Ivica Crnkovic
Volker Gruhn
Matthias Book (Eds.)

LNCS 6903

Software Architecture

5th European Conference, ECSA 2011
Essen, Germany, September 2011
Proceedings

Springer

Lecture Notes in Computer Science 6903

Ivica Crnkovic Volker Gruhn
Matthias Book (Eds.)

Software Architecture

5th European Conference, ECSA 2011
Essen, Germany, September 13-16, 2011
Proceedings

Springer

Volume Editors

Ivica Crnkovic
Mälardalen University, IDT
P.O. Box 883, 721 23 Västerås, Sweden
E-mail: ivica.crnkovic@mdh.se

Volker Gruhn
University of Duisburg-Essen
paluno - The Ruhr Institute for Software Technology
Gerlingstraße 16, 45127 Essen, Germany
E-mail: volker.gruhn@paluno.uni-due.de

Matthias Book
University of Duisburg-Essen
paluno - The Ruhr Institute for Software Technology
Gerlingstraße 16, 45127 Essen, Germany
E-mail: matthias.book@paluno.uni-due.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-23797-3 e-ISBN 978-3-642-23798-0
DOI 10.1007/978-3-642-23798-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011935337

CR Subject Classification (1998): D.2, D.3, F.3, H.4, C.2, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The 5th European Conference on Software Architecture (ECSA 2011) provided researchers and practitioners with a unique platform to present and discuss the most recent, innovative, and significant advances, findings and experiences in the field of software architecture research and practice. This edition of ECSA built upon a history of a successful series of European workshops on software architecture held from 2004 through 2006, and a series of European software architecture conferences from 2007 through 2010.

We received more than 90 submissions in the three main categories: full research and experience papers (60 papers), emerging research papers (30 papers), and research challenge papers (4 papers). The conference attracted papers co-authored by researchers, practitioners, and academics from 32 countries. Each paper, independently of the category, was peer-reviewed by at least three reviewers, and discussed by the Program Committee. Based on the recommendations and the discussions, we accepted 13 full papers out of 60 full papers submitted. The acceptance rate for the full papers is 21.37%. In the Emerging Research category, we accepted a total of 24 papers, 8 of which were originally submitted in this category, and 16 were submitted as full papers. Finally, we accepted 7 papers as Research Challenge (Poster) papers.

In addition to the technical program consisting of academic and industrial keynote talks, a main research track, and a poster session, the scope of ECSA 2011 was broadened by two workshops on related topics: the Workshop on Traceability, Dependencies and Software Architecture (TDSA 2011), and the First International Workshop on Software Architecture Variability (SAVA 2011).

It was a great pleasure to have six eminent keynote speakers at ECSA 2011, three from academia and three from industry. On the academic side, Albrecht Schmidt from the Institute for Visualisation and Interactive Systems (VIS) at the University of Stuttgart presented trends and visions of interactive ubiquitous computing systems; Harald Gall, Director of the Software Evolution and Architecture Lab (s.e.a.l.) at University of Zurich, talked about software analysis as a service; and Raffaela Mirandola from the Dipartimento di Elettronica e Informazione (DEI) at Politecnico di Milano discussed software performance engineering in and for dynamic environments. On the industrial side, Eberhard Wolff, Architecture & Technology Manager at adesso AG, pondered what it really means to be an architect; Jörg Koletzki, Member of the Management Board of E.ON IT GmbH, talked about enterprise architecture management; and Magnus Larsson, Software Manager at ABB Corporate Research, shared insights on balancing long-term research and industrial applicability.

We are grateful to the members of the Program Committee for helping us to seek submissions and provide valuable and timely reviews. Their efforts enabled us to put together a high-quality technical program for ECSA 2011.

We are indebted to the local arrangements team at The Ruhr Institute for Software Technology (paluno) for the successful organization of all conference, social and co-located events. We also thank Workshops Chair Wilhelm Hasselbring, who made a significant contribution to the success of an extended version of ECSA. The ECSA 2011 submission, review and proceedings process was extensively supported by the EasyChair Conference Management System. We also acknowledge the prompt and professional support from Springer, who published these proceedings in printed and electronic volumes as part of the *Lecture Notes in Computer Science* series. Finally, we would like to thank our platinum sponsors adesso AG and E.ON IT GmbH for their generous support of this conference.

Most importantly, we would like to thank all authors and participants of ECSA 2011 for their insightful works and discussions!

July 2011

Volker Gruhn, General Chair
Ivica Crnkovic, Program Chair
Matthias Book, Proceedings Editor

Organization

Program Committee

Muhammad Ali Babar	IT University of Copenhagen, Denmark
Jesper Andersson	University of Växjö, Sweden
Paris Avgeriou	University of Groningen, The Netherlands
Thais Batista	Federal University of Rio Grande do Norte, Brazil
Steffen Becker	University of Paderborn, Germany
Tomas Bures	Charles University in Prague, Czech Republic
Ivica Crnkovic	Mälardalen University, Sweden
Carlos Cuesta	Rey Juan Carlos University, Spain
Paulo De Figueiredo Pires	Federal University of Rio Grande do Norte, Brazil
Rogerio De Lemos	University of Kent, UK
Khalil Drira	LAAS-CNRS, France
Laurence Duchien	INRIA - University of Lille, France
Roberto E. Lopez-Herrejon	Johannes Kepler University, Austria
Katrina Falkner	University of Adelaide, Australia
Ian Gorton	PNNL, USA
Darko Huljenic	Ericsson Nikola Tesla, Croatia
Rick Kazman	Carnegie Mellon University, University of Hawaii, USA
Gerald Kononya	Lancaster University, UK
Kai Koskimies	Tampere University of Technology, Finland
Philippe Kruchten	University of British Columbia, Canada
Patricia Lago	VU University Amsterdam, The Netherlands
Nicole Levy	Cédric Laboratory, CNAM, France
Grace Lewis	Carnegie Mellon University, USA
Anna Liu	NICTA/UNSW, Australia
Sam Malek	George Mason University, USA
Raffaela Mirandola	Politecnico di Milano, Italy
Henry Muccini	University of L'Aquila, Italy
Robert Nord	Carnegie Mellon University, USA
Flavio Oquendo	European University of Brittany - UBS/VALORIA, France
Mourad Oussalah	University of Nantes, France
Eila Ovaska	VTT, Finland
Claus Pahl	Dublin City University, Ireland
George Papadopoulos	University of Cyprus
Hongyu Pei-Breivold	ABB Corporate Research, Sweden

