characteristics as stated in the requirements. For example, the requirements R3 and R7 in our application example describe a restriction on the domain display.

Each domain in a frame diagram is marked by a character such as X, C, or B to distinguish the different domain types. The designed domain text message is a *lexical domain* (marked X) which has symbolic phenomena associated to it. The display is a *causal domain* (marked C) which does not need to be built but can be controlled using phenomena, too. The user is represented by a *biddable domain* (marked B). His or her behavior cannot be predicted or controlled by the machine. Indeed, the user can make inputs to the software, but cannot be forced by requirements to act in a predetermined way. However, assumptions as part of the domain knowledge are used to make explicit the expected user behavior (cf. Tab. 1).

Symbolic values are indicated by Y, *events* are annotated with E, and C indicates *causal phenomena*. The characters are numbered for indexing the shared phenomena. The exclamation mark (!) specifies which domain *controls* a shared phenomenon. However, this does not imply control flow. For example, LC!Y7 in Fig. 2 in fact expresses that the merged domain list of participants & course of the chat (abbreviated LC) is responsible for administrating the symbolic phenomena in the set Y7. However, the chat application determines when to query the required information. All subproblems contain exactly one machine domain (cf. Fig. 2, Fig. 3, Fig. 4, and Fig. 6). In contrast to Jackson who gives different names to each machine domain, we prefer identical names (here: chat application) to indicate that they constitute parts of a common machine.

3.2 Identify Quality Characteristics

The problem frames approach and more common software development notations such as UML share the same deficiency: they do not offer adequate notations to record software quality characteristic. Although it is commonly accepted that software quality is mainly reflected by non-functional properties (soft goals) [12], only a few approaches exist to elicitate and document them systematically [4]. Software quality characteristics are difficult to grasp, and its hard to maintain them during the software life cycle appropriately. One reason can be that software quality actually is seen as a global attribute of the overall software product, which cannot be related explicitly to local functionality of parts of the software. The idea that software quality is related to the system as a whole rather than to individual system features can be found likewise in requirements engineering [15] and in architectural design [14]. In our approach, we identify and assign quality characteristics to a local set of software functionality. To do so we use problem frames to represent the local behavior and annotate relevant local quality characteristics to them. We extend Jackson's problem frames approach by explicitly annotating software quality characteristics in frame diagrams.

3.3 Classify Subproblems According to Quality Demands

Based on our chat application example, we show how the basic behavior of a system can be expressed using problem frames and how quality characteristics such as usability and security can be considered by detailing the core software features with the help of special usability and security problem frames.

Usability Engineering Using Problem Frames. Usability Engineering contributes to the improvement of human-computer interaction (HCI). It takes psychological aspects into consideration to support the design of software and user interfaces that are easy to use. Although various usability techniques (guidelines, standards, and patterns) exist, they lack of systematical applicability, because often no technical description is available. Some authors of HCI design patterns [16,13] refuse a technical detailing of patterns to keep them comprehensible for non-experts. In contrast, other authors [5] transformed HCI design patterns into UML, but do not offer a process of how and when to apply them exactly. There is an urgent need to integrate usability aspects into

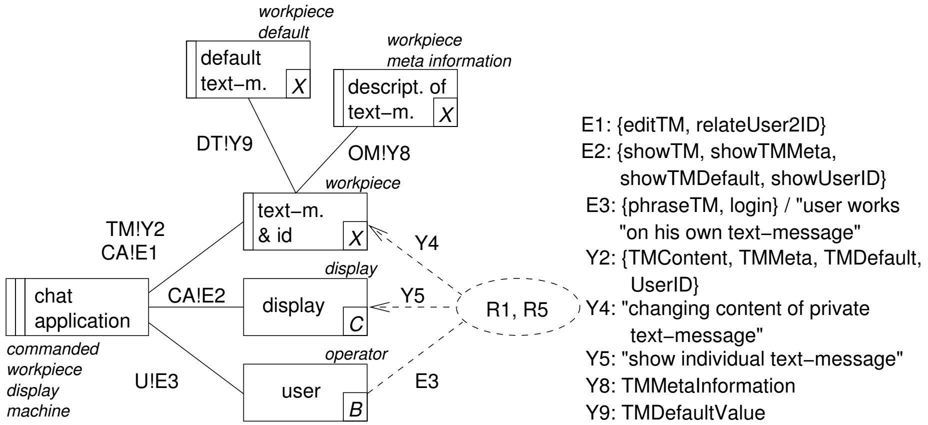


Fig. 3. Instance of the *HCI*Frame "commanded workpiece display"

the software development process. To bridge the gap between informal descriptions of HCI design patterns and the wish to apply usability concerns systematically during the software development process, *the problem descriptions* of HCI design patterns are used in requirements analysis using HCI-oriented problem frames (*HCI*Frames)[18]. *HCI*Frames allow to identify and express usability demands already in the early software development phases. As we will show, usability problems which can be elicited and documented in software analysis can be considered in software design more easily and finally lead to a software product which realizes software quality requirements in a traceable way.

Instantiating HCIFrames. Figures 3 and 4 show selected subproblems that describe different aspects of the given problem situation. They already consider usability requirements. The problem frame "commanded workpiece display" in Fig. 3 which is a composite of the basic problem frames "simple workpiece" and "commanded behaviour" from Jackson is extended by HCI-related problem descriptions taken from the HCI design patterns of Tidwell [16], namely *input hints*: "place a sentence to explain what is required" (*workpiece meta information*) and *input prompt*: "prefills telling the user what to do" (*workpiece default*). The two new domains default text-message (indicating that a initial text-message should have a default value like "Hello World!") and description of text-message (requiring a label or explanation of what kind of input is expected from the user, for instance "type your chat message here:") which are related to the *workpiece* text-message. A *workpiece* is a lexical domain that can be altered.

In fact, a software development problem that fits into a problem frame containing a *workpiece* can explicitly describe quality characteristics like in this example for usability, if the *workpiece* domain is extended by *workpiece meta information* and a *workpiece default* which support self-explanatory user interfaces.

The problem frame "commanded transformation" in Fig. 4 consists of the basic frames "commanded behaviour" and "transformation". It is extended by the problem

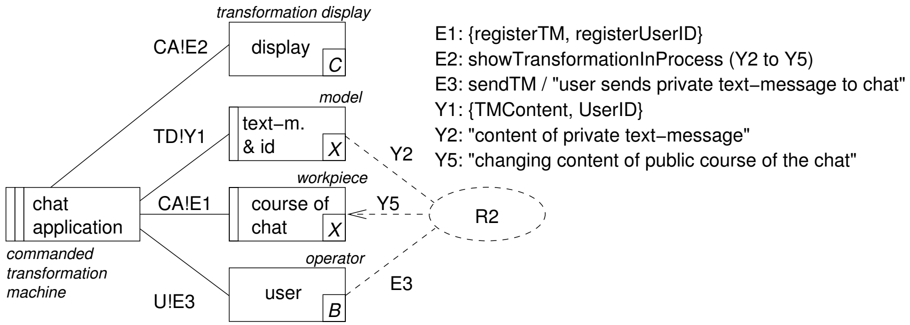


Fig. 4. Instance of the *HCI*Frame ”commanded transformation”

domain *transformation display*, which reflects the *problem description* of the HCI design pattern *progress*: ”tell user whether or not an operation is still performed and how long it will take” from van Welie [10]. The meaning of this *HCI*Frame is that whenever there is a transformation in progress (machine internal working process represented by a transformation problem frame) this is indicated to the user. For instance, in the following software design we decide to realize this transformation information by a progress bar or an information like ”sending your text-message” or ”please wait...operation in process” on a *transformation display* (display). In software analysis, we are only interested in identifying this usability requirement, whereas in software design, we will decide on its implementation. Figures 3 and 4 show that software quality aspects such as usability can be expressed with the help of *HCI*Frames. After we have introduced problem frames to express security requirements, we show how the final instantiated problem frames for our chat application example can be mapped to patterns of software architecture and design, preserving all specified quality characteristics.

Security Engineering Using Problem Frames. To meet the special demands of software development problems occurring in the area of security engineering, we are developing a catalog of security problem frames considering different security problems [8], [7]. Security problem frames consider *security requirements*. The goal is constructing a machine that fulfills the security requirements. The security problem frames strictly refer to the *problems* concerning security. They do not anticipate a solution. For example, we may require the confidential transmission of data without being obliged to mention encryption, which is a means to achieve confidentiality. *Solving* a security problem is achieved by choosing generic security mechanisms (e.g., encryption to keep data confidential). For this purpose we are developing a catalog of concretized security problem frames [8], [7]. They consider *concretized security requirements*, which take the functional aspects of a security problem into account. For each of the developed security problem frames there is at least one concretized counterpart providing a generic security mechanism. The security problem frame and its concretized counterpart used in this paper serve to treat the security requirement of *anonymity* and the concretized security requirement of *pseudonymity*, respectively.

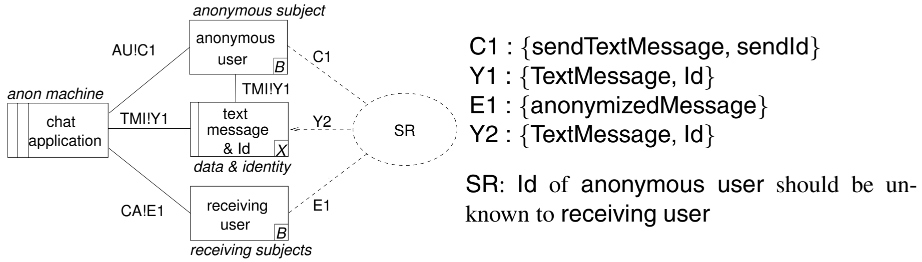


Fig. 5. GUI anonymity frame diagram

Instantiating Security Problem Frames. We now extend the requirements to be considered when developing the chat system by the security requirement *anonymity of the chat participants*. Anonymity of users and other systems is an important issue in many security-critical systems. Anonymity is the state of being not identifiable within a set of subjects [11].

Anonymity can be considered from different views, e.g., anonymity on the level of the graphical user interface (GUI), and anonymity on the level of the network. In this paper, we focus on GUI anonymity. Figure 5 depicts the security problem diagram for GUI anonymity. It is an instance of the underlying security problem frame for anonymity, which is not depicted separately in this paper.

The problem diagram in Fig. 5 contains the machine domain chat application. The lexical domain text message & Id represents the sent text message including the real id of the sender represented by the biddable domain anonymous user. The text message is received by the other chat participants represented by the biddable domain receiving user. Both, anonymous user and receiving user are specializations of the domain user (see Fig. 1). The security requirement SR states that the receiver of the text message should not know sender's real Id.

Resolve conflicting Quality Characteristics. The chat application's functional requirements and quality characteristics are now identified and described. In order to be able to derive a specification, we must ensure that the elicited requirements do not contain any conflicts. When checking for conflicts it becomes apparent that the requirement R5 (cf. 1) "Each text-message is related to its respective user so that the originator of a message can be identified." is at odds with the required SR (cf. 5) "anonymity of the chat participants". A convincing compromise to resolve this conflict can be found by negotiating both requirements. As a result, we decide to choose a generic security mechanism that uses pseudonyms [11]. With this approach, the text messages can be related to their originators (represented by pseudonyms), and at the same time the originators' identity is kept confidential.

Instantiating Concretized Security Problem Frames. In the course of transforming the security requirement for GUI anonymity into a concretized security requirement, the domain text message & nick name (see Fig. 6) is introduced.

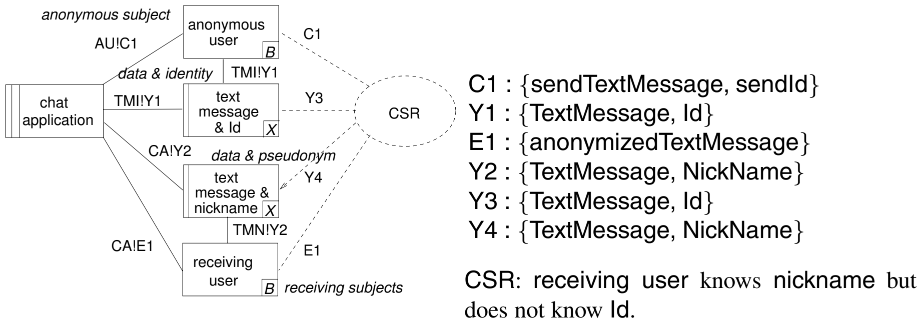


Fig. 6. Concretized GUI anonymity frame diagram

The domain part nickname represents a pseudonym. A pseudonym is a string which, to be meaningful in a certain context, is

- unique as ID
- suitable to be used to authenticate the holder and his/her "items of interest" (e.g., messages and network packets)

The holder of a pseudonym must be linked to the pseudonym itself. Then, the links must be kept confidential in order to achieve anonymity. In Fig. 6 the links are administrated by the machine domain chat application. Therefore, we may assume that chat application will keep the links confidential. The concretized security requirement CSR in Fig. 6 constrains the domain text message & nickname. The domain part nickname should be known to the receiving user, while the domain part Id should be unknown to them.

4 Role-Driven Mapping of Requirements Analysis Documents to Architectural Design Artifacts

Whatever a pattern is used for (in software analysis or design, in security or usability engineering), it generally assigns roles to entities or facets and their interaction in an abstract fashion. Patterns need to be instantiated to specify who take a certain role and to bring them into action for a concrete situation. We make use of this observation to match patterns used in requirements analysis with patterns of software architecture and design.

Starting with the problem frame instance in Fig. 2, we identify the different roles taken by the respective domains and shared phenomena. Three roles can easily identified from the problem frame diagram, namely *operator* (user), *display* (display) and *model* (list of participants & course of the chat). Now, we search for a corresponding architectural style or design pattern reflecting these roles.

Figure 7 shows the architectural style "Model-View-Controller" [2] in its upper half. The classes of this architectural style can be regarded as descriptions of the roles that objects of these classes can take. Accordingly, we map the roles *model*, *display*, *operator* of our problem frame to the classes Model, View, Controller of the architectural style.