Jennifer Perez Benedi	Technical University of Madrid (UPM), Spain
T.V. Prabhakar	Indian Institute of Technology Kanpur, India
Claudia Raibulet	University of Milano-Bicocca, Italy
Alexander Romanovsky	Newcastle University, UK
Cecilia Rubira	Unicamp, Brazil
Eduardo Santana de Almeida	Federal University of Bahia and RiSE, Brazil
Juha Savolainen	Nokia, Finland
Bradley Schmerl	Carnegie Mellon University, USA
Clemens Schäfer	it factum GmbH, Germany
Bedir Tekinerdoğan	Bilkent University, Turkey
Christelle Urtado	LGI2P / Ecole des Mines d'Alès, France
Danny Weyns	Katholieke Universiteit Leuven, Belgium
Semih Çetin	Cybersoft Information Technologies, Turkey

Additional Reviewers

Achilleos, Achilleas
Agrawal, Ashish
Amirat, Abdelkrim
Aoussat, Fadila
Ben Alaya, Mahdi
Ben Halima, Riadh
Bouassida, Ismail
Brenner, Christian
Brondum, John
Delicato, Flavia
Desnos, Nicolas
Feugas, Alexandre
Galster, Matthias
Garbajosa, Juan
Grondin, Guillaume
Guabtni, Adnene
Hannachi, Mohamed-Amine
Hock-Koon, Anthony
Holtmann, Jörg
Iliasov, Alexei
Kakousis, Konstantinos
Kamoun, Aymen
Kapitsaki, Georgia
Keznikl, Jaroslav
Le Goaer, Olivier
Leite, Jair
Liu, Jenny
Liu, Yan
Loiret, Frédéric
Malavolta, Ivano
Malohlava, Michal
Merle, Philippe
Michalik, Bartosz
Mosser, Sébastien
Parra, Carlos
Paspallis, Nearchos
Payne, Richard
Pop, Tomas
Priesterjahn, Claudia
Ramanathan, Sakkaravarthi
Razavian, Maryam
Romay, M. Pilar
Simko, Viliam
Stol, Klaas-Jan
Suleiman, Basem
Tamburri, Damian Andrew
Tamura, Gabriel
Tang, Antony
Tell, Paolo
Tibermacine, Chouki
Tofan, Dan
Travkin, Dietrich
V, Kiran Kumar
Vajja, Kiran Kumar
Van Heesch, Uwe
Vauttier, Sylvain
Venkatasubramanian, Smrithi Rekha
Von Detten, Markus

Sponsors

UNIVERSITÄT
DUISBURG
ESSEN

PALUNO
The Ruhr Institute for Software Technology

adesso
business.
people.
technology.

Table of Contents

Requirements and Software Architectures

Software Architecture, Components, and Compositions

Quality Attributes and Software Architectures

Software Product Line Architectures

Architectural Models, Patterns and Styles

Short Papers

Process and Management of Architectural Decisions

Software Architecture Run-Time Aspects

ADLs and Metamodels

Services and Software Architectures

Supervising the Evolution of Web Service Orchestrations Using Quality Requirements

Chouki Tibermacine[1] and Tarek Zernadji[2]

[1] LIRMM, CNRS and Montpellier-II University, France
[2] Computer Science Department, University of Biskra, Algeria
Chouki.Tibermacine@lirmm.fr, zernadji@yahoo.fr

Abstract. Since many years, Web services have confirmed their status of one of the most pertinent solutions for a given service provider, like Google, Amazon or FedEx, to open its solutions for third party software development. New business logic can be implemented through orchestrations of existing Web services. This helps development teams in capitalizing resources held by the providers of these services. Nonetheless, these service-oriented software architectures, like any other software artifact, are subject to changes during their lifecycle, and thus can undergo an evolution phenomenon. In this phenomenon, it is argued that quality can be weakened after successive changes (Lehman's 7th law of software evolution), and this is mainly due to the lack of architecture documentation and tool support to supervise architecture changes. In this paper, we present an approach to supervise the evolution of Web service orchestrations, with quality requirements considered as a support documentation. First, we show how important design decisions, like the choice of a service-oriented architecture pattern can be formalized as a documentation for the quality they implement. Then, we detail how this documentation can be used to supervise architecture changes. In this way, the impact of changes made on a software architecture are analyzed on-the-fly to determine which quality is affected.

1 Introduction: Context and Motivation

Building distributed software by orchestrating existing Web services is a new paradigm, which has been proposed as a possible implementation for the service-oriented architecture specification. It has been greatly influenced by the well-known business process engineering field, where processes can be designed as collaborations between a set of services published by some providers. New business logic can thus be implemented, as an extension of existing Web services, through these orchestrations. This helps development teams in capitalizing resources held by the providers of these services. Indeed, Web service providers, which hold some precious resources (like large databases of products to retail of Amazon, or weather forecast data of Meteo France), offer third party developers the opportunity (for free or not) to build new applications by extending their public services, and thus capitalize on these resources.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 1–16, 2011.

Nonetheless, these service-oriented software architectures, like any other software artifact, are subject to changes during their lifecycle, and thus can be affected by the consequences of an evolution phenomenon [14]. In this phenomenon, it is argued that quality can be weakened after successive changes (Lehman's 7th law of software evolution [14]). This is mainly due to: i) the lack of architecture documentation that can be used by developers to better understand the design decisions made on the system, and ii) the lack of tool support to supervise architecture changes.

In this paper, we present an approach to supervise the evolution of Web service orchestrations, with quality requirements considered as a support documentation. First, we show how important design decisions, like the choice of a service-oriented architecture pattern can be formalized as a documentation for the quality they implement. Then, we detail how this documentation can be used to supervise architecture changes. In this way, the impact of changes made on a software architecture are analyzed on-the-fly to determine which quality is affected.

In the following section, we show an example that illustrates the problems which we tackle in this paper. In Section 3, we expose the overall approach that we propose in this paper to solve the identified problems. Then, in Section 4, we detail how the quality documentation is specified in our approach. This documentation is used by an evolution assistance algorithm, which is described in Section 5. Section 6 illustrates the use of our approach through an example. Before concluding this paper, we present in Section 7 the related work.

2 Illustrative Example

In this section we show, through an example of a Web service orchestration – a BPEL (Business Process Execution Language [17]) process, how some evolution scenarios can have consequences on quality requirements. Let us suppose that we have an Appealed Assessments System [9] built using Web services. This system is responsible for managing and enforcing policies that pertain to private companies involved with the forestry and lumber trade. Briefly, this service is dedicated to producing a range of reports related to already assessed claims that have been successfully or unsuccessfully appealed.

The decisions made by the architects to design this service-oriented architecture involved the use of a `Data Controler` service. This one provides all the logic required to fulfill the capabilities of the assessment reports service. It allows access up to six different repositories in order to gather all of the required data depending on the requested reports.

Furthermore, the architects observe that they may have to access additional databases in the future and therefore the `Data Controler` service have to undergo some changes. This results in a portability problem in the service architecture design. Consequently, the architects decide to design a facade service (architecture decision AD1) using the service facade pattern [9], to ensure the portability requirement (quality attribute QA1) into the service architecture.

The facade service is named `Data Relayer`, and its role is to receive service consumer requests, relay them to the `Data Controler` service, and then relay the responses back to the service consumer. The facade service ensures the adaptation between the message format used by the Appealed Assessments Service and the data format managed by the `Data Controler` service. Also, it validates the reports received from the `Data Controler` service. The facade service decouples then the consumers of the Appealed Assements Service from the changes that may occur on the `Data Controler`, and compensates its behavior modifications so that the consumers are not impacted.

Preventing from unauthorised access to the ressources of the Appealed Assessments service, the architects decide to establish a service account (based on the *Trusted Subsystem Pattern* [9] (AD2)) to secure the service from direct access to the databases. The security requirement (QA2) is thus defined and implemented inside the whole service orchestration.

Let us assume now, that the maintenance team receives two changes requests. The first concerns the performance enhancement of the overall service (Appealed Assessments Service). So, the developer team proceeds by short-circuiting the authentication service account in the orchestration (removing simply the invoke activity). The architect performing this change, ignore that the usefulness of that service was also (as imposed by the Trusted Subsystem pattern) to prevent direct access to the service resources from malicious attackers. Therefore, this change breaks AD2 and thus the quality requirement that it implements (QA2).

Over the years, the company providing these services has significantly expanded, and consequently more users requested the Appealed Assessment Service. In such a highly concurrent environment, the service may manage a large amount of data and thus increasing resources consumption which may compromise the overall service performance and availability. The architects realize that the service-oriented architecture has to adapt to this scalability requirement problem keeping the performance of the service unaffected. Hence, they examine different kinds of data used by the service and find out that, the policy data remains frequently unchanged during the working days. Therefore, the architects decide (AD3) to use the Partial *State Deferral Pattern* [9] to temporarily hold a copy of data on a local database server increasing the performance of the service (QA3). This scenario implies some architectural changes: first, invoking directly the Data Controler after an authentication through the service account service; second, designing a new standardized contract for an archival service that takes the responsibility of populating the data in the database server; finally, for performance optimization purpose, moving the functionality of the Data Relayer service into the Archival service which results in removing the later from the architecture. This breaks the facade design pattern (AD1) and causes the loose of the portability quality requirement.

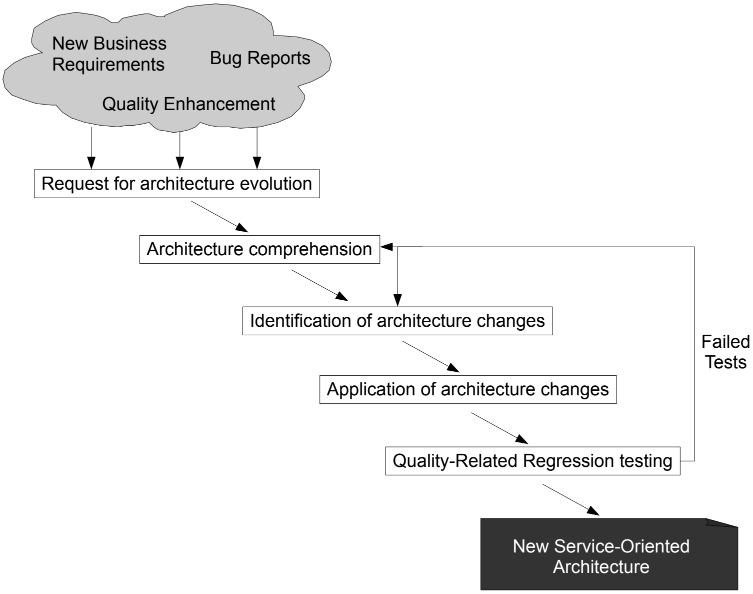

New Business Requirements
Bug Reports
Quality Enhancement
Request for architecture evolution
Architecture comprehension
Identification of architecture changes
Application of architecture changes
Quality-Related Regression testing
Failed Tests
New Service-Oriented Architecture