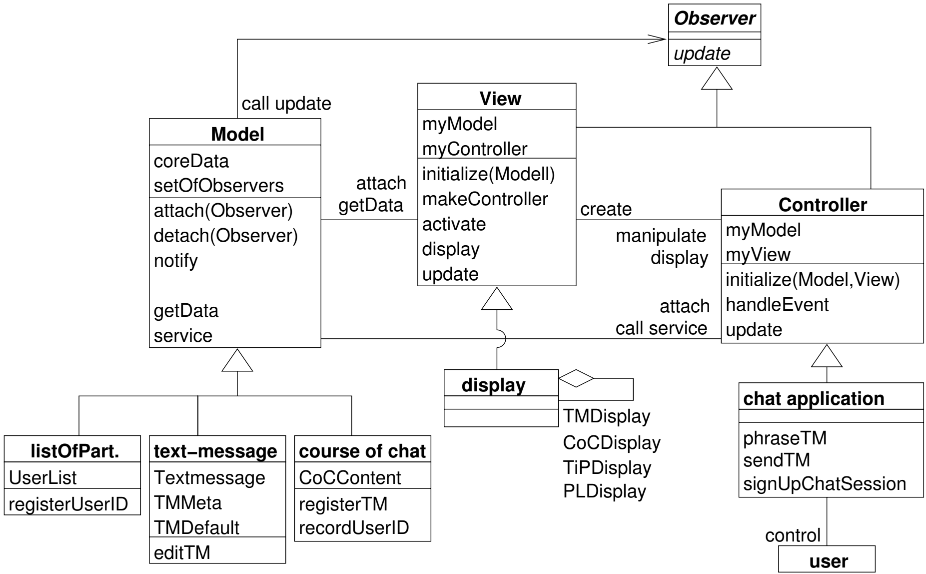


Fig. 7. Instantiated architectural style ”Model-View-Controller” for the chat application

This mapping is appropriate, because the domain roles of the problem frame and the roles of the corresponding classes in the architectural style are comparable. Since the problem frame in Fig. 2 is instantiated, we can reuse its concrete domains and shared phenomena to instantiate the chosen architectural style, too. The lower half of Fig. 7 shows how the domains and shared phenomena are mapped to the respective classes of ”Model-View-Controller” via an inheritance relation, which can be used in the UML to express assignment of roles. The chat application becomes the controller, because the machine operates according to the user commands. For the given problem situation in Fig. 2, we found one possible design which satisfies its requirements R3, R4, R7, and R10 for the chat application example.

To illustrate how quality characteristics can be preserved from requirements engineering to software design we consider the *HCI*Frame instance of ”commanded workpiece display” in Fig. 3 in detail. Similar to the previous problem frame instance of Fig. 2, it can be mapped to model-view-controller, because the role of a *workpiece* is comparable to the role of a *model*, both relying on being processed by the machine. To trace the quality characteristics stated in the *HCI*Frame instance of Fig. 3, it is necessary to consider the class text-message in Fig. 7 in more detail. The default text-message and the description of the text-message were translated from the interfaces Y8 and Y9 of the usability requirements into the interface Y2 containing TMDefault and TMMeta of the specification in Fig. 3. The latter have become attributes of the class text-message in Fig. 7. The frame instance of ”commanded transformation” in Fig. 4 requires an additional View in order to consider the *transformation (in process) display* domain display. The usability requirement which demands a *transformation display* is considered by

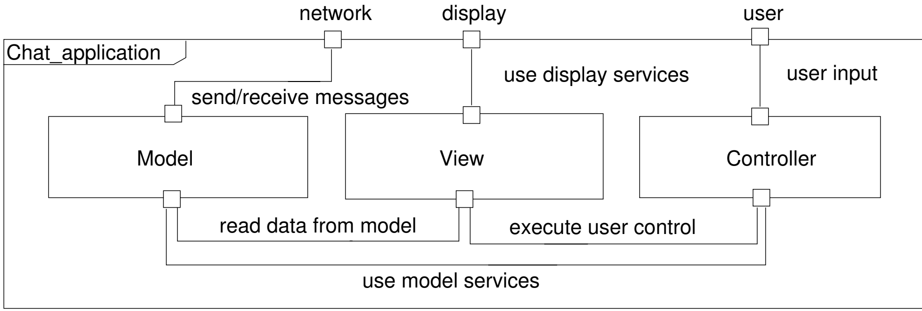


Fig. 8. Global architecture of the chat application

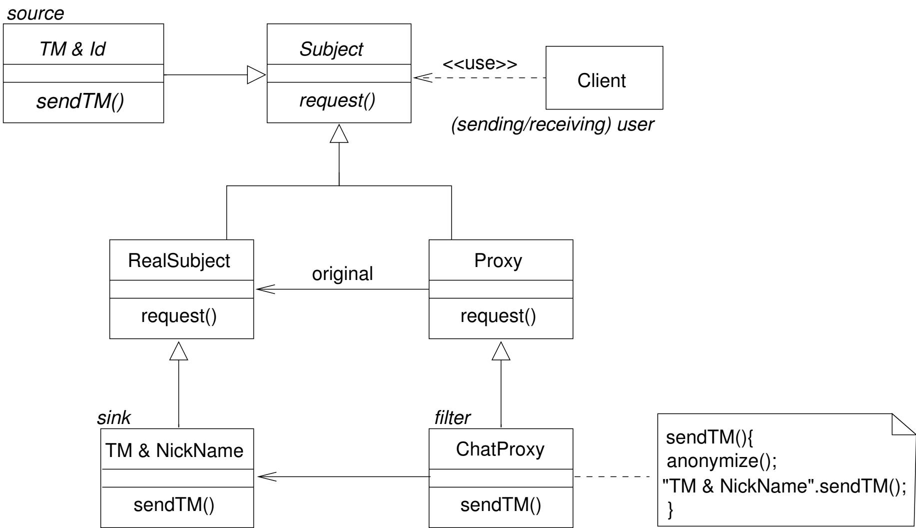


Fig. 9. Proxy design pattern/instance for GUI anonymity

the rolename TiPDisplay at the class display of Fig. 7. The role-driven mapping of the *HCI*Frame instances in Fig. 3 and Fig. 4 shows that all specified quality characteristics are preserved.

The architectural style ”Model-View-Controller” is instantiated to become the global architecture of the chat application. This architecture is depicted in Fig. 8. The ”Model-View-Controller” architecture consists of the three components Model, View, and Controller, which themself have an architecture.

Analyzing the security requirements concerning the GUI anonymity in Sect. 3.3 shows that the machine to be developed must be able to act like a *placeholder*. The text messages including the Id of the user are received by the placeholder. Then, the placeholder is responsible for exchanging the user’s Id by a pseudonym and sending the text messages and the pseudonym to the receiving chat participants. Because the behavior described above can be generally observed when applying the concretized

security problem frame for anonymity using pseudonyms, we link this frame to the design pattern *Proxy* [6] in combination with the architectural style *Pipe-and-Filter* [1].

The "Proxy" design pattern (in combination with its instance for GUI anonymity) is depicted in Fig. 9. The "Proxy" design pattern is a structural pattern that introduces a placeholder (Proxy) in order to control access to the originator *Subject*. Hence, we can map the domains of the concretized security problem diagram shown in Fig. 6 to the components of the "Proxy" pattern. The domain text message & Id is represented by the class TM & Id, the domain text message & nickname is represented by the class TM & NickName, and the machine domain chat application is represented by the class ChatProxy. The domains Anonymous user and Receiving user are represented by Client. ChatProxy receives the text messages including the chat participants's Id. Then, the ChatProxy anonymizes the received data and forwards the text message including a nickname to the actual receiving chat participants.

When anonymizing received data, the "Pipe-and-Filter" architectural style comes into play. It sees a system as a series of filters (or transformations) on input data. Data enter the system and then flow through the components one at a time until they reach some final destination. Filters are connected by pipes that transfer data. We consider the linear pipeline, in which each filter has precisely one input pipe (*source* in Fig. 9) and one output pipe (*sink* in Fig. 9). Additionally, only one filter (*filter* in Fig. 9) is needed to exchange an Id by a nickname. This functionality is reflected by the function anonymize().

Both, the "Proxy" and the "Pipe-and-Filter" architectures describe the internal architecture of the component View in Fig. 8.

Role-driven mapping enables a smooth transition of patterns used to represent the problem in software analysis to patterns used to represent a solution detailed by architectural software design. In particular, role-driven mapping serves to preserve quality characteristics.

5 Conclusion

We have shown that functional requirements as well as quality characteristics such as security and usability can be treated using our extension of Jackson's problem frames approach. We presented a software development method that preserves usability and security quality characteristics using a role-driven mapping of requirements analysis documents to architectural design artifacts.

With this approach, software engineers can hope to cover large parts of the early phases in software development using patterns.

In the future, we intend to find new patterns to extend the catalogs of *HCI*Frames, security problem frames, and concretized security problem frames. Furthermore, we intend to apply our approach to other quality characteristics such as performance and scalability.

Additionally, we plan to elaborate more on the later phases of software development. For example, we want to investigate how to integrate component technology in the development process. Finally, we plan to provide tool support for our pattern-based software development method.

Acknowledgements

The authors appreciate the in-depth comments given by the anonymous reviewers to improve this work. We would like to thank our doctoral thesis supervisor Prof. Dr. Maritta Heisel for her support.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
3. C. Choppy and M. Heisel. Une approache à base de patrons pour la spécification et le développement de systèmes d'information. *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL*, 2004.
4. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulus. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, Boston, USA, 2000.
5. Eelke Folmer and Martijn van Welie and J. Bosch. Bridging Patterns: An approach to bridge gaps between SE and HCI. *Information and Software Technology*, 48(2):69–98, 2006.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.
7. D. Hatebur, M. Heisel, and H. Schmidt. Pattern- and Component-Driven Security Engineering. Technical report, Universität Duisburg-Essen, 2006. `http://swe.uni-duisburg-essen.de/intern/seceng06.pdf`.
8. D. Hatebur, M. Heisel, and H. Schmidt. Security Engineering using Problem Frames. In G. Müller, editor, *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, LNCS 3995, pages 238–253. Springer-Verlag, 2006.
9. M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
10. M. van Welie. Patterns in Interaction Design, 2003-2006. `http://www.welie.com` Online catalogue for interaction design patterns.
11. A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology. In H. Federrath, editor, *Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2001 / 2009, pages 1–9. Springer-Verlag, 2000.
12. S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, Boston, USA, 1999.
13. T. Schümmer. *A Pattern Approach for End-User Centered Groupware Development*. PhD thesis, FernUniversität Hagen, 2005.
14. M. Shaw and S. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, Eaglewood Cliffs, New Jersey, USA, 1996.
15. I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.
16. J. Tidwell. *Designing Interfaces*. O'Reilly Media, Sebastopol, USA, 2005.
17. UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, August 2005. `http://www.uml.org`.
18. I. Wentzlaff and M. Specker. Pattern-Based Development of User-Friendly Web Applications. In *Proceedings of the 2nd International Workshop on Model-Driven Web Engineering (MDWE 2006)*, Palo Alto, USA, 2006. ACM.

Identifying "Interesting" Component Assemblies for NFRs Using Imperfect Information

Hernán Astudillo[1], Javier Pereira[2], and Claudia López[1]

[1] Universidad Técnica Federico Santa María, Departamento de Informática, Avenida España 1680, Valparaíso, Chile
{hernan, clopez}@inf.utfsm.cl

[2] Universidad Diego Portales, Escuela de Ingeniería Informática, Av. Ejército 441, Santiago, Chile
javier.pereira@udp.cl

Abstract. Component-based software elaboration becomes unwieldy for some practical situations with large numbers of components for which information is imperfect (incomplete, imprecise and/or uncertain). This article addresses the problem of identifying "interesting" component sets for some given non-functional requirements (NFRs), using imperfect information about large number of components. Rather than providing completely specified solutions, this approach allows architects to identify and compare whole assemblies, and focus eventual information-improvement efforts only on those components that are part of candidate assemblies. The proposed technique builds on the Azimut layered architectural abstractions, adapting an algorithmic approach used to mine association rules, and taking three parameters: a minimal "support score" that candidate assemblies must meet, and two credibility-value thresholds about the catalog themselves. An example illustrates the approach.

1 Introduction

Component-Based Software Development (CBD) suggests reusing existing components to build new systems, attending to benefits like shorter development times, lower costs and higher product quality. Thus, a key ingredient of CBD is components selection.

This article builds on the Azimut approach [1], which proposed progressive refinement of architectural abstractions and artifacts via architectural policies, mechanisms, components and assemblies, and on "support scores" [2] that reflect the aggregate credibility of component assemblies to satisfy specific sets of requirements. It uses an algorithmic approach to generate and compare whole component assemblies, taking imperfect information (incomplete, imprecise and/or uncertain) and identifying assemblies that are solutions and/or that may deserve a second look (and focused information gathering).

The reminder of this article is structured as follows: section 2 motivates the problem, proposes key characteristics of a solution, and examines some previous and related work; section 3 describes the Azimut architectural abstractions and a

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 204–211, 2006.
© Springer-Verlag Berlin Heidelberg 2006

measure to relate component assemblies to requirements; section 4 presents and illustrates a parametric algorithm for systematic identification of "interesting" assemblies; and section 5 presents future work and conclusions.

2 Motivation and Related Work

The construction of software systems using components offers great promise of reducing development times and costs while increasing quality, but its realization requires that architects be able to choose among alternative solutions composed from available components. A straightforward strategy could identify all possible component combinations, perhaps incorporating some technical matching restrictions, followed by their evaluation and comparison; yet this brute force approach is unfeasible due to three main issues.

1. It is sometimes quite complex to relate components (and sets thereof) to specific requirements, and specially to NFRs (non-functional requirements) due to their systemic nature.
2. In real situations, architects have at hand incomplete, imprecise and uncertain component information.
3. Resulting search spaces may be quite hard to explore systematically by humans; generation and evaluation of component-sets could be very time-consuming: with n components there are $\sum_{i=1}^{n} \binom{n}{i}$ possible component-sets.

Thus, a practical, scalable approach to component-sets identification must:

1. Relate component-sets to requirements (especially to NFRs).
2. Record imperfect information; perhaps it should even stimulate it, in the spirit of incremental gathering of information from the field.
3. Avoid the combinatorial explosion of testing each possible combination (actually, for enough components even *valid* combinations are huge in number).

We are concerned with the problem of identifying candidate component assemblies from imperfect information about huge numbers of components. In this context, "components" is to be taken as a coarse-grained COTS component.

Several techniques have been proposed for component evaluation and selection [3,4,5,6,7,8,9] that identify individual components as reuse candidates, using search criteria such as functionality, non-functional requirements (NFRs) or architectural restrictions. However, none of them allows to deal with situations where large number of components are characterized with imperfect information (i.e. imprecise, incomplete and/or unreliable).

Some MDA (Model-Driven Architecture) [10] projects, such as CoSMIC [11] and UniFrame [12], generate component-based systems, but require the use of formal component specification languages to describe available components, and from these descriptions (consistent and precise) they automate the component selection and integration process. Such approaches are appropriate when architects have good information and relatively few components at hand; when these conditions do not hold, architects should hope at least for help in identifying candidate assemblies.

3 Azimut Approach

The Azimut approach [1] supports architects in generating component assemblies for a given set of requirements by using intermediate abstract constructs as stepping stones in a derivation chain (see Fig. 1). Key constructs are:

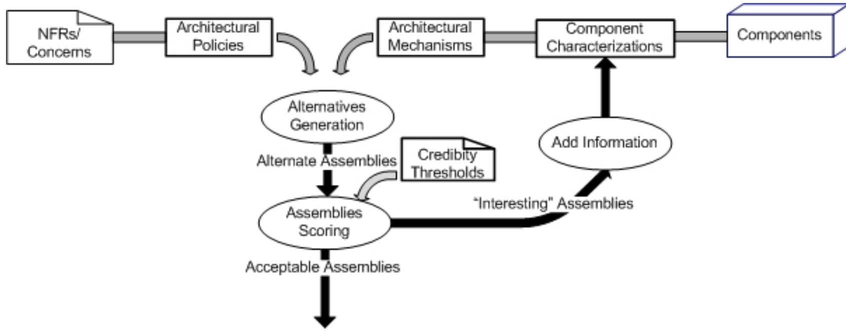


Fig. 1. Matching process

Architectural Policy: The first reification from NFRs to architectural concepts. Architectural policies can be characterized with dimensions that describe NFR-related concerns. Valid dimensions and policies are collected from globally authoritative sources of the relevant discipline (e.g. Tannenbaum [13] for replication and Britton [14] for middleware communication).