Fig. 1. A Micro-Process of Architecture Evolution

3 Proposed Approach

Figure 1 shows a simple micro-process of service-oriented architecture evolution[1]. In this process, the triggers for requesting architecture evolution can be either new business requirements (for perfective evolution), bug reports (for corrective evolution) or quality enhancement (for perfective, adaptive or preventive evolution). Then the developer has to go through multiple steps, ranging from architecture comprehension to the proposition of a new architecture.

Among these steps, the developer performs some testing to check if there is a "clean" progression (verify if the additional services, operations or activities work correctly) and no regression (existing features are not negatively impacted by the additions). In this paper, we address exclusively quality-related regression testing. In practice there are few works that dealt with this aspect by proposing some automatic support. Even with the existence of such approaches, if some tests fail, the developer iterates (eventually many times) to fix the problems. She/He is asked to look for the architecture changes to be applied, and sometimes she/he is led to the step of "Architecture comprehension".

The proposed approach aims at assisting this process by notifying the developer on-the-fly if there are some architecture changes that affect quality requirements. This is illustrated in Figure 2.

[1] This micro-process addresses software evolution in general, and not service-oriented architectures in particular. Adaptations to this specific context are detailed later.

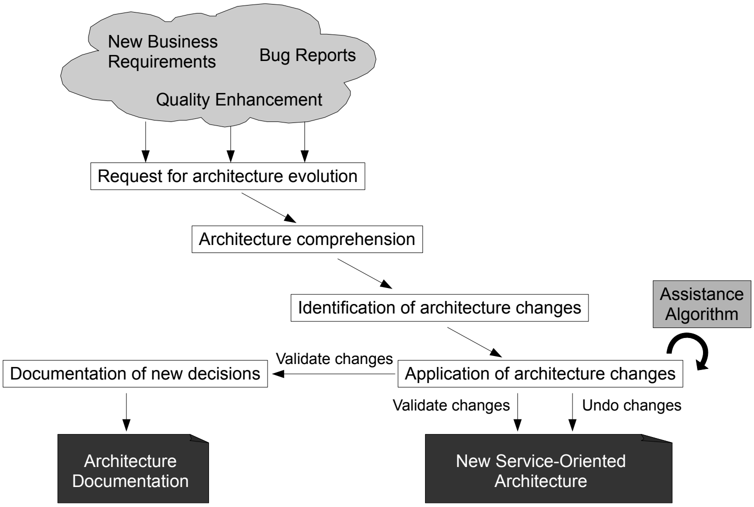

New Business Requirements
Bug Reports
Quality Enhancement
Request for architecture evolution
Architecture comprehension
Identification of architecture changes
Assistance Algorithm
Documentation of new decisions
Validate changes
Application of architecture changes
Validate changes
Undo changes
Architecture Documentation
New Service-Oriented Architecture

Fig. 2. The Proposed Micro-Process of Architecture Evolution

The approach introduces two concepts: an architecture documentation (bottom left of Figure 2) and an assistance algorithm. The assistance algorithm is used when developers apply changes on an architecture to notify them with the possible impact of their changes on quality requirements. Then, it is the developer's responsibility to validate or undo changes. If changes are validated the developer is asked to document the new decisions taken while evolving the service-oriented architecture. These two concepts are detailed in the following two sections.

4 Architecture Decision Documentation

The concept of architecture decision documentation has been firstly introduced in [19]. In this paper, we present an improvement to the old version of this documentation. It defines in a formal way the links between architecture decisions and quality attributes implemented by these decisions. We consider thus architecture decisions, which are entities that can be formalized, as a way to indirectly check automatically quality requirements, which are properties that cannot generally be formalized directly (or are very difficult to formalize[2]).

An architecture decision documentation abstracts the links between a given quality attribute and an architecture decision associated to this attribute. Figure 3 shows how these links are organized. We associate to a link a degree

[2] By "formalization", we simply mean here the specification of a given artifact in an unambiguous and structured or semi-structured way using a language that can be processed by tools (not using the natural language).