Architectural Mechanism: The constructs that satisfy architectural policies. Several mechanisms may satisfy any given architectural policy. This information may be gathered from globally or locally authoritative sources for the relevant discipline.

Component Characterization: Components are characterized by the architectural mechanisms that they implement (which implies coarse-grained components). A given component may implement several mechanisms, and several components may implement a same mechanism. Instead of a global, perfect components catalog, we propose that component characterizations be done locally at organization level, and that architects integrate local catalogs with others obtained from third-party suppliers. To deal with unreliability, credibility levels must be assigned to catalog entries (see [15] for details).

Architects identify some NFRs for a specific concern, and refine them to some architectural policies (defined by the Azimut vocabulary,), which are systematically reified to mechanisms and then components, to finally obtain a set of component assemblies (candidate solutions), among which architects may choose.

As a brief example (see [1] for details), consider communication among applications. One architectural concern is the communication type, which might have the dimensions of sessions, topology, sender, and integrity v/s timeliness

[14]; to this we add synchrony. Then, the requirement *send a private report to subscribers by Internet* might be mapped in some project (in architectural terms) as requiring communication 'asynchronous, with sessions, with 1:M topology, with a push initiator mechanism, and priorizing integrity over timeliness'. Based on these architectural requirements, an architect could search among known mechanisms for a combination that provides this specified policy; lacking additional restrictions and using well-known standards and software, a good first fit for mechanism is `SMTP` (the standard e-mail protocol). Finally, in absence of further restrictions, any available component that provides `SMTP` should be a good fit for the given requirement.

3.1 Support Scores: Modeling Imperfect Information

Consider D, P, M, C be the sets of dimensions, mechanisms and components, and n_d, n_p, n_m, n_c the respective set sizes. A policy $p \in P$ is represented by dimensions in D if there exists a set $D_p \subset D$ helping to describe it. Let $\mu(x, y) \in [0, 1]$ be defined as the *credibility level* that an abstraction x supports (satisfies, implements) an abstraction y; then, a mechanism $m \in M$ *supports* a policy p on the dimension $d \in D_p$ with credibility $\mu(m, d)$. Analogously, a component $c_i \in C \quad (i = 1, \ldots, n_c)$ is described with the set $c_i = \{m_{i,1}, m_{i,2}, \ldots, m_{i,n_m} \in M\}$ of mechanisms it implements.

Given a n-item component assembly A, the **support score of a component assembly A for policy p** is

$$S_{assem}(p, A, \alpha, \beta) = \frac{\mid C(A, D_p) \mid}{\mid D_p \mid} \tag{1}$$

S_{assem} counts dimensions in favor of the statement "assembly A satisfies policy p". $\mid C(A, D_p) \mid$ is the number of policy dimensions in p satisfied by the components in A, and $\mid D_p \mid$ is the total number of dimensions referenced by p, to be satisfied. Given that information on mechanisms and components may be unreliable [15], parameters α and β denote the minimum credibility level for a mechanism-policy or component-mechanism relation, respectively.

In normal circumstances [16], $\alpha \geq 0.5$ (or $\beta \geq 0.5$) suggests that architects need strong arguments, whilst $\alpha < 0.5$ (or $\beta < 0.5$) accepts weak arguments.

4 Generation of "Interesting" Component Assemblies

Given large number of components and imperfect information about them, a systematic way is needed to identify component assemblies that may deserve a second look (i.e. are "interesting"). We adapt an algorithm for mining association rules [17] (see Fig. 2), which incrementally generates candidate assemblies.

Let D_k, L_k be k-item sets[1]; C_k a collection of k-item assemblies; *minsup* the score threshold to be defined by the architect; and p a policy. On each

[1] A k-item set is a set containing k elements.

iteration k, the function *candidates*(.) lexicographically orders and returns the $(k+1)$-item assemblies based on k-item unsatisfactory assemblies in L_k, minimizing redundancy because it does not further consider any satisfactory assemblies.

```
begin
  D_0 = {c ∈ A | support(c) >= minsup};
  L_1 = A \ D_0;
  for (k = 1; L_k ≠ ∅; k++) do
      begin
        C_k = candidates(L_k);
        for c ∈ C_k do support(c) = S_assem(p, c, α, β); od
        L_{k+1} = {c ∈ C_k | support(c) < minsup};
        D_k = D_{k-1} ∪ (C_k \ L_{k+1});
      end od
end
funct candidates(F_k);
  begin
    C = ∅;
    for (f_1, f_2 ∈ F_k where f_1 = {c_1, ..., c_{k-1}, c_k} ∧ f_2 = {c_1, ..., c_{k-1}, c'_k} ∧ c_k < c'_k) do
        begin
          f = {c_1, ..., c_{k-1}, c_k, c'_k};
          if ∀c ∈ f : f − {c} ∈ F_k then C = C ∪ {f}; fi
        end
    .C;
  end
```

Fig. 2. Algorithm to generate potential component assemblies

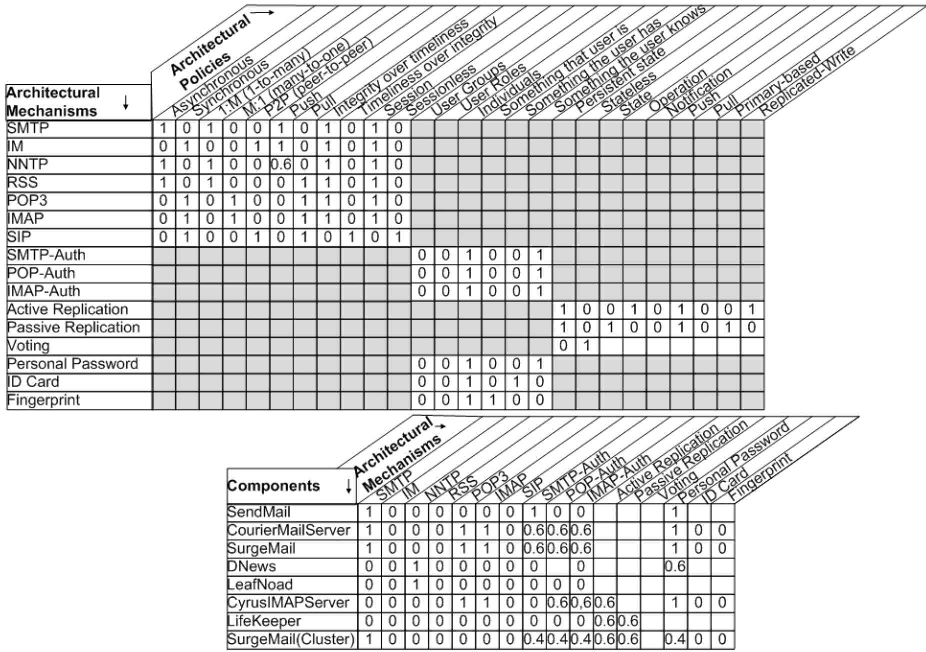
Example of assemblies generation. Consider how candidate assemblies are generated for different parameter values of α, β and *minsup*. Mechanism and component information may be recorded in catalogs such as partially shown in figure 3. Take a required architectural policy with *p communication*: (e_1: asynchronous, e_2: 1:M topology, e_3: push receiver, e_4: integrity over timeliness); *security*: (e_5: individual authorization; e_6: authentication based on something the user knows); *availability*: (e_7: persistent state replication, and e_8: replicated-write consistency).

The components shown in Figure 3 are labeled in Table 1 as: c_1 = SendMail, c_2 = Courier Mail Server, c_3 = Surge Mail, c_4 = DNews, c_5 = Leaf Noad, c_6 = Cyrus IMAP Server, c_7 = LifeKeeper, c_8 = SurgeMail (Cluster).

In Table 1, supports for 1, 2 an 3-itemset assemblies are presented, when $\alpha = 0.5$ and $\beta = 0.6$. The in-parenthesis numbers represent the component-mechanism credibilities. In this case, $minsup = 0.75$.

The set of solutions is reduced from $\sum_{i=1}^{8} \binom{8}{i} = 255$ to 20 combinations to analyze. There are only 7 potential assemblies proposed to the architect, satisfying $minsup = 6/8$. These solutions may be reserved by the architect for any subsequent analysis.

Examination of "interesting" component assemblies. An examination of Table 1 reveals that several assemblies (with 1, 2 and 3 components) have a high



Architectural Mechanisms ↓ / Architectural Policies →	Asynchronous	Synchronous	1:M (1-to-many)	M:1 (many-to-one)	P2P (peer-to-peer)	Push	Pull	Integrity over timeliness	Timeliness over integrity	Session	Sessionless	User Groups	User Roles	Individuals	Something that user is	Something the user has	Something the user knows	Persistent state	Stateless	State	Operation	Notification	Push	Pull	Primary-based	Replicated-Write
SMTP	1	0	1	0	0	1	0	1	0	1	0															
IM	0	1	0	0	1	1	0	1	0	1	0															
NNTP	1	0	1	0	0	0.6	0	1	0	1	0															
RSS	1	0	1	0	0	0	1	1	0	1	0															
POP3	0	1	0	1	0	0	1	1	0	1	0															
IMAP	0	1	0	1	0	0	1	1	0	1	0															
SIP	0	1	0	0	1	0	1	0	1	0	1															
SMTP-Auth												0	0	1	0	0	1									
POP-Auth												0	0	1	0	0	1									
IMAP-Auth												0	0	1	0	0	1									
Active Replication																		1	0	0	1	0	1	0	0	1
Passive Replication																		1	0	1	0	0	1	0	1	0
Voting																		0	1							
Personal Password												0	0	1	0	0	1									
ID Card												0	0	1	0	1	0									
Fingerprint												0	0	1	1	0	0									

Components ↓ / Architectural Mechanisms →	SMTP	IM	NNTP	RSS	POP3	IMAP	SIP	SMTP-Auth	POP-Auth	IMAP-Auth	Active Replication	Passive Replication	Voting	Personal Password	ID Card	Fingerprint
SendMail	1	0	0	0	0	0	0	1	0	0				1		
CourierMailServer	1	0	0	0	1	1	0	0.6	0.6	0.6				1	0	0
SurgeMail	1	0	0	0	1	1	0	0.6	0.6	0.6				1	0	0
DNews	0	0	1	0	0	0	0	0		0				0.6		
LeafNoad	0	0	1	0	0	0	0	0	0	0						
CyrusIMAPServer	0	0	0	0	1	1	0	0	0.6	0,6	0.6			1	0	0
LifeKeeper	0	0	0	0	0	0	0	0	0	0	0.6	0.6				
SurgeMail(Cluster)	1	0	0	0	0	0	0	0.4	0.4	0.4	0.6	0.6		0.4	0	0

Fig. 3. Partial content of mechanisms and components catalogs

Table 1. Support of 1-, 2- and 3-item component assemblies ($\alpha = 0.5$; $\beta = 0.6$)

policy $p >$	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	Support
c_1	1	1	1	1	1	1			6/8
c_2	1	1	1	1	1(0.6)	1(0.6)			6/8
c_3	1	1	1	1	1(0.6)	1(0.6)			6/8
c_4	1	1	1	1					4/8
c_5	1	1	1	1					4/8
c_6				1	1(0.6)	1(0.6)			3/8
c_7							1(0.6)	1(0.6)	2/8
c_8							1(0.6)		1/8
(c_4, c_5)	1	1	1	1					4/8
(c_4, c_6)	1	1	1	1	1(0.6)	1(0.6)			6/8
(c_4, c_7)	1	1	1	1			1(0.6)	1(0.6)	6/8
(c_4, c_8)	1	1	1	1			1(0.6)		5/8
(c_5, c_6)	1	1	1	1	1(0.6)	1(0.6)			6/8
(c_5, c_7)	1	1	1	1			1(0.6)	1(0.6)	6/8
(c_5, c_8)	1	1	1	1			1(0.6)		5/8
(c_6, c_7)					1(0.6)	1(0.6)	1(0.6)	1(0.6)	4/8
(c_6, c_8)					1(0.6)	1(0.6)	1(0.6)		3/8
(c_7, c_8)							1(0.6)	1(0.6)	2/8
(c_4, c_5, c_8)	1	1	1	1			1(0.6)		5/8
(c_6, c_7, c_8)				1	1(0.6)	1(0.6)	1(0.6)	1(0.6)	5/8

support score. Even in absence of detailed technical knowledge about compatibility and collaboration among each assembly's components, already some good candidates have been identified, and can be a starting point for the adopted solution.

An inter-assembly analysis indicates that components e_5, e_6, e_7 and e_8 appear in several highly-scored assemblies, although information about them is not top quality. An architecture team that wished to reduce the risk in assembly selection would focus its (always scarce) resources into improving these components' information Another team could privilege only e_5 and e_6 because they are part of smaller candidate assemblies, if such assemblies are better liked (e.g. due to some cost or complexity measure).

5 Conclusions

This article has presented an approach to component-sets identification. It is based on the Azimut framework [1], which support systematic derivation of component-sets from NFRs via architectural policies and mechanisms; a scoring process borrowed from decision-aid for preference-establishment based on voting [18]; and a generation algorithm to identify component assemblies that are potential solutions and/or deserve a closer look (and possibly improved information gathering). The described process to generate and evaluate potential assemblies allows architects to engage in iterative exploration of design spaces.

Three main lines of further work are ongoing. First, the model currently only records the credibility of a relation between a component and a mechanism, but not its strength (i.e. how well one supports the other); current work will extend modeling to both strength and credibility of relations. Second, given that the robustness of solutions strongly depends on the credibility thresholds, the proposed approach encourages architects to use aggregate credibility information to focus information improvement on interesting assemblies; unfortunately, a detailed discussion of robustness is beyond the scope of the present paper.

Finally, generated assemblies must be checked by architects for technical feasibility of integration, as well as for business-related properties (e.g. global cost, or development complexity). Domain-specific catalogs are being gathered from industry sources.

References

1. López, C., Astudillo, H.: Explicit architectural policies to satisfy NFRs using COTS. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference. Volume 3844., Lecture Notes in Computer Science, Springer (2006) 227 – 236
2. Astudillo, H., Pereira, J., López, C.: Evaluating alternative COTS assemblies from unreliable information. In: QoSA'06: 3rd International Workshop on Quality of Software Architecture, Lecture Notes in Computer Science, Springer (2006) to appear.