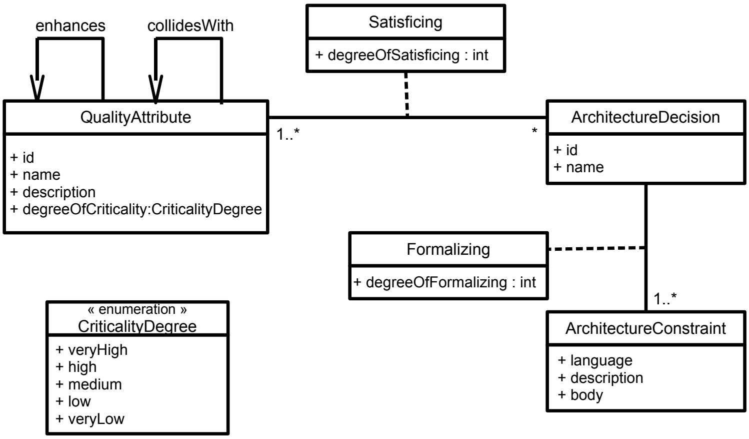


Fig. 3. Links between Architecture Decisions and Quality Attributes

of satisfaction. An architecture decision in collaboration with other decisions contribute to the satisfaction of a given quality attribute. Each degree of satisfaction represents a percentage. In the ideal situation (where the developers are confident in the pertinence of their design decisions), the sum of all degrees associated to the same quality attribute (within the same architectural element) would be equal to 100%. For example, a portability quality attribute can be concretized by three different architecture decisions: the choice of the facade service pattern [9][3], the choice of the MVC pattern [3] and the use of an API. If the developers consider that the two first decisions contribute more, in the concretization of the portability quality attribute, than the third one, because they are critical, they can associate to them high scores (for example 40 % to each decision) and the last architecture decision a lower score (20 % for example). This is done in the same manner as in software requirements engineering where the project manager assigns values like high, medium or low for the technical difficulty of the realization of each requirement or for their functional priority. In our case, we chose to give them numerical values voluntarily because of the complementarity which exists between architecture decisions to reach a quality goal, as illustrated in the example above.

We voluntarily simplify, in this documentation, the specification of architecture decisions. An architecture decision is thus formalized by an architecture constraint (see the "Related Work" section for richer specifications of architecture decisions). Here again, a formalization degree is a percentage associated to the link between an architecture decision and an architecture constraint. This score represents the extent to which the constraint formalizes the design decision. If we consider that several constraints formalize the same architecture decision, it is possible for the developer to state how the different constraints share the

[3] This pattern is originally inspired from [10].

formalization of the design decision. In some cases, a given constraint may have a degree of formalization more important than others. In the ideal situation (where the developers are sure of the completeness of their formalization), the sum of all degrees associated to the same architecture decision would be equal to 100%. The constraints written in a given documentation are defined with a predefined constraint language.

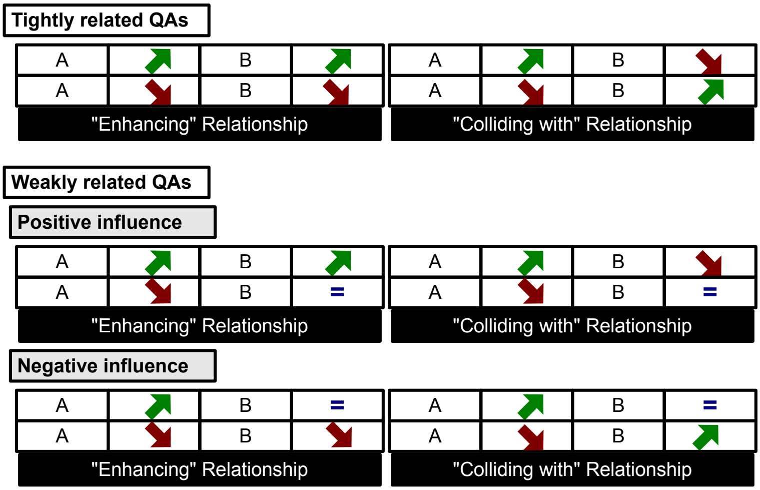
A quality attribute in this documentation is a non-functional property representing an ISO 9126[4] characteristic or sub-characteristic (Reliability, Maintainability, Portability, ...). It has a degree of criticality (inspired from Kazman's quality attribute scores and Clements' quality attribute priorities [7]) which is specified by developers and represents the importance of this quality attribute in the architecture. Its possible values are: very high, high, medium, low and very low.

Associated to a given architecture decision, a quality attribute can enhance (affect positively) other quality attributes. For example, the choice of the pipeline architecture style targets the maintainability quality attribute, which enhances in this case the efficiency attribute of the system. Contrarily, a given quality attribute can collide with (affect negatively) other quality attributes. For example, the security quality attribute collides generally with the efficiency attribute. This depends of course on the documented architecture decision and the application context. It is on the responsibility of developers, fully aware of the application's context and the architecture decisions they made, to document these optional parts (the other quality attributes that collide-with or enhance the documented quality attribute) of an architecture decision documentation.

A given quality attribute can be tightly- or weakly-coupled to another one. In the first case, if a quality attribute A affects positively another attribute B, if we enhance A, B will B enhanced; and if A is weakened, B will be weakened too. In the second case (weakly-coupled attributes), if A affects positively another attribute B, if we enhance A, B will be enhanced; and if A is weakened, B will not be affected. Inversely, the same thing can be considered, if A affects negatively B. This is illustrated in Figure 4.

For example, in a service orchestration, adding an invocation to an encryption service before transmitting information to a remote server is a simple architecture decision taken to enhance the security quality attribute. This makes less efficient the whole orchestration (affects negatively the efficiency attribute). If we decide in another context, to remove a binding to an authentication service which is invoked before a given business service, this will obviously affect positively the efficiency quality attribute (there is less time to execute the business service). We conclude here that the two quality attributes, in the two contexts, are tightly coupled.