3. Ncube, C., Maiden, N.: PORE: Procurement-oriented requirements engineering method for the CBSE development paradigm. In: International Workshop on Component-based Software Engineering. (1999)
4. Ochs, M., Pfahl, D., Chrobok-Diening, G., Nothhelfer-Kolb, B.: A COTS acquisition process: Definition and application experience. In: ESCOM'00: 11th European Software Control and Metric Conference. (2000)
5. Phillips, B.C., Polen, S.M.: Add decision analysis to your COTS selection process. The Journal of Defense Software Engineering, Software Technology Support Center Crosstalk (2002)
6. Kontio, J.: A case study in applying a systematic method for COTS selection. In: ICSE'96: Proceedings of the 18th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (1996) 201–209
7. Douglas, K., Laurence, B.: Applying social-technical approach for COTS selection. In: Proceedings of 4th UKAIS Conference, University of York, McGraw Hill (1999)
8. Alves, C., Castro, J.: CRE: A systematic method for COTS components selection. In: SBES 2001: 15th Brazilian Symposium on Software Engineering. (2001)
9. Chung, L., Cooper, K.: COTS-aware requirements engineering and software architecting. Software Engineering Research and Practice (2004) 57–63
10. Group, O.M.: MDA Guide Version 1.0.1. Object Management Group (OMG). (2003)
11. Gokhale, A., Balasubramanian, K., Lu, T.: CoSMIC: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In: OOPSLA'04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, New York, NY, USA, ACM Press (2004) 218–219
12. Cao, F., Bryant, B.R., Burt, C.C., Raje, R.R., Olson, A.M., Augustona, M.: A component assembly approach based on aspect-oriented generative domain modeling. In: Electronic Notes in Theoretical Computer Science, SC 2004: Procs. of the Software Composition Workshop. Volume 114., Science Direct (2005) 119–136
13. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice Hall (2002)
14. Britton, C., Bye, P.: IT Architectures and Middleware: Strategies for Building Large, Integrated Systems. 2nd edn. Addison-Wesley Professional (2004)
15. López, C., Astudillo, H.: Multidimensional catalogs for systematic exploration of component-based design spaces. In: IWASE'06: 1st International Workshop on Advanced Software Engineering. Proceedings of IFIP World Congress 2006. (2006)
16. Pomerol, J.C., Barba-Romero, S.: Multicriterion Decision in Management: Principles and Practice. Kluwer Academic Publishers (2000)
17. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. Morgan Kaufmann, San Francisco, CA, USA (1998) 580–592
18. Roy, B.: Multicriteria Methodology for Decision Aiding. Kluwer (1996)

Towards More Flexible Architecture Description Languages for Industrial Applications

Rabih Bashroush, Ivor Spence, Peter Kilpatrick, and John Brown

School of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast, N. Ireland, UK
{r.bashroush, i.spence, p.kilpatrick, tj.brown}@qub.ac.uk

Abstract. Architecture Description Languages (ADLs) have emerged in recent years as a tool for providing high-level descriptions of software systems in terms of their architectural elements and the relationships among them. Most of the current ADLs exhibit limitations which prevent their widespread use in industrial applications. In this paper, we discuss these limitations and introduce ALI, an ADL that has been developed to address such limitations. The ALI language provides a rich and flexible syntax for describing component interfaces, architectural patterns, and meta-information. Multiple graphical architectural views can then be derived from ALI's textual notation.

Keywords: Software Architecture, Architecture Description Languages, Architectural Patterns.

1 Introduction

Architecture Description Languages (ADLs) have emerged as viable tools for formally representing the architectures of systems at a reasonably high level of abstraction to enable better intellectual control over the systems [1]. ADLs usually help in architectural analysis with issues such as consistency, modifiability, performance, etc. However, there is no general agreement on what ADLs are expected to capture/represent about an architecture (behavior, structure, interfaces, etc.). Most work on ADLs today has been undertaken with academic rather than commercial goals in mind and they tend to be very vertically optimized towards a particular kind of analysis [2].

The ADL community generally agrees that a Software Architecture is a set of components and the connections among them conforming to a set of constraints. Component interfaces usually comprise a set of provided and required services (a service could be a function call, a message type, etc.).

Although some ADLs have been put to industrial use [3], the majority of ADLs have not scaled up well, and their use remains confined to small-scale case studies.

In this paper we discuss a number of limitations evident in most current ADLs which might have constrained their use to small-scale academic applications. We then present the major concepts behind the ALI ADL which has been designed with the identified limitations in mind. ALI also built upon our experience with the ADLARS

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 212 – 219, 2006.
© Springer-Verlag Berlin Heidelberg 2006

[4] ADL and adopted much of the solution space provided by ADLARS such as its support for Software Product Lines.

In the following, we begin in Section 2 by discussing the limitations within current ADLs. Section 3 then highlights the rationale behind the ALI language. Finally, discussion and future work is presented in Section 4.

2 Limitations Within Existing ADLs

In this section we discuss the potential limitations identified by examining a number of existing and mature ADLs selected from across the literature to reflect the state-of-the-art in the domain. Among these ADLs are: ACME [7], Koala [3], Rapide [8], and Wright [9].

It is worth mentioning here that the Unified Modeling Language (UML) [5], even though it is used within different stages of the development process (and without doubt a *de facto* modeling language), is not considered a strong candidate as an ADL due to many issues including it being a pure graphical notation and the fact that it does not treat connectors as first class citizens (even though UML 2.0 [6] took one step further in the ADLs' direction with the introduction of ports and interfaces). Furthermore, UML initially was geared more towards code description rather than architecture description.

We have examined and experimented with these ADLs to identify the novelty and the strengths of each. We have also identified a number of shared limitations, particularly in the context of real-life applications. These are summarized below.

2.1 ADLs Are Over-Constraining

Current ADLs force architects to use specific styles/interface types throughout their architecture by providing a single component interface type model. For example, while interfaces are described in terms of input and output ports in Wright, interfaces are described in terms of services provided/required in Koala, and messages sent/received in ADLARS [4]. With current advances in different domains including Service Oriented Architectures (SOA) and adaptive systems, within a single system we could have a number of different interface types used (which is often the case). Capturing such architectures with most current ADLs entails abstracting a number of interface types to the single interface type supported by the ADL. This could be problematic especially when the interface types form a crucial part of the architecture description (e.g. in SOAs). Also, by requiring that components have specific types of interfaces (hardware-like input/output ports, e.g. ACME; message based communication, e.g. ADLARS; etc.), ADLs may be indirectly enforcing the style of communication to be used in the system on the architect.

2.2 ADLs Provide a Single View of the System

It has become widely recognized in the software architecture community that software architectures contain too much information to be adequately captured and displayed in one view. Multiple views are needed to describe an architecture where each view can encompass a set of related concerns. This has been recognised in a number of

industrial approaches [10, 11] and standards [ANSI/IEEE 1471-2000] (while others went one step further to consider also perspectives [12]). When this is the trend in industry, there is no reason why ADLs should not support multiple views. The reason why most ADLs are restricted to one view of a system may be attributed to the fact that ADLs inherently focus on the structural aspects of the architecture which has traditionally been the central issue. Hence, ADLs provide only the structural view of the system. Today's concerns have gone beyond purely structural factors, and issues such as quality attributes, design decisions, etc. are now considered an intrinsic part of architecture description [13].

2.3 ADLs Lack Proper CASE Tool Support

CASE tool support availability varies from one ADL to another. Some ADLs have parser/syntax validation tool support, others have basic simulation tools, while others have no tool support at all. For an industrial buy-in, tool support is a major selling point for any ADL due to the size and complexity involved in real-life systems. Even for those ADLs with tool support, most of the tools developed do not scale up to work with large system descriptions (e.g. hundreds of components and connectors). While some simulators are unable to cope with systems comprising over 100 components, most graphical tools have no mechanism to properly display systems with 30-40 components or more. This problem, however, differs from the previous two in the sense that for a commercial level tool support to be developed for an ADL, the ADL should be adopted by a tool vendor. For a tool vendor to adopt an ADL, the ADL should demonstrate a commercial potential (which is best done using proper tools!). A potential solution to this problem would be to make use of existing tool support for other notations such as UML in the first stage. This could perhaps be done by transforming back and forth between the ADL notation and UML (e.g. using meta ADLs like in [14]).

In the following section, we will introduce the rationale behind the ALI language which was designed with the aforementioned limitations in mind.

3 ALI Rationale

ALI has been designed on the basis of our previous work on ADLs, including the ADLARS notation [4]. It seeks to address a number of the issues discussed above.

While adopting successful concepts from ADLARS, such as the relationship between the feature model [15] and the architectural structure [16], ALI introduces, among other things, a high level of flexibility for interface description. Major concepts behind the ALI ADL are discussed in this section.

3.1 Flexible Interface Description

Revisiting the first limitation discussed in the previous section, current ADLs allow only for fixed interface types. Providing a specific interface type restricts the usage of an ADL to domains where most components would only have that particular type of interface. This is in addition to restricting the architect to use a specific style of communication among components (e.g. message-based, method invocation, hardware-like ports, etc.).

The ALI ADL attempts to address this limitation by providing no pre-defined interface types. Instead, ALI introduces a sub-language (which is a sub-set of the JavaCC [17] notation) that gives users the flexibility to define their own interface types.

For example, consider a simple web service having a WSDL (Web Services Description Language) interface and containing a number of components which are described with input/output ports as interfaces. Assume also, that each component contains a number of objects/classes that have interfaces defined in terms of functions provided/required (summarized in Fig. 1). This is a fairly standard level of nesting/ abstraction within today's service oriented architectures.

If we were to model this using any of the existing ADLs, we would have to abstract the different interface types with the single interface type supported by the ADL used. By doing so, we would be unnecessarily abstracting away useful and important architectural information - especially in domains such as SOA where interface descriptions/types are of important architectural value.

It would also be difficult to identify a comprehensive set of interface types beforehand to be provided by an ADL due to the large number of interface types that already exist in the literature. In addition, new interface types emerge with the advancement of different technologies (e.g. GWSDL emerging from the work on grid computing, etc.). So, an ADL may benefit from a flexible mechanism that allows the architect to define his/her own interface types along with the binding constraints. This is the model that is adopted by ALI.

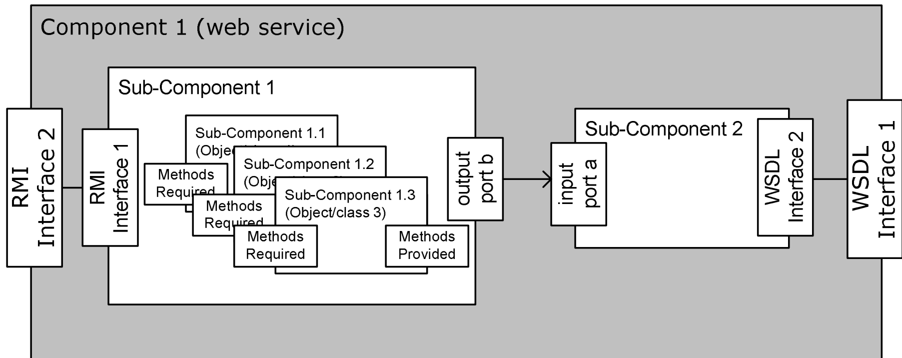


Fig. 1. An example architecture of a simple web service

3.2 Architectural Pattern Description

Architectural patterns (or architectural styles) express a fundamental structural organization or schema for software systems and sub-systems. As these patterns are often reused within the same system (and sub-systems) or across multiple systems, providing syntax for capturing/describing these patterns to enable better pattern reuse is important. This is another major aspect of the ALI notation. ALI envisages architectural patterns as the architectural level equivalent of functions (methods) in programming languages.

Within ALI, patterns are defined and reused as functions. *Pattern templates* are first defined by specifying the way components are connected to form the architectural pattern. Then, these pattern templates are instantiated throughout the architecture definition to connect sets of components (whose interfaces are passed as arguments to the pattern template) according to the pattern template definition (e.g. Fig 2).

As shown in Fig. 2, simple architectures can be constructed through the usage of a number of patterns.

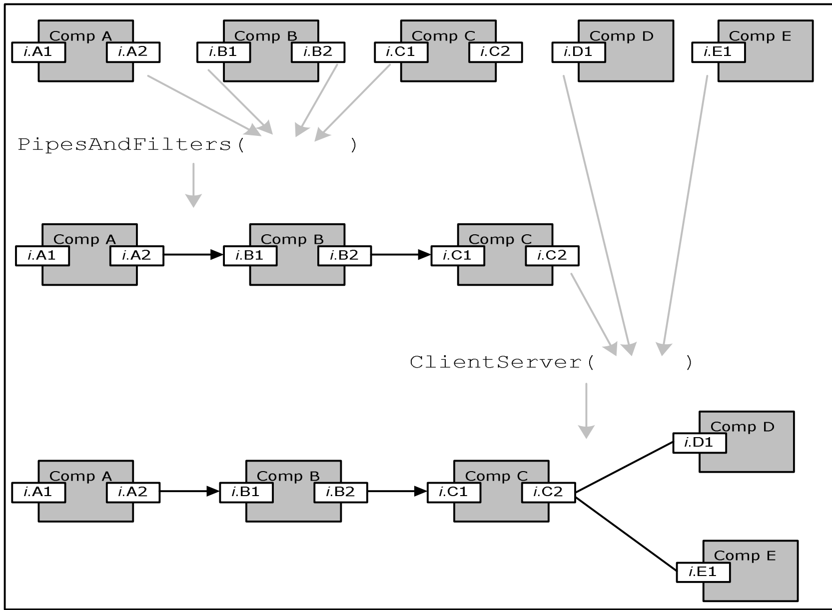


Fig. 2. A simple architecture assembled from a number of components using two pattern templates: *PipesAndFilters* and *ClientServer*

3.3 Formal Syntax for Capturing Meta-information

As discussed in section 2, there is more to architecture than the structural aspects of the system. Issues such as component implementation cost/benefit, design decisions, versions, quality attributes, etc. have not been the focus of most existing ADLs. ADLs such as ADLARS [4] and few others allow the addition of free textual comments to the architecture description using standard commenting syntax similar to that used in programming languages (e.g. through the usage of "/*", "//", etc.). This, however, proves to be problematic if CASE tools are to be used to analyze or produce useful documentation from the free textual comments.

One of the challenges with formalizing the syntax for capturing the meta-information is in deciding on the information to be captured in the architecture description. Although there is some information that is usually captured in most architecture documentations (e.g. design decisions, quality attributes, etc.) some other

information may vary from one domain to the other and from one enterprise to another (depending on the nature of the domain, the structure of the enterprise, etc.).

In ALI, a special syntax has been introduced to allow for creating *meta types*. Different meta types can be created within a system to act as packages of information (quality attributes, versions, design decisions) which could be attached to different architectural structures throughout the system description.

3.4 Linking the Feature and Architecture Spaces

As Feature Models [18] are built to capture end-users' and stake-holders' concerns and architectures are designed from technical and business perspectives, a gap exists between the two spaces. This gap introduces a number of challenges including: feature (requirements) traceability into the architecture; the ability to verify variability implementation (in Software Product Lines), etc.

ALI attempts at bridging this gap by allowing the architect to link directly the architectural structures to the feature model. Within ALI, it is possible to relate components, connectors, patterns etc. in an architecture description to features in the feature model using first order logic. This permits the capture of complex relationships that might arise between the two spaces in real-life systems.

ALI has adopted and enhanced this concept from ADLARS [4] which was the first ADL to introduce support for linking the feature space to architectural components.

4 Discussion and Future Work

In this paper we have discussed the main issues that might be restricting most current ADLs to small-scale case-studies rather than real-life industrial applications. Restrictive syntax/structure, lack of tool support, and single view presentation are among the limitations discussed.

ALI was created with these limitations in mind and was designed to provide a blend between flexibility and formalism. While flexibility gives freedom for the architect during the design process, formalism allows for architecture analysis and potential automation using proper CASE tool support (e.g. on-the-fly architecture documentation, code generation, etc.).

This paper has focused on the concepts behind ALI. Further information about the ALI notation can be found in [19].

ALI adopts a flexible model for its graphical notation. The textual notation serves as a central database of the architecture description. CASE tools use this information as the source to derive the different relevant architectural views (which can be customized using CASE tools). This model will help alleviate the problem of mismatches among multiple views of the system when maintained separately.

As different architects in different domains (e.g. IS, Telecom, Grid, etc.) would be more comfortable drawing or representing architectures using their own set of symbols/figures (e.g. a cylinder to show a database rather than the standard box of ADLs, etc.), ALI allows for replacing boxes in the graphical notation with any figure the architect chooses as long as interfaces are displayed and labeled properly on that figure. As a comparison between the two approaches (boxes vs figures to represent components), the problem with boxes is that all boxes look basically alike, so it would

be relatively difficult to identify and locate a component in a large architecture. On the other hand, the problem with having different images for different components is that, with a large number of component types, the architecture may appear unduly cluttered. So, whether to use boxes or images is left to the architect to decide upon based on the nature and size of the system in any particular project.

As for future work, two major issues top the list for the work on the ALI project:

- *Tool support*: while the work on a toolset for ALI is in progress (using the ADLARS toolset as a starting point), the plan is to make the ALI toolset (and the notation) an open source project. In this way the notation and the toolset will, it is hoped, benefit from a broad range of contributions, both from industry and academia.
- *Providing "round-trip" to code*: the ability to go from architecture to code and back has always been an appealing concept for people working in industry. Work on Model Driven Architectures (MDA) is one successful example of communities working on code generation from architecture specification. In ALI, the possibility of attaching code to components (and glue code to connectors) will be studied. This, if found feasible, will potentially allow for automated generation of substantial parts of the system implementation.

Finally, as the major idea behind ALI is to bridge the gap between industry and academia in the field of ADLs, devising a proper "roll-out" plan for the adoption of ALI in industrial pilot projects (in the first instance) will be considered. Once experience is gained with the language in industrial settings, the aim is to have libraries of meta types, interface types, connector types, etc. for each application domain which architects could then use off-the-shelf.

Acknowledgments. We would like to thank Felix Bachmann, Senior Member of Technical Staff at the Software Engineering Institute, Carnegie Mellon University as well as the Product Line Practice PLP group for their valuable feedback during the initial stage of this work. We would also like to thank David Garlan, Professor, School of Computer Science, Carnegie Mellon University, for his constructive comments on the notation when it started taking shape a few years ago.

References

1. P. Clements, R. Kazman, and M. Klein, Evaluating Software Architecture: Methods and Case Studies. 2002: SEI series in software engineering. Addison-Wesley.
2. E. Woods and R. Hilliard, WICSA 5 Working Group Report "Architecture Description Languages in Practice". November 2005.
3. R.van Ommering, F. van der Linden, J. Kramer, and J. Magee, The Koala Component Model for Consumer Electronics Software. IEEE Computer, March 2000: p. 78-85.
4. R. Bashroush, T.J. Brown, I. Spence, and P. Kilpatrick. ADLARS: An Architecture Description Language for Software Product Lines. Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop. April 2005. Greenbelt, Maryland, USA.
5. G. Booch, I. Jacobson, and J. Rumbaugh, The Unified Modeling Language User Guide. 1998: Addison-Wesley.

6. OMG, UML 2.0 Specification. October 2004, http://www.uml.org.
7. D. Garlan, R. Monroe, and D. Wile, Acme: Architectural Description of Component-Based Systems, in Foundations of Component-Based Systems, G.T. Leavens and M. Sitaraman, Editors. 2000, Cambridge University Press. p. 47-68.
8. D.C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Proceedings of DIMACS Partial Order Methods Workshop IV. July 1996. Princeton University.
9. R. Allen, A Formal Approach to Software Architecture. 1997, CMU: PhD Thesis.
10. C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture. 2000, Boston: Addison-Wesley.
11. P. Kruchten, Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software, November 1995. 12 (6): p. 42-50.
12. N. Rozanski and E. Woods, Software Systems Architecture. 2005: Addison Wesley.
13. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures. SEI Series on Software Engineering. 2002: Addison Wiesley Longman.
14. Smeda, M. Oussalah, and T. Khammaci. Mapping ADLs into UML 2 Using a Meta ADL. Proceedings of The 5th IEEE/IFIP Working International Conference on Software Architecture. November 2005. Pittsburgh, USA.
15. T.J. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan. Weaving Behavior into Feature Models for Embedded System Families. Proceedings of the 10th International Software Product Line Conference SPLC 2006 [to appear]. August 2006. Baltimore, Maryland, USA.
16. T.J. Brown, R. Bashroush, C. Gillan, I. Spence, and P. Kilpatrick. Feature Guided Architecture Development for Embedded System Families. Proceedings of the 5th IEEE Working Conference on Software Architecture WICSA-5. November 2005. Pittsburgh, PA, USA.
17. The Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. https://javacc.dev.java.net/.
18. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Patterson, Feature Oriented Domain Analysis (FODA) feasibility study. 1990, Software Engineering Institute, Carnegie Mellon University.
19. R. Bashroush, T.J. Brown, I. Spence, and P. Kilpatrick. Flexible Component-Based Architecture Description using ALI. Submitted to the 13th IEEE Asia Pacific Software Engineering Conference (APSEC06). December 2006. Bangalore, India.

Architecture Transformation and Refinement for Model-Driven Adaptability Management: Application to QoS Provisioning in Group Communication

Christophe Chassot[1,2], Karim Guennoun[1], Khalil Drira[1],
François Armando[1,⋆], Ernesto Exposito[1,2], and André Lozes[1,2]

[1] LAAS-CNRS Toulouse - France
[2] University of Toulouse / INSA / IUT Toulouse - France

Abstract. In this paper, we identify and define the architectural properties of the different levels of abstraction necessary for adaptability management. The distinguished levels allow describing service-level, component-level and process-level architectural properties. Using graph grammars and graph transformation, we enforce the conventional graph-based representation of system and software architectures. We go beyond past informal studies by providing formal rules for architecture refinement and transformation. We focus on applications where communication is used to support the cooperation between distributed group members. We consider a concrete case study of Military Emergency Operation (MEO). Operation management.

Keywords: Graph grammars, self-adaptability, model-oriented automated management, service-oriented dynamic architecture, QoS, cooperative and mobile applications.

1 Introduction

Adaptability management still remains an unsolved problem. Several issues need to be tackled such as the discovery of objects, components and services. Evaluation of resources availability for network and machines as well as evaluation of the current adaptation policy has also to be defined. New parameters and services have also to be considered. Adaptability is required at several levels simultaneously. For instance, for QoS adaptability in group communication-based activities, both changing communication and computation resources and evolving group structures have to be managed. However, managing adaptation at various levels requires coordination without which it can lead to performances way below the targeted ones. For example, having to react to network congestion, an adaptation of the sending rate both at the Application layer (e.g.

⋆ This author is supported by the DGA (Délégation Générale de l'Armement). This work is also partially supported by the IST NetQoS project.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 220–227, 2006.
© Springer-Verlag Berlin Heidelberg 2006

by reduction of images size) and at the Transport layer (e.g. by a rate control) could result in over-reaction and as such a non-optimal solution. Different abstraction levels of adaptability have to be distinguished, and adaptation has to be managed in a coordinated manner both within and between these abstraction levels [1,2]. In this way, [1] proposes an approach for identifying the adaptation level (in this case, OSI layers) to be considered depending on the network predictability. However, this work does not consider important aspects associated with group activities. For group communication-based cooperative activities, different abstraction levels have to be addressed in order to provide group-aware QoS management policies. Adaptation at the highest levels should be guided by the evolving of the activity requirements. Adaptation at the lowest levels should be driven by the changes due to device/network constraints.

In this paper, we identify and define the architectural properties of the different levels of abstraction necessary for adaptability management. The distinguished levels allow describing service-level, and component/process-level architectural properties. Using graph grammars and graph transformation, we enforce the conventional graph-based representation of system and software architectures. We go beyond past informal studies by providing formal rules for architecture refinement and transformation. We focus on applications where communication is used to support the cooperation between distributed group members. We consider a concrete case study of Military Emergency Operation (MEO) management involving several participants with different roles carrying machines with different capacities for communication and computation. The activity is organized around two main steps: the exploration step and the acting step. For each of these steps, we have elaborated the grammars that define all the possible correct architectures that may implement communication in conformance with the cooperation requirements. Switching between these architectures constitutes an architectural adaptation action that may solve lower level constraints ! while remaining in conformance with the upper level requirements. We also implemented graph transformation rules that allow architecture to be adapted when moving from one step to another in both directions. We defined the graph grammars that allow implementing the mapping of a given level architecture onto the underlying level. This paper is organized as follows. The related work is described in section 2. Section 3 presents a case study to which the introduced modeling approach is applied. The graph-based description and the transformation grammars and rules are presented in section 4. Conclusions and future work are finally presented in section 5.

2 Related Work for Adaptation in Distributed and Communicating Systems

Adaptation objectives, actions and properties are among the main facets of adaptability. They are studied and classified in this section.

2.1 Adaptability Objectives

Several objectives are targeted by dynamic provisioning and adaptability. QoS such as connectivity or access bandwidth when roaming from a wireless network to another (i.e. handover management) is very considered [3,4]. End to end QoS optimization is also very addressed, for instance in the Best Effort Internet [5,7]. Security, such as data confidentiality, firewalls activation, bypass and deactivation, is more and more considered, particularly when wireless networks are involved [8]. Resource optimization is also addressed. They may be related to devices energy, computation or storage capability [9]. Cooperation aspects have to be considered in group activities. For instance, this can deal with management of user input/output for a distributed game exercise.

2.2 Adaptation Actions and Scopes

Several adaptation solutions have been developed in the last decade. Their scope covers both high and low layers of the OSI model. Application layer. At the Application level, [5] addresses the need for adaptation in video streaming applications distributed over the Best-Effort Internet. Several techniques have been proposed based on two mechanisms: an applicative congestion control, which can be implemented in several ways: rate control, rate-adaptive video encoding, rate shaping; and error control integrating concepts such as delay-constrained retransmissions and forward error correction. At Middleware and Transport levels, TCP's reaction to network congestion is a well-known adaptation example; however, it does not handle applicative QoS requirements such as transit delay or bandwidth requirements. The IETF DCCP protocol [10] allows users to activate a less penalizing congestion control. However, it does not provide more QoS guarantees than UDP. SCTP targets the need for adaptation to network failures via the multi connection and multi homing concepts [11].

2.3 Adaptation Properties

The adaptation solutions suggested in the literature are defined in various ways. Behavioral vs. Architectural adaptation. Adaptation may be ruled by architecture or behavior-based transformation laws. In general, the adaptation is behavioural (or algorithmic) when the behaviour of the adaptive service can be modified, without modifying its structure. Standard protocols such as TCP and specific protocols such as [5] [7] provide behavior-based adaptation mechanisms. Behavioural adaptation is easy to implement but limits the adaptability properties. Indeed, the addition of new behaviours may be required. In this case, the component has to be recompiled and the adaptation can no longer be performed during run-time. The adaptation is architectural when the structure of adaptive services can be modified. [12,14] provides frameworks for designing middleware/Transport protocols whose internal structure can be modified according to the application requirements and network constraints. The replacement of a processing module by another(s) can be easily implemented, following a plug

and play approach where the new component has the same interfaces as the replaced one. Vertical vs. Horizontal adaptation. Adaptation may have a vertical (or local) or an horizontal (or distributed) scope. Adaptive components can be deployed on a single machine or distributed on several machines. In the first case, the adaptation is vertical and only local changes are performed. In the second case, it is horizontal and synchronization problems between peer adaptive entities have to be managed [15].

3 Cooperation and Communication in MEO-Like Activities

We consider a MEO team composed of a fixed controller, say C, and two investigators, say A and B, moving within the exploration field. Functions performed by investigators include Observing the explored field and Reporting feedbacks to the controller. Two kinds of feedbacks are distinguished. Feedbacks D are Descriptive data; they represent information describing the situation. They are transmitted as audio/video (a,v) data. Feedbacks P are Produced data; they may represent comments, reports or any analysis information explaining the situation. They are transmitted by means of audio (a). The controller's function includes Supervising the whole mission, i.e. deciding actions to be performed from the analysis of the observation feedbacks D and P.

The scenario is divided into two successive steps. The first step, Step 1, is the investigation step. Two investigators, A and B, provide continuous feedbacks D to a controller, C; they also provide periodical feedbacks P. There is no priority difference between communication links A-C and B-C, but transmitting feedbacks D is considered to be more important than transmitting feedbacks P.

The first step ends when a critical situation is discovered by an investigator, say A for instance. In the action step, Step 2, A conserves the same functions of observing and reporting (O, R) as in the exploration step but provides feedbacks D to both controller C and investigator B. A also provides feedbacks P to C. B reports now only feedbacks of type P to controller C on the basis of the feedback D received from investigator A. Due to the criticism of the situation reported by A, communication link A-C is considered to be more important than A-B and B-C. Moreover, we consider that exchanges of feedbacks D between A and C have the highest importance; feedbacks P between A and C have a medium importance; feedbacks D between A and B, and feedback P between B and C have the lowest importance.

4 Architectural Adaptation Models

We distinguish two abstraction levels for adaptability management the service adaptation level (S-Adapt) and the middleware adaptation level (M-Adapt). For space limitation, we present only the model of the S-Adapt abstraction level. An architectural reconfiguration, or horizontal model transformation, consists in

transforming an architecture into another architecture of the same abstraction level. Such reconfigurations may be guided by constraints evolving (such as link bandwidth variations) or by new objectives required by the group communication activity. A refinement, or vertical model transformation, associates a high level architectural model with a lower level model. Different mappings are possible for a given model.

The Service-level Adaptation (S-Adapt). Two services are provided by the investigators: observing (O) and reporting (R). A single service is provided by the controller: supervising (S) the mission. Two kinds of data are exchanged for performing controller's and investigators' services: descriptive data (D) and produced data (P).

Considering that $M1, M2$ and $M3$ represent the machines used respectively by A, B and C, we can define the communication links and priorities using graph-based notations as follows.

- $G_{step\ 1} = (Inv_{O,R}(A, M1) \xrightarrow{\langle D_{high}, P_{medium}\rangle} Cont_S(C, M3),$
  $Inv_{O,R}(B, M2) \xrightarrow{\langle D_{high}, P_{medium}\rangle} Cont_S(C, M3)).$
- $G_{step\ 2} = (Inv_{O,R}(A, M1) \xrightarrow{\langle D_{high}, P_{medium}\rangle} Cont_S(C, M3),$
  $Inv_R(B, M2) \xrightarrow{\langle P_{low}\rangle} Cont_S(C, M3),$
  $Inv_{O,R}(A, M1) \xrightarrow{\langle D_{low}\rangle} Inv_R(B, M2)).$

Where nodes are labelled by the roles and the provided services (O, R, S), the participant identifer (A, B, C) and the machine she/he uses (M_1, M_2, M_3). The edges are labelled by the type of exchanged information (D, P). The communication priority $(high, medium, low)$ is associated with each type of information.