[4] Software engineering – Product quality – Part 1: Quality model. The International Organization for Standardization Website: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749



Tightly related QAs

A	↗	B	↗	A	↗	B	↘
A	↘	B	↘	A	↘	B	↗
"Enhancing" Relationship				"Colliding with" Relationship			

Weakly related QAs

Positive influence

A	↗	B	↗	A	↗	B	↘
A	↘	B	=	A	↘	B	=
"Enhancing" Relationship				"Colliding with" Relationship			

Negative influence

A	↗	B	=	A	↗	B	=
A	↘	B	↘	A	↘	B	↗
"Enhancing" Relationship				"Colliding with" Relationship			

Fig. 4. Relationships between Quality Attributes

In another illustrative example, designing a system using the facade service design pattern aiming to enhance its portability affects negatively the reliability quality characteristic (more precisely, the availability sub-characteristic). Indeed, in the presence of a single service providing the business service to clients, if this service crashes, the provided functionality will not be anymore available. Let us suppose now that a given service is provided by a component within a web application in order to abstract details of the different Internet browsers in which the application is executed at the client side (portability purpose). The removal of such a service will not affect in any way the reliability attribute. This is an example of two quality attributes which are weakly coupled.

Between weakly coupled quality attributes, we identified two kinds of relationships. There can be a positive or a negative influence. In the first case (positive influence), it is the enhancement of the first attribute which has an influence on the second one; however in the other case, it is its weakness which produces an effect on the second attribute. This is shown on Figure 4.

In the current implementation of architecture decision documentation, architecture constraints are specified using a modified version of OMG's OCL [18]. An architecture constraint in this language navigates in a metamodel of BPEL Web service orchestrations, but apply to only one instance of that metamodel (a model which represents a BPEL process). The evaluation of a given constraint tells the developer whether the architecture description conforms to the constraint or not.

In addition to architecture decision documentation, we propose (as an optional feature) to build a catalog of quality attribute relationships. Designing such a catalog consists in:

1. Identifying the quality attributes defined in the quality model of the company
2. Identifying the attributes defined in the quality plan of the software project

3. Building a bi-dimensional table with all the quality attributes (one per line and one per column)
4. Completing progressively the correlation between the quality attributes (on the basis of information gathered from previous projects and the experience of developers)
5. Each time, adapting the table to the service-oriented architecture context

Once this table validated by the project manager, the assistance algorithm can exploit it in accompanying developers in the architecture change step.

5 Change Assistance Algorithm

During architecture change, the information encapsulated in the architecture documentation is exploited by an assistance algorithm in order to assist developers. The main purpose is to drive software architecture evolution to a situation where the initially required quality is minimally affected. This algorithm is presented throughout this section as several functions.

```
(01)  algorithm ArchitectureChangeAssistance {
(02)    let AE := Architectural Element
(03)    and AD := Architectural Decision
(04)    and AC := Architectural Constraint
(05)    and QA := Quality Attribute
(06)    and AT := Architecture Tactic //a couple composed of a QA and an AD
(07)    and Doc := architecture documentation associated to changed AE
(08)    and affectedQAs := { } //an empty set
(09)    function main() {
(10)      on RequestForAssistance {// an event listener
(11)        for-each (AT in Doc) {
(12)          QA := QA in AT
(13)          AD := AD in AT
(14)          checkArchitecturalConstraint (AD)
(15)        }
(16)        let newAD := ask for AD associated to the new architecture, if any
(17)        if(newAD != null) let newQA := ask for the QA associated to newAD
(18)        addNewArchitecturalTactic (newAD,newQA)
(19)      }
(20)      checkAffectedQAs ()
(21)    }
(22)  }
```

During the step of architecture change application (Figure 2) the developer can ask for an assistance. This triggers the listener on Line 10 in the algorithm above. The algorithm starts first by looking for the architecture documentation associated to the architectural element (the orchestration or the Web service description) which has been changed. Then, the algorithm checks each constraint in the documentation (by calling a function which is detailed in the following paragraphs). After that, the developer is asked to pinpoint the architecture decision and the quality attribute associated to the changes, if any (Lines 16 and 17).

At last, if the changes generate a new architecture decision, the algorithm try to add, to the documentation, the couple composed of this new decision associated to its quality attribute, which is called in this work an architectural tactic (Line 18). In addition the algorithm tries to infer the quality attributes affected by this new tactic (Line 20).

The function detailed in the listing below, checks the constraints associated to a given architecture decision received as an argument. It starts by checking the constraint expressions associated to the decision. If the checking does not succeed for a given constraint, a set of warnings are displayed to the developer. The displayed information includes the architecture decision, the precise architectural element impacted by the change, the degree of formalization of the decision, the quality attribute, its degree of satisficing and its criticality degree (Lines 11 to 14 in the algorithm below). In addition this function shows to the developer the list of quality attributes which are eventually impacted by this change (Lines 15 and 16). For doing so, it uses the table of relationships between quality attributes presented in the previous section. It limits the selected quality attributes to the ones which are tightly coupled with an "enhacing" relationship. This ensures the selection of the most pertinent quality attributes in this situation.

```
(01)  function checkArchitecturalConstraint (AD) {
(02)    for-each(AC associated to AD)
(03)      let result := check AC
(04)      if (result == false) {
(05)        AE := AE in the context of AC
(06)        QA := QA associated to AD
(07)        warn "The following architecture decision " +AD+" is affected."
(08)        warn "This concerns the architectural element: "+AE
(09)        warn "The affected architecture decision is formalized
(10)        by the constraint up to " + degreeOfFormalization (AD,AC)+ "%"
(11)        warn "The affected architecture decision is satisficing "+QA
(12)        + " up to " +degreeOfSatisficing(AD,QA)+"%"
(13)        warn "The degree of criticality of this QA is: "
(14)        + degreeOfCriticality(QA)
(15)        warn "Other QAs may be affected. This concerns: " +
(16)        QA_Relationships (QA,"enhances", "tight")
(17)        ask to validate the new architecture or undo changes
(18)        according to the warnings above
(19)        if(new architecture maintained) {
(20)          affectedQAs := affectedQAs + QA
(21)          + QA_Relationships(QA, "enhances", "tight")
(22)          warn "Architecture documentation will be changed ..."
(23)          Doc := Doc - AT(AD,QA)
(24)          ask to review satisficing degrees of ATs related to
(25)          QA_Relationships(QA, "enhances", "tight")
(26)          ask to review Non-Functional Requirements specification
(27)        }
(28)      }
(29)    }
(30)  }
```

Then, the developer is asked to validate the new architecture fully aware with the possible consequences of her/his changes, or to undo changes. In this last case, the architecture documentation should be updated by the algorithm (this is the second important role of this assistance algorithm). The affected decisions and their associated quality attributes are removed from the documentation (Line 23 in the algorithm above). The developer is at last asked to review the degrees in the documentation, as some tactics are removed. In addition, she/he is invited to review the non-functional (or quality) requirements specification.

The function `addNewArchitecturalTactic(...)` creates a new architectural tactic and adds it to the documentation. Before that, if the quality attribute has been voluntarily added by the developer, it is removed from the set of affected quality attributes (Line 04 in the listing below). Else this attribute is considered as a new quality and a checking is performed to alert the developer of the other qualities that are possibly affected by this attribute (Lines 06 – 07). At last, the algorithm asks the developer to change the quality requirements specification.

```
(01)  function addNewArchitecturalTactic (AD,QA) {
(02)    newAT := new AT(AD,QA)
(03)    if (QA is in affectedQAs)
(04)      affectedQAs := affectedQAs - QA
(05)    else {
(06)      warn "Other QAs may be in conflict with "+QA+": "
(07)      + QA_Relationships (QA,"collidesWith","both")
(08)    }
(09)    warn "Architecture documentation will be changed ..."
(10)    Doc := Doc + newAT
(11)    ask to change Non-Functional Requirements specification
(12)  }
```

The last function (see below) just recalls to the developer that there still remain some affected quality attributes, if any. The developer is asked to review the architecture documentation and the quality requirements specification.

```
function checkAffectedQAs () {
  if (affectedQAs <> {} ) {
    for-each (QA in affectedQAs) {
      warn QA + "is still affected by your changes"
      ask to review satisficing degrees of ATs implying QA
    }
    ask to change Non-Functional Requirements specification
  }
}
```

The overall goal of this algorithm is twofold. First, it assists developers during architecture evolution with information about the impact of their changes on architecture design decisions and on quality attributes. Second, it helps to maintain the documentation of non-functional (or quality) requirements up-to-date in a semi-automatic fashion. This can be observed in updates made automatically on the documentation, requests to review satisficing degrees of the affected quality attributes, and requests to change or review NFRs specification.

6 The Proposed Approach in Practice

In this section, we show an example of an architecture documentation and its use by the evolution assistance algorithm. Let us take the example of Section 2. Its architecture documentation is presented in a synthetic way (in order to not be too verbose with its original XML-based description) in the listing below:

```
Architecture-Documentation :
1. Architecture-Tactic :
    This tactic guarantees the Portability quality requirement by using
    a Service facade pattern
    - Quality-Attribute name="Portability" degreeOfCriticality="high"
    - Related-Quality name="Performance" relationship="CollidesWith"
                               relationType="tight"
    - Architecture-Decision name="Service facade pattern"
                                        degreeOfSatisficing="90"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="80"

2. Architecture-Tactic :
    This tactic ensures the Security quality requirement by using
    a Trusted subsystem pattern
    - Quality-Attribute name="Security" degreeOfCriticality="very high"
    - Related-Quality name="Availability" relationship="Enhances"
                               relationType="weak" influence="negative"
    - Architecture-Decision name="Trusted subsystem pattern"
                                         degreeOfSatisficing="70"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="90"

3. Architecture-Tactic :
    This tactic ensures the Performance quality requirement by using
    a Partial state deferral pattern
    - Quality-Attribute name="Performance" degreeOfCriticality="high"
    - Related-Quality name="Security" relationship="CollidesWith"
                                relationType="tight"
    - Architecture-Decision name="Trusted subsystem pattern"
                                         degreeOfSatisficing="60"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="70"
```

The architecture documentation contains three architectural tactics[5]. They document the links between architectural decisions (AD1, AD2, AD3) presented in Section 2 and their corresponding quality attributes (QA1, QA2, QA3). In this documentation we can see among others the different relations between

[5] We recall that a tactic is the couple composed of an architecture decision and its quality attribute.

quality attributes (`Related-Quality` element in the listing above). For example, in the first tactic, the `Related-Quality` element shows that the portability and performance quality attributes are colliding and are tightly coupled.

Let us see now the use of the assistance algorithm, given the evolution scenario described in the example of Section 2: short-circuiting the authentication service. The assistance algorithm checks the constraints[6] formalizing the architectural decisions for each architectural tactic, and detects that the constraint formalizing the decision AD2 (The Authentication service implementing the Trusted subsystem pattern) was violated. Therefore, the security quality attribute (QA2) is affected. It then notifies the developer that the violated constraint formalizes AD2 up to 90% (an important constraint in the formalization of this decision), the affected architecture decision satisfies Security up to 70% with a very high degree of criticality, and that the security is weakly coupled to the availability quality attribute by an enhancement relation with a negative effect. This means that the availability quality attribute is directly affected by this change. Based on this notification, the developer decides to abort the change she/he made, aware that short-circuiting the authentication service makes the service not secured.