When passing from step1 to step2, architectural transformation includes changes in the whole set of communications. A new communication is introduced between machines $M1(A)$ and $M2(B)$; and the communication of descriptive data (D) between $M2(B)$ and $M3(C)$ is removed. Moreover, priorities between communications are also changed.

The Middleware-level Adaptation (M-Adapt). Different architectures of the M-Adapt level may be considered to implement the current architecture of the S-Adapt level: they differ depending on the push/pull modes repartition, on the kind and the number of channels implemented by the channel manager and on the deployment of these channels. For instance in this example, two channels are considered for the channel manager: each channel is in charge of a specific pair (data, priority). Assuming a mobile participant is always allowed to host one event service component, the two channel managers ($CM1$ and $CM2$) are deployed on machines $M1(A)$ and $M2(B)$. $M3(C)$, a fixed machine with

permanent energy uses the pull mode. $M1(A)$, $M2(B)$ are mobile machines whose energy has to be preserved; for consumers. They use the push mode that consumes less energy than the pull mode.

4.1 The Graph Grammars for Architecture Transformation

This section provides an example of graph grammar to implement architecture reconfiguration. For space limitation reasons, we do not present grammars of the architecture refinement. In both cases, the proposed grammars generalize the use case by considering a variable number of investigators.

We use productions of type (L; K; R; C) where (L; K; R) corresponds to the structure of a DPO production [16] and where C is a set of connection instructions. The instructions belonging to C are of the edNCE type [17]. They are specified by a system $(n, p/q, \delta, d, d')$ where n corresponds to a node belonging to the daughter graph R, p and q are two edge labels, δ is a node label, and d and d' are elements of the set in, out. For example, a production defined by the system (L; K; R; $(n, p/q, \delta, d, d')$) is applicable to a graph G if it contains an occurrence of the mother graph L. The application of this production involves transforming G by deleting the subgraph (Del = L\K) and adding the subgraph (Add = R\K) while the subgraph K remains unchanged. All dangling edges will be removed. The execution of the connection instruction implies the introduction of an edge between the node n belonging to the daughter graph R and all nodes n' that are p-neighbours[1] of and d-neighbours[2]. This edge is introduced following the direction indicated by d' and labelled by q.

The following transformation rules allow transforming an architecture of the S-Adapt level in the exploration phase (step 1) to its corresponding configuration in the action phase (step 2). The graph grammar is reduced to a single production grammar $P_{exp \rightarrow act}$ (Table 1) which is parameterized by the identification of the investigator (here, noted A) that has discovered the critical situation. The architecture is transformed by splitting the communication channels between the controller and the other investigators into a communication channel of type P between these investigators and the controller and another communication channel of type D between them and A.

Table 1. Production grammar $P_{exp \rightarrow act}$

$$
\begin{aligned}
&P_{exp \rightarrow act}(A) = (p_1, C = \{ic_1, ic_2\}) \text{ with:}\\
&p_1 = (L = Cont(c, M1), Inv(A, M2), A \xrightarrow{<D_{high}, P_{medium}>} c,\\
&\qquad K = \{\},\\
&\qquad R = Cont(c, M1), Inv(A, M2), A \xrightarrow{<D_{high}, P_{medium}>} c)\\
&ic1 = (c, < D_{high}, P_{medium} > / < P_{low} >, Inv, in/in)\\
&ic2 = (A, < D_{high}, P_{medium} > / < D_{low} >, Inv, in/out)
\end{aligned}
$$

[1] p-neighbours of a node n are all nodes n' such that there exists an edge labelled by p which connects n and n'.

[2] In-neighbours if d=in and out-neighbours otherwise.

5 Conclusion

In this paper, we studied adaptability approaches aiming to support QoS provisioning in service-oriented communication-centric systems. We proposed graph-based architectural adaptability management models. The application of adaptability models and their transformation and mapping rules can help in automatically managing service provisioning in general. This may include functions such as power saving and service robustness. The management scope can involve the different steps of service provisioning including deployment. It can also address the design-time and the run-time architectural evolving management problems. Our models are now being completed and implemented. For this purpose, we have implemented a scalable graph transformation module. We built, on top of this module, an architecture transformation generic interface. For behavior-oriented adaptability actions, we have implemented a new transport protocol that supports run-time configuration actions for QoS optimization. Our present and future works focus on implementing the architecture-oriented and the behavior-oriented solutions by integrating our two separate solutions.

References

1. K. Farkas, O. Wellnitz, M. Dick, X. Gu, M. Busse, W. Effelsberg, Y. Rebahi, D. Sisalem, D. Grigoras, K. Stefanidis, D.N. Serpanos, "Real-time service provisioning for mobile and wireless networks", Elsevier Computer Communication Journal, Vol. 29, n 5, pages 540-550, march 2006.
2. R. Landry, K. Grace, A. Saidi, "On the Design and Management of Heterogeneous Networks: A Predictability-Based Perspective", IEEE Communications Magazine, Military and Tactical Communications, November 2004.
3. S. Balasubramaniam, J. Indulska, "Vertical handover supporting pervasive computing", Elsevier Computer Communication, Special Issue on 4G/Future Wireless networks, vol. 27/8, pp.708-719, 2004.
4. A. Kaloxylos, G. Lampropoulos, N. Passas, L. Merakos, "A flexible handover mechanism for seamless service continuity in heterogeneous environments", Elsevier Computer Communications, Vol. 29, Issue 6, Pages 717-729, March 2006,
5. D. Wu, Student Member, Y. T. Hou, W. Zhu, "Streaming Video over the Internet: Approaches and Directions", IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, no. 1, February 2001.
6. F. Yu, Q. Zhang, W. Zhu, Y-Q. Zhang. "QoS-adaptive proxy caching for multimedia streaming over the Internet", Circuits and Systems for Video Technology, IEEE Transactions on Volume 13, Issue 3, Page(s):257 - 269, March 2003.
7. . B. Akan, I. F. Akyildiz. "ATL: An Adaptive Transport Layer Suite for Next-Generation Wireless Internet", IEEE Journal on Selected Areas in Communications, vol.22, no.5, June 2004.
8. G.M. Perez, A.F. Gomez Skarmeta, "Policy-Based Dynamic Provision of IP Services in a Secure VPN Coalition Scenario", IEEE Communications Magazine, Military and Tactical Communications, November 2004.
9. I.W. Marshall, C. Roadknight, "Provision of quality of service for active services", Elsevier Computer Networks, Volume 36, Issue 1, Pages 75-85, June 2001.

10. Floyd S., Kohler E. "Profile for DCCP Congestion Control ID 3: TFRC Congestion Control", Internet Draft, December 2004.
11. Stewart R., Xie Q., et al. "Stream Control Transmission Protocol", IETF, RFC 2960, 2000.
12. G. T. Wong, M. A. Hiltunen, R. D. Schlichting, "A Configurable and Extensible Transport Protocol", IEEE INFOCOM, Anchorage, Alaska, April 22-26, 2001.
13. J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, A. Lopes, "Context Adaptation of the Communication Stack", Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'05), 6-10 June 2005, Columbus, OH, USA.
14. E. Exposito, M. Diaz, P. Snac. "FPTP: the XQoS aware and fully programmable transport protocol", 11th IEEE International Conference on Networks (ICON'2003), Sydney, Australia, 28 September - 1st October 2003.
15. P. G. Bridges, W-K. Chen, M. A. Hiltunen, Richard D. Schlichting, "Supporting Coordinated Adaptation in Networked Systems", 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001.
16. H. Ehrig and M. Korff and M. Lowe. Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts, 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, Bremen, Germany, LNCS 532, Springer-Verlag, March, 1990, pp24-37.
17. G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific Publishing, ISBN 981-02-2884-8, 1997.

Automating the Building of Software Component Architectures

Nicolas Desnos[1], Sylvain Vauttier[1], Christelle Urtado[1], and Marianne Huchard[2]

[1] LGI2P / Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes, France
{Nicolas.Desnos, Sylvain.Vauttier, Christelle.Urtado}@site-eerie.ema.fr
[2] LIRMM - UMR 5506 - CNRS and Univ. Montpellier 2
34 392 Montpellier cedex 05, France
huchard@lirmm.fr

Abstract. Assembling software components into an architecture is a difficult task because of its combinatorial complexity. There is thus a need for automating this building process, either to assist architects at design time or to manage the self-assembly of components at runtime. This paper proposes an automatic architecture building process that uses ports, and more precisely composite ports, to manage the connection of components. Our solution extends the Fractal component model. It has been implemented and experiments have been run to verify its good time performance, thanks to several optimization heuristics and strategies.

1 Introduction and Motivation

Software engineering aims at optimizing the cost of design and maintenance while preserving both the quality and reliability of the produced software. Component-based development techniques try to enhance reuse [1,2,3]. The design process of an application is led by an architect and decomposes into three steps: he selects components, defines an architecture by assembling them[1] and then uses a tool to control the consistency of the assembly to determine if the assembled components are compatible. Components are generally described as a set of interfaces that define what a component can provide and must require. Component assemblies are then built by connecting component interfaces together [4,5,6,7,8].

Most existing works do not provide architects with any guidance during the selection and assembly steps. They rather focus on checking the validity of a previously built architecture [6,9,10,11,12]. The consistency check techniques cannot be used in an iterative building process because of the combinatorial complexity [13]. To guide the architect, we propose an efficient approach to automatically build potentially valid architectures. It produces a reduced set of preselected component assemblies on which it is relevant to perform checks to find valid architectures. It relies on the use of ports, and more precisely of composite ports, to describe known usages of components. A construction

[1] In these works, we will consider that the selected components need no adaptation (or might have already been adapted).

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 228–235, 2006.
© Springer-Verlag Berlin Heidelberg 2006

algorithm has been successfully implemented and experimented in the Fractal component model [7].

The remainder of this paper is organized as follows. Section 2 discusses the issues raised by the building of valid architectures and introduces a component model which features primitive and composite ports. Section 3 describes a basic algorithm to automatically build architectures along with its optimizations. Section 4 concludes and draws perspectives.

2 Building Valid Architectures

2.1 An Augmented Component Model to Ease Construction

Not to start from scratch, we choose to extend an existing component model named Fractal [7][2]. Classically, a Fractal component is described as a black box that defines the services the component provides and requires through server and client **interfaces** and a content (called the **architecture**) that allows a component to be recursively described. Fractal components are assembled into architectures by connecting client interfaces to server interfaces. This allows components to collaborate by exchanging messages along these connections.

The Fractal model is first extended with ports. As in UML2 [4], ports are used to group together the client and server interfaces that are used by a component in a given collaboration. Ports are thus used to specify various usage contexts for components. We define two kinds of ports. **Primitive ports** are composed of interfaces, as in many other component models [4,6,10,12,14]. **Composite ports** are composed of other ports. Composite ports are introduced to structurally represent complex collaborations. Figure 1 shows an architecture where *ATM* is an example of component, *Question* one of its provided interfaces, *Transaction* one of its required interfaces and *Money_Withdraw* its composite port which is composed of the two *Money_Dialogue* and *Money_Transaction* primitive ports. Two primitive ports are connected together when all the interfaces of the first port are connected to interfaces of the second port (and reciprocally). A composite port is connected when all the primitive ports it is composed of (directly or indirectly) are connected. Component architectures can then be built by connecting together component ports (what entails interface connections). Next section details how ports, and more precisely composite ports, make the building of architectures easier.

2.2 Validity of an Architecture

An architecture is said to be valid if it is both correct and complete.

Correctness. Stating the correctness of an architecture relies on techniques that verify the coherence of connections, to check whether they correspond to

[2] We choose Fractal mainly because it is a hierarchical composition model that supports component sharing, its structure is simple but extensible and respects the separation of concerns principle and an open-source implementation exists.

possible collaborations between the linked components. These verifications use various kinds of meta-information (types, protocols, assertions, etc.) associated with various structures (interfaces, contracts, ports, etc.).

A first level of correctness, called **syntactic correctness**, can be verified by comparing the types of the connected interfaces [5,7]. This ensures that components can "interact" because the signatures of the functionalities to be called through the required interface match the signatures of the functionalities of the provided interface. A second level of correctness, called **semantic correctness** [15,9], can then be verified to determine if the connected components can "collaborate" i.e. exchange sequences of messages that are coherent with each other's behavior. Semantic verifications require that **protocols** – valid sequences of messages – be defined. The semantic correctness of the connection between two ports is handled as a classic comparison of their associated protocols. This is a time-consuming process because of the highly combinatorial complexity of the algorithms used to compare all the possible message sequences [13].

Completeness. A component architecture is built to achieve some **functional objectives** [1,15,16]. Functional objectives are defined as a set of functionalities to be executed on selected components. The set of connections in the architecture must be sufficient to allow the execution of collaborations that reach (include) all the functional objectives. Such an architecture is said to be **complete**.

Starting from a set of components corresponding to the functional objectives, a naive algorithm can be to try to build an architecture where all the interfaces of all the components are connected, so that all the execution scenarios may be executed. When no solution exists in the current architecture to connect an interface, the repository is searched for a component that has a compatible interface. If one exists, it is added to the architecture and the interfaces are connected. If several connections are possible, they represent alternative building paths to be explored. In case a dead end is reached, the construction is backtracked to a previous configuration, in order to try alternative connection combinations. The problem with this building process is the size of the solution space to be explored. It is amplified by the cost of the semantic verifications that must be calculated for any candidate connection between two components. Therefore, the automatic construction of valid architectures still is an open problem. We then have studied different ways to reduce the complexity of the building process.

3 Taming the Complexity of Automation

3.1 Using Composite Ports to Connect Components

To reduce the complexity, the building process can try to connect only the interfaces that are useful to reach the functional objectives. However, the proper use of a functionality of a component is not independent from other functionalities. The behavior protocol of a component specifies the different valid execution scenarios where a functionality is called. The execution of a scenario

requires the connection of all the interfaces that it uses: regarding the scenario, these interfaces are said to be **dependent**. Thus, a given functional objective can be reached only when precise sets of (dependent) interfaces, corresponding to valid scenarios, are connected. An analysis of the behavior protocol of a component could be used to determine those scenarios but a means is required to capture and to express this information in an explicit and simple way, in order to ease the connection process. Ports are introduced as a kind of structural meta-information, complementary to interfaces, that group together the interfaces of a component corresponding to a given valid scenario. Ports could be produced automatically, by the analysis of behavior protocols or be manually added by the designer in order to document a given usage of the component.

Port connections make the building process more abstract (port-to-port connections) and more efficient (no useless connections). Considering a port that needs to be connected, the availability of a compatible port is an important issue. The more numerous interfaces are in a given port, the more specific the port type is and the less chances exist to find compatible ports. Composite ports are used to solve this issue: they allow short scenarios, composed of few interfaces, to be described as small primitive ports that are then composed together to describe more complex scenarios. Large flat primitive ports can then be replaced by small primitive ports hierarchically structured into larger composite ports. The result is that smaller ports are less specialized and thus provide more connection possibilities. From a different point of view, a primitive port can be considered as the expression of a constraint to connect a set of interfaces both at the same time and to a unique component. A composite port is the expression of a constraint to connect a set of interfaces at the same time but possibly to different components. As they relax constraints, composite ports increase the amount of possible connection combinations. Moreover, composite ports provide a means to precisely specify how interfaces must be connected: to a unique component – for functionality calls to produce cumulative effects – or to distinct components.

3.2 Building Quasi-valid Architectures

Semantic verifications are very expensive. Our approach keeps semantic verifications separated from the building process so as not to waste time verifying the semantics of connections as long as the completeness of the architecture cannot be guaranteed. To achieve this, a **quasi-valid** architecture is first built. A quasi-valid architecture is a syntactically correct and complete architecture. The connection of a port enforces the completeness of an architecture, regarding the execution of a scenario. Once all the ports corresponding to the functional objectives are connected, an architecture is quasi-valid. Quasi-validity is a precondition for an architecture to be valid.

We wrote an algorithm that automatically builds quasi-valid architectures. The building process uses a set containing the ports that still have to be connected – the functional objective set (FO-set). The FO-set contains only

primitive ports: composite ports are systematically decomposed into the set of primitive ports they are directly or indirectly composed of. The FO-set is initialized with the ports that correspond to the functional objectives. One of the primitive ports is picked up from the FO-set and a compatible port is searched for. If a compatible unconnected port is found, the ports are connected together. If the compatible port belongs to a component that does not yet belong to the architecture, the component is added to the architecture. If the chosen compatible port belongs to a composite port, all the other primitive ports that composed the composite port are added to the FO-set. This way, no port dependencies – and therefore no interface dependencies – are left unsatisfied. The building process is iterated until the FO-set is empty. All the initial primitive ports that represent functional objectives are then connected along with all ports they are recursively dependent upon: the resulting architecture is quasi-valid.

Figure 1 shows the example of an architecture built by our algorithm. It starts with a FO-set that contains the *Money_Withdraw* primitive port of the *Client* component. This port is taken out of the FO-set and a connection is searched for. It is connected to the compatible *Money_Dialogue* primitive port of the *ATM* component. As this latter port belongs to the *Money_Withdraw* composite port, it depends on the *Money_Transaction* primitive port which is thus added to the FO-set before the building process iterates. The *Money_Transaction* primitive port of the *ATM* component is now considered for connection. It is compatible with the *Money_Transaction* primitive port of the *Bank* component which belongs to the composite port *Money_Withdraw*. After connection, the other primitive port of this composite port, *Request_Data*, is in turn added to the FO-set. At the next iteration, the *Request_Data* primitive port of the *Bank* component is connected with the compatible primitive port *Provide_Data* of the *Database* component. As this primitive port does not belong to a composite port, no primitive port is to be added to the FO-set. The FO-set is now empty: the architecture of Fig.1 is quasi-valid.

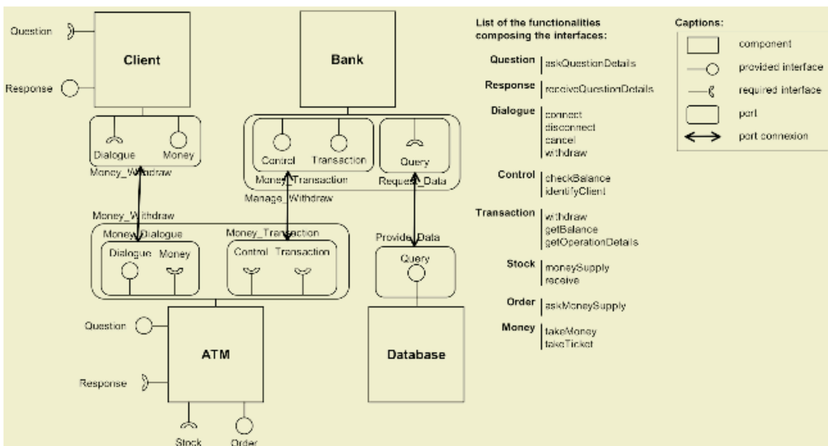


Fig. 1. A quasi-valid architecture built with the support of composite ports

Several special situations can occur during this process. When several free compatible ports are candidate for connection, they correspond to alternate solutions that are to be explored. Conversely, when no free compatible port is found the building algorithm has reached a dead end. The construction is then backtracked to a previous situation where unexplored connection possibilities exist. Our algorithm is implemented as the searching of a construction tree using a depth-first policy. Breadth search is used to explore all the alternate construction paths. This complete exploration of the construction tree is used to guarantee that any possible solution is always found.

3.3 Strategies, Heuristics and Experiments

The performance of the building algorithm has been measured. For this purpose, we have implemented a small environment that generates random component sets which provide different building contexts, in size and complexity. Once a component set is generated, an arbitrary number of ports can be chosen as functional objectives and the building algorithm be launched. Our experiments show that the combinatorial complexity of the building process is very high. To be able to use our approach in demanding situations, such as the deployment and configuration of components at runtime, we have studied various heuristics that speed up the building process.

Building Minimal Architectures. A first strategy is to try to find not all the possible architectures but only the most interesting ones. Minimality is an interesting metrics for the quality of an architecture [17]. We apply this minimality criterion to the number of connection. Less connections entail less semantic verifications, less interactions and therefore less conflict risks. Less connections also entail more evolution capabilities (free ports). To efficiently search for minimal architectures, we have added a branch-and-bound strategy to our building algorithm. The bound is the maximum number of connections allowed for the construction of the architecture. When this maximum is reached when exploring a branch of the construction tree, the rest of the branch can be discarded as any new solution will be less optimal than the previously found (pruning).

Min Domain Heuristic. This heuristic is used to efficiently choose ports from the FO-set. The port for which a minimum of free compatible ports exists is chosen first. This minimizes the effort to try all the connection possibilities: in case of repeated failures, this allows impossible constructions to be detected sooner.

Minimum Effort Heuristic. In the branch-and-bound strategy, every time the bound is lowered, the traversal of the tree is speeded up. To connect a primitive port, the algorithm first chooses the free compatible primitive port that belongs to the "smallest" composite port. It corresponds to the choice of the less dependent ports, that minimize future efforts to connect them.

No New Dependency Heuristic. When a compatible port can be found in the FO-set its connection will add no new dependency, and furthermore, satisfy two dependencies at once. Indeed, when a port belongs to the FO-set, the other primitive ports it depends on are already in the FO-set.

Look-ahead Strategy. Calculi can be used to predict if the traversal of the current construction branch can lead to a minimal solution. They are based on an estimate of the minimum number of connections required to complete the building. A soon as the sum of the existing connections with this estimate is greater than the bound, the current branch can be pruned. A simple example of this estimate is the number of ports in the FO-set divided by two.

Experimental Results: An Outline. Experiments show that performance mainly depends on the number of initial functional objectives. This is logical since more functional objectives implies not only a larger search space but also more constraints, thus more failures and backtracks. For example, series of experiments have been run with a library of 38 generated components. Each component had at most 4 primitive ports and at most 2 composite ports. Each primitive port had at most 5 interfaces. Starting with 5 initial functional objectives, the following typical results are obtained. A basic construction algorithm, implemented in Java and executed on a standard computer, without any of the above optimizations, is able to find 325 000 quasi-valid architectures, when stopped after 15 hours. This gives an idea of the gigantic size of the search space. Among those quasi-valid architectures, the largest ones are composed of 48 connections. The smallest architecture found is composed of 18 connections. As a comparison, the optimized construction algorithm finds the only minimal architecture composed of 7 connections in less than a second. This motivates our proposal for an efficient building approach. It is difficult to build quasi-valid architectures, because the more frequent ones are rather large (around 40 connections in the above example). It is even more difficult to build minimal ones, because they are scarce in a large search space.

4 Conclusion and Perspectives

While other works focus on the validation of complete architectures, our work studies the building process of architectures and proposes a practical solution to automate it. It enables the candidate architectures, on which validation algorithms are to be applied, to be systematically searched for. Besides the many optimization strategies and heuristics used for the traversal of the construction space, the use of ports, and particularly of composite ports, is prominent in our approach. As they express the dependencies that exist between interfaces, ports provide a simple means to evaluate the completeness of an architecture. Finally, being composed of interfaces, they provide means to abstract the many connections of interfaces to single connections and thus reducing the combinatorial complexity of the building.

A perspective for this work is to integrate it to a component-based development framework, for example as part of a trading service, to provide a means to manage the self-assembling of components in open, dynamic systems (autonomic computing).

References

1. Crnkovic, I.: Component-based software engineering - new challenges in software development. Software Focus (2001)
2. Garlan, D.: Software Architecture: a Roadmap. In: The Future of Software Engineering. ACM Press (2000) 91–101
3. Brown, A.W., Wallnau, K.C.: The current state of CBSE. IEEE Software **15**(5) (1998) 37–46
4. OMG: Unified modeling language: Superstructure, version 2.0 (2002) http://www.omg.org/uml/.
5. OMG: Corba components, version 3.0, http://www.omg.org/docs/formal/02-06-65.pdf(2002)
6. Traverson, B.: Abstract model of contract-based component assembly (2003) ACCORD RNTL project number 4 deliverable (in french).
7. Bruneton, E., Coupaye, T., Stefani, J.: Fractal specification - v 2.0.3 (2004) http://fractal.objectweb.org/specification/index.html.
8. Plásil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: Proceedings of the Int. Conf. on Configurable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1998) 43–52
9. Plásil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11) (2002) 1056–1076
10. Hacklinger, F.: Java/A - Taking Components into Java. In: IASSE. (2004) 163–168
11. Farías, A., Sudholt, M.: On components with explicit protocols satisfying a notion of correctness by construction. In Meersman, R., Tari, Z., et al., eds.: On the Move to Meaningful Internet Systems: Int. Conf. CoopIS, DOA, and ODBASE Proc. Volume 2519 of LNCS., Springer (2002) 995–1012
12. de Boer, F.S., Jacob, J.F., Bonsangue, M.M.: The OMEGA component model. Deliverable of the IST-2001-33522 OMEGA project (2002)
13. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Trans. Softw. Eng. Methodol. **9**(3) (2000) 239–272
14. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: Proceedings of ICSE, Orlando, Florida, USA, ACM Press (2002) 187–197
15. Dijkman, R.M., Almeida, J.P.A., Quartel, D.A.: Verifying the correctness of component-based applications that support business processes. In Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proc. of the 6th Workshop on CBSE: Automated Reasoning and Prediction, Portland, Oregon, USA (2003) 43–48
16. Inverardi, P., Tivoli, M.: Software Architecture for Correct Components Assembly. In: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. Volume 2804 of LNCS. Springer (2003) 92–121
17. Cechich, A., Piattini, M., Vallecillo, A., eds.: Component-Based Software Quality: Methods and Techniques. Volume 2693 of LNCS. Springer (2003)

Component Deployment Evolution Driven by Architecture Patterns and Resource Requirements

Didier Hoareau and Chouki Tibermacine

VALORIA Lab., University of South Brittany, France
{Didier.Hoareau,Chouki.Tibermacine}@univ-ubs.fr

Abstract. Software architectures are often designed with respect to some architecture patterns, like the pipeline and peer-to-peer. These patterns are the guarantee of some quality attributes, like maintainability or performance. These patterns should be dynamically enforced in the running system to benefit from their associated quality characteristics at runtime. In dynamic hosting platforms where machines can enter the network, offering new resources, or fail, making the components they host unavailable, these patterns can be affected. In addition, in this kind of infrastructures, some resource requirements can also be altered. In this paper we present an approach which aims at dynamically assist deployment process with information about architectural patterns and resource constraints. This ensures that, faced with disconnections or machine failures, the runtime system complies permanently with the original architectural pattern and the initial resource requirements.

1 Introduction

When we design software architectures, we often make use of architecture patterns, like for example the pipe and filter, the client and server, peer-to-peer pattern, etc. These vocabularies of recurrent solutions to recurrent problems[1] are the guarantee of some quality attributes in the designed system. These quality attributes include maintainability, portability, reliability and performance. Starting from these high-level design documents (architecture descriptions), we can produce low-level implementation entities that will be deployed.

One of the characteristics of emerging distributed platforms is their dynamism. Indeed, such dynamic platforms are not only composed of powerful and fixed workstations but also of mobile and resource-constrained devices (laptops, PDAs, smart-phones, sensors, etc.). Due to the mobility and the volatility of the hosts, connectivity cannot be ensured between all hosts, e.g. a PDA with a wireless connection may become unaccessible because of its range limit. As a consequence, in a dynamic network, partitions may occur, resulting in the fragmentation of the network into islands. Machines within the same island can communicate whereas, no communication is possible between two machines that are in two

[1] With analogy to design patterns but at a more coarse-grained level of abstraction.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 236–243, 2006.
© Springer-Verlag Berlin Heidelberg 2006

different islands. Moreover, as some devices are characterized by their mobility, the topology of islands may evolve.

Dynamism in the kind of networks we target is not only due to the nature of the devices but also to their heterogeneity making difficult to base a deployment on resource's availability. When deploying component-based software in dynamic distributed infrastructures it is required that the deployed system complies permanently with its corresponding architecture pattern(s). By taking advantages of changes in the environment (e.g. availability of a required resource), the initial deployment can evolve but any reconfiguration must respect architectural choices. This makes the running system benefit from the targeted quality attributes, and more particularly those which are dynamically observed, like performance or reliability.

In this paper, we present an approach to drive component deployment and component deployment evolution in this kind of dynamic networks, based on information about architecture patterns and resource requirements. This approach uses two kinds of constraints: the first one represents patterns and resource requirements formalisation; the second one corresponds to the result of transforming the former constraints into run-time ones. These run-time constraints are checked dynamically and are used to drive component deployment and component deployment evolution.

In the next section we present how we can formalize architecture patterns and resource requirements using a constraint language, and we illustrate this formalization by a short example of a client/server pattern. We present in section 3, the deployment process and the resolution mechanisms of these constrained component-based software in dynamic infrastructures. Before concluding, we present some related work in section 4.

2 Formalization of Architectural Decisions with ACL

In order to make explicit architectural decisions, we proposed ACL, an Architecture Constraint Language [11]. Architectural decisions are thus formalised as architecture predicates which have as a context an architectural element (component, connector, etc.) that belongs to an architecture metamodel. ACL is a language with two levels of expression. The first level encapsulates concepts used for basic predicate-level expression, like quantifiers, collection operations, etc. It is represented by a slightly modified version of UML's OCL [9], called CCL (Core Constraint Language). The second level embeds architectural abstractions that can be constrained by the first level. It is represented by a set of MOF architecture metamodels. Architectural constraints are first-order predicates that navigate in a given metamodel and which have as a scope a specific element in the architecture description. Each couple composed of CCL and a given metamodel is called an ACL profile. We defined many profiles, like the ACL profile for xAcme (which is an XML extension of Acme ADL [3].), for UML 2, for OMG's CORBA Components [8] (CCM) or the profile for ObjectWeb's Fractal [1].

2.1 Architectecture Pattern Description

Suppose that we have developed, at architecture design-time, a component software that represents a company printing system. We would like to automate the installation and the reconfiguration of this system to all company employees. This printing system is organised according to the client/server pattern. The printing service is based on a ServerPrinter which receives print jobs from ClientPrinter. The client/server pattern is characterized by the following constraints: i) there is no direct communication between ClientPrinters, ii) each ServerPrinter can accept jobs from at most 10 different clients, and iii) a ClientPrinter can use at most two ServerPrinters. These first two constraints can be described using ACL profile for Fractal as following:

```
context ClientServer:CompositeComponent inv:
ClientServer.binding->forAll(b|b.client.component.kind
<> b.server.component.kind)
and
ClientServer.subComponents->select(c|c.kind='Server')
.interface->oclAsType(Server).binding->size() <= 10
```

These constraints navigate in the MOF metamodel of Fractal ADL which is presented in Figure 1. This metamodel abstracts components, which can be composite or primitive. Composite (or hierarchical) components are entities which have an explicit description of their internal parts. Primitive (or atomic) components are directly implemented by an object class. Components express their functionalities and requirements through respectively, server and client interfaces. In addition, controller interfaces embed non-functional specifications, such as predefined operations which manage the lifecycle or the contents of a given component. A composite component specifies also a set of bindings which are simple method invocation connectors. These bindings are attachments between client and server interfaces. Bindings can represent either hierarchical or assembly connectors (with analogy to UML's delegation and assembly connectors). Hierarchical connectors bind interfaces of composite components to interfaces of their sub-components. Assembly connectors bind interfaces of components of the same level of hierarchy.

2.2 Resource and Location Requirements Description

In addition to these architecture design constraints, the deployment of each component is governed by some resource and location requirements. Indeed, at design time, we are unlikely to know the machines that are involved in the deployment and thus where to deploy each component. However, one can define for each component its requirements in term of resources, that is, the characteristics of the machines that will host the component. For example, a ServerPrinter must be hosted by a machine that has at least 512 MB of free memory, a CPU scale greater than 1 GHz (1000 MHz) and that is connected to a printer.

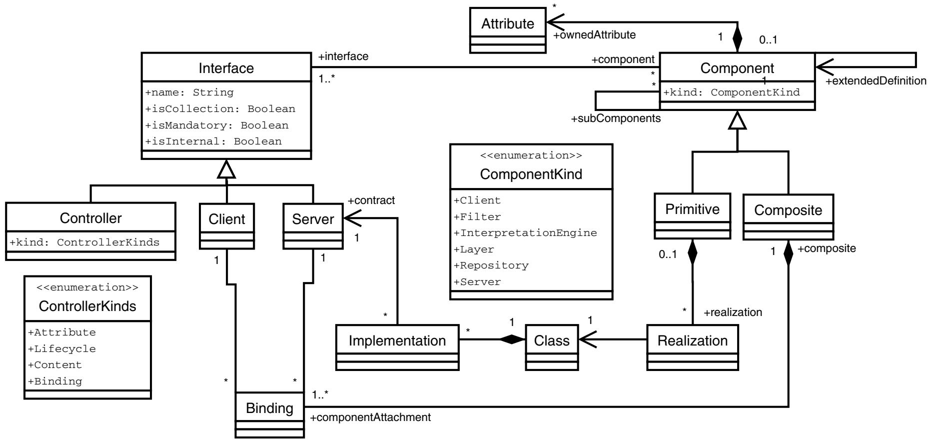


Fig. 1. A MOF metamodel of Fractal ADL

Resource constraints can be defined using an ACL profile (i.e. a CCL and a metamodel), called R-ACL (Resources-ACL). R-ACL integrates in its metamodels concepts related to system resources and their properties[2]. Resource constraints introduced above and related to ServerPrinters components are described in R-ACL as following:

```
context ServerPrinter:Component inv:
ServerPrinter.resource->oclAsType(Memory).free >= 512
and
ServerPrinter.resource->oclAsType(CPU).processors
->select(cpu:CPU_Model|cpu.speed > 1000)->size() >= 1
and
ServerPrinter.resource->oclAsType(Devices)
->select(printer:Printer)->size() >= 1
```

As discussed above, these constraints navigate in the resources metamodel, but have as a scope a specific architectural element (ServerPrinter component).

Besides resource constraints, it is sometimes required to control the placement of the components, especially when several machine can host the same component. For example in the Client/Server system we designed, we would require that for reliability reasons (redundancy at the server side), all ServerPrinters have to be located on distinct hosts. The following listing illustrates this constraint expressed in R-ACL.

```
context ClientServer:CompositeComponent inv:
ClientServer.subComponent->select(c1,c2:Component|c1.kind='Server'
and c2.kind='Server' and c1.location.id <> c2.location.id)
```

[2] The resources metamodel is not presented in this paper due to space limitations.

3 Constrained Components' Deployment in Dynamic Infrastructures

When the choice of the placement of every component has to be made, the initial configuration of the target platform may not fulfil all resources' requirements of the application and some needed machines may not be connected. We are thus interested in a deployment that allows the instantiation of the components as soon as resources become available or new machines become connected. We qualify this deployment as *propagative*. We propose a general framework to guarantee the designed architecture and its instances for each deployment evolution.

We present first the requirements of a deployment driven by pattern and resource specifications. Then, we detail the deployment process and the resolution of constraints in dynamic environments.

3.1 From Architectural Constraints to Runtime Constraints

At design time, we are unlikely to know what are the machines that are involved in the deployment and thus what are their characteristics. Hence, a valid configuration of the client/server pattern presented in section 2, can only be computed at runtime. A valid configuration is a set of component instances, interconnected and for which, a target host has been chosen. Every architectural constraint (e.g. on bindings or number of instances) has to be verified and the selected hosts must not contradict the resource and location constraints.

Our approach consists in manipulating all the architectural and resource constraints at runtime in order to reflect the state of the deployed system with respect to these constraints. The reified constraints are generated automatically from the R-ACL constraints and correspond to a constraint satisfaction problem (CSP). In a CSP, one only states the properties of the solution to be found by defining variables with finite domains and a set of constraints restricting the values that the variables can simultaneously take. The use of solvers such as Cream[3] can then be used to find one or several solutions. The CSP that corresponds to our patterns consists of the following constraints:

C1 the number of instances allowed for each component
C2 the resource constraints (e.g. $Mem.free \geq 512$)
C3 the location constraints (e.g. $x \neq y$)
C4 a binding constraint between every component that can be bound
C5 the number of outgoing bindings allowed on a client interface
C6 the number of incoming bindings allowed on a server interface

Each C_i corresponds to a set of constraints. As we will detail below, these sets are sufficient to generate a valid configuration regarding to an architectural pattern. The deployment process that is presented in the next section relies on these constraints in order to build a mapping between the component instances and the hosts of the target platform.

[3] `http://kurt.scitec.kobe-u.ac.jp/~shuji/cream/`

3.2 Deployment Process

When dealing with dynamic networks where partitions may occur and hosts availability has to be faced with, it is hardly feasible to rely on a specific machine which would be responsible of the deployment. We made the most of the results obtained in [5] in which we have used a consensus algorithm to elect a manager that decides on the placement of a set of components. The consensus algorithm ensures that no contradictory decisions can be made in two different islands, e.g. the same component cannot be instantiated in two distinct islands.

The deployment descriptor contains the identity of the machines that are involved in the deployment. This requirement is necessary in order to define the notion of majority on which the consensus relies. However, when the deployment is triggered, some machines may not be connected. The first step of the deployment consists in broadcasting the architecture and deployment descriptors to at least one machine that belongs to the deployment target, which in turn broadcasts the descriptors to all the machines that are connected in the network. Each machine that receives these descriptors, creates the constraints described in the listing above depending on the deployment and architecture descriptors. Then a process is launched on each host. Locally, each machine maintains its own set of constraints (C1 to C6) and tries to make the deployment evolve until (a) solution(s) exist(s) for constraints C1, that is, some components can still be instantiated. The main steps of this process for the machine m_i and a component C that can be deployed on m_i are:

- For each resource constraint associated with C, a dedicated probe is launched (e.g. a probe to get the amount of free memory required by component C) in order to check if locally, all the required resources are available (C2). The observation of the resources is made periodically.
- If this is the case, that is, the component can be hosted locally, m_i sends its candidatures to all the machines involved in the deployment. This candidature indicates that m_i can host component C.
- Thus, m_i may receive several candidatures from others for the instantiation of C. When a candidature is received, m_i has to resolve a placement solution regarding to constraints C3. Depending on location constraints, a placement solution may require a sufficient number of candidatures.
- Once a solution has been found by m_i, it tries to make it adopt by the consensus algorithm. If the consensus terminates, m_i updates the deployment descriptor with the new information of placement and broadcasts it to all the nodes that are currently connected.
- When a new descriptor is received, m_i updates the set C1 and C3 in order to take into account the placement decision made previously.
- m_i can then resolve some bindings towards newly instantiated (remote) components (C4) by sending a request to the machines hosting them. This is possible only if constraints C5 are still verified.
- When m_i receives a request of bindings, according to C6, it can accept or not this request and inform the sender of its answer.

– Depending on the answer, the definition domain that corresponds to the binding constraint (C4) is updated (removed from the constraint set if the binding is not possible or set to the remote host otherwise).

This process defines a propagative deployment driven by architectural and resources concerns. Since the observation of resources is made periodically, when a resource becomes available on a specific machine, this may yield the deployment to evolve. Similarly, when a machine enters the network (e.g. it is switch on), it announces its presence to the other nodes which will send it the current version of the architecture and deployment descriptors, making possible this newly connected machine to participate in the deployment evolution. In our current prototype, each machine maintains the list of connected hosts.

4 Related Work

Many ADLs provide capabilities to describe architecture patterns. Medvidovic and Taylor in [7] makes an overview of some existing ADLs offering such functionalities. Descriptions of architecture patterns with these ADLs make possible some reasoning about the modeled system, analysing its structure and evaluating its quality. At the best of our knowledge, only a few of these ADLs allow the enforcement of architecture patterns on an implementation deployed at runtime in a dynamic infrastructure. Some of the works targeting this goal are presented below. In addition, resource and location requirements are not well handled in all these languages in a homogeneous manner with architecture patterns like in R-ACL. If we would like to change an implementation technology (from Fractal to CORBA components, for example), R-ACL constraints can be easily transformed, as demonstrated in [12]. The solution adopted in this work which aims at transforming R-ACL constraints into runtime constraints makes also simpler the transformation of these new R-ACL constraints (in CORBA components ACL profile, for example).

We share similarities with researches on self-healing and self-organizing systems [6]. Indeed, the proposed approach here ressembles to the approach of [4,10] in which the architecture of the system to deploy is not described in terms of component instances and their interconnections but rather by a set of constraints that define how components can be assembled. In both cases the running system is modelled by a graph. The main difference with our work is that reconfigurations of the systems are explicitly defined in a programmatic way while this is achieved automatically by the resolution of the constraints C_i in our work.

The work presented in [2] shares the same motivation to define high level deployment description with regard to constraints on the application assembly and on the resources the hosts of the target platform should meet. The authors present the Deladas language that allows the definition of a deployment goal in terms of architectural and location constraints. A constraint solver is used to generate a valid configuration of the placement of components and reconfiguration of the placement is possible when a constraint becomes inconsistent. This centralized approach does not consider resource requirements.

5 Conclusion and Ongoing Work

Deploying distributed systems in dynamic infrastructures remains a challenging task as resources and hosts availability cannot be predicted. In this paper we presented an approach which helps at assisting the deployment process with information about architecture patterns and resource requirements. This information is formally specified at design-time as constraints, written with a specific predicate language. These constraints allow the definition of complex component interaction and platform dependencies. In order to react on changes in the environment, these constraints are transformed and manipulated dynamically. By using these constraints, a propagative deployment is defined: components are instantiated as soon as needed resources become available and required hosts become connected while ensuring architecture consistency.

Our implementation is based on existing prototypes: ACE [11] for the description and the evaluation of ACL constraints, and a deployment manager based on Cream to maintain and solve runtime constraints. An evaluation of the behavior of our approach in a dynamic network is in progress.

References

1. E. Bruneton, C. T., M. Leclercq, V. Quéma, and S. Jean-Bernard. An open component model and its support in java. In *Proceedings of CBSE'04*, may 2004.
2. A. Dearle, G. N. C. Kirby, and A. J. McCarthy. A framework for constraint-based deployment and autonomic management of distributed applications. In *Proceedings of ICAC'04*, pages 300–301, 2004.
3. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge Univ. Press, 2000.
4. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of WOSS'02*, pages 33–38, 2002.
5. D. Hoareau and Y. Mahéo. Constraint-based deployment of distributed components in a dynamic network. In *Proceedings of ARCS 2006, LNCS, volume 3864*, pages 450–464, 2006.
6. J. Magee and J. Kramer. Self organising software architectures. In *Proceedings of FSE'96*, pages 35–38, 1996.
7. N. Medvidovic and N. R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.
8. OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. OMG Web Site: http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.
9. OMG. Uml 2.0 ocl final adopted specification, document ptc/03-10-14. OMG Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.
10. B. R. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *In proceedings of SEKE'02*, pages 241–248, 2002.
11. C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of WICSA'05*, pages 121–130, November 2005.
12. C. Tibermacine, R. Fleurquin, and S. Sadou. Simplifying transformations of architectural constraints. In *Proceedings of SAC'06, Track on Model Transformation*, April 2006.

Author Index

Acuña, Cesar J. 127
Araújo, João 159
Ardissono, Liliana 2
Armando, François 220
Astudillo, Hernán 204
Autili, Marco 17

Barrett, Ronan 144
Bashroush, Rabih 212
Batista, Thais 82
Brown, John 212
Büchner, Thomas 33
Bulej, Lubomír 50
Bureš, Tomáš 50

Chassot, Christophe 220
Chavez, Christina 82
Cortellessa, Vittorio 66
Cuesta, Carlos E. 127

Desnos, Nicolas 228
Drira, Khalil 220

Exposito, Ernesto 220

Flammini, Michele 17
Fuentes, Lidia 159
Furnari, Roberto 2

Garcia, Alessandro 82
Goy, Anna 2
Guennoun, Karim 220

Haugen, Øystein 98
Hirsch, Dan 113
Hoareau, Didier 236
Huchard, Marianne 228

Inverardi, Paola 17

Kilpatrick, Peter 212
Kramer, Jeff 113
Kulesza, Uirá 82

López, Claudia 204
Lozes, André 220
Lucena, Carlos 82

Magee, Jeff 113
Magno, José 159
Marcos, Esperanza 127
Marinelli, Fabrizio 66
Matthes, Florian 33
Møller-Pedersen, Birger 98
Moreira, Ana 159

Navarra, Alfredo 17

Pahl, Claus 144
Pereira, Javier 204
Petrone, Giovanna 2
Potena, Pasqualina 66

Rashid, Awais 82

Sánchez, Pablo 159
Sant'anna, Claudio 82
Schäfer, Clemens 175
Schmidt, Holger 189
Segnan, Marino 2
Spence, Ivor 212

Taylor, Richard N. 1
Tibermacine, Chouki 236
Tivoli, Massimo 17

Uchitel, Sebastian 113
Urtado, Christelle 228

Vauttier, Sylvain 228

Wentzlaff, Ina 189