The second change to the orchestration consists of adding a new functionality service namely the Archival service which leads to remove the Data Relayer service. This change aims to improve the performance and the availability of the service. The algorithm detects that the constraint representing the decision AD1 (service facade pattern) does not hold anymore, which means that the portability quality attribute (QA1) is affected. Hence, it informs the developer that the violated constraint formalizes AD1 with a degree of 80%, satisfies QA1 up to 90% which has a high degree of criticality, and that QA1 and QA3 (performance) are colliding but tightly coupled. The developer concludes that if she/he wants to improve the performance of the service she/he will probably affect the portability quality attribute, and therefore, decides that performance is more important than portability and validates the change. The corresponding tactic of the affected decision is removed from the documentation and a new tactic is added. The developer is invited to update the NFRs specification and to review the satisficing degrees for the affected qualities. This information serves for possible changes that may occur on the service architecture in the future.

7 Related Work

In the literature, there are many works on the documentation of architeure design decisions. Clements et al. in [6] present an approach which provides a framework for documenting different views of a software architecture. The authors propose a template for architecture description encompassing the documentation of architectural decisions. In [20], Tyree and Akerman discuss the importance of documenting architecture decisions and their specification as first-class entities in an architecture description. They present a template specifically designed for

[6] For reasons of space limitations, constraints are not presented here. They are defined using a modified version of OCL and navigate in a metamodel of BPEL.

architecture decision documentation, which embeds interesting information characterizing architecture design decisions (`status`, `assumptions`, `implications`, `related artifacts`, `constraints`, ...). Philippe Kruchten introduced a taxonomy of design decisions [13]. He presents a model for describing architecture decisions, including `rationale`, `scope`, `state`, `history of changes`, `categories`, `cost` and `risk`. He identifies in this ontlogy the different possible relationships between design decisions and links between design decisions and design artifacts. In [11], Jansen and Bosch present a new way of building software architectures. They propose to define these design models as a composition of architecture design decisions. The authors introduce a model for architecture design decisions, including a `description`, the `rationale`, the `design rules`, the `design constraints`, the `consequences`, the `pros` and `cons`. In [4], the authors proposed a way to characterize architectural decisions. They defined attributes to describe architectural decisions by separating mandatory and optional attributes according to their degree of importance. The first class introduces information associated to architectural decisions that must be defined throughout the system life cycle, including a `decision name and description`, the `constraints`, the `dependencies`, the `status`, the `rationale`, the `design patterns`, the `architectural solution`, and the `requirements`. The second class provides additional information that can be choosen according to user preferences such as, the `alternative decisions`, `assumptions`, `pros and cons`, `category of decision`, or `quality attributes`. In addition to these attributes, they have defined attributes to support the evolution of architectural decisions, including the `date and version`, the `obsolete decision`, the `validity`, the `reuse times and rating`, and the `trace links`.

As in these approaches, our work proposed a new way to document architecture decisions. But contrarily to these works, we focused on the use of this documentation in the architecture change assistance. These works can complement our proposed documentation in order to make it richer.

Many works have been proposed on Non-Functional Requirements capture and specification. One of the major works in the literature is that of Mylopoulos et al. [16]. Following a process-oriented approach the authors propose a framework for the representation and use of Non-functional requirements during the development process. The framework includes five components allowing, following a goal-oriented process, to justify and argue design choices made to satisfy certain software quality requirements. The authors consider non-functional requirements as `goals` to be achieved by validating the right design decisions and their rationale, considered in turn as goals. In [8] Cyneirios et al. propose an approach based on Mylopoulos's framework for capturing and representing NFRs and their interdependencies. Their approach shows the integration of NFRs in functional requirements models. The authors were interested in conceptual models expressed in UML by incorporating NFRs descriptions in class, sequence, and collaboration diagrams. Bass et al. [1], proposed ADD method (*Attribute-Driven Design*) that follows an architectural design process guided by quality requirements. It uses the concept of attribute primitives, which are collections of components and connectors collaborating to satisfy some quality attributes. These attributes are documented as general scenarios. In [2], the authors proposed architectural tactics, in the same spirit as the primitive attributes to guarantee quality characteristics in software architectural design. Kim et al. [12] presented an approach for representing NFRs in software architecture using architectural tactics as reusable architectural building blocks. The later and their relationship are represented as *Feature Models* and their semantics is defined with the RBML language (*Role-Based Metamodeling Language*). Architectural tactics satisfying quality attributes are selected and composed into one tactic encompassing all the desired qualities. The resulting tactic is then instantiated to create a software architecture that incorporates NFRs for the system under development. In [15], the authors present an approach inspired from [16,5]. It aims at integrating NFRs handling in analysis and design phases as with functional requirements to fill the gap between the elicitation and implementation of NFRs.

All these approaches focus on the design stage of software development. Our work is complementary to these approaches, since it addresses a stage which is situated downstream in the development process, the evolution stage.

8 Conclusion and Future Work

Since some years, architecture decision and design rationale are two topics which received a lot of attention from the software architecture research community. In this paper, we proposed an approach which provides: i) a language to document a basic form of architecture decisions as architecture constraints, and the rationale of these decisions, which are quality attributes, together with some fine-grained information about the relationships between these two concepts; ii) a method which makes operational this documentation through its use during architecture evolution; and iii) an algorithm which implements the supervision of architecture evolution. This supervision aims at deducing on-the-fly the possible impact of a given architectural change on design decisions and consequently identify the affected quality requirements.

Our approach has been applied on a specific kind of software architectures, which are service-oriented ones. A concrete implementation of this kind of software architectures has been considered in our work, which are Web service orchestrations.

On the conceptual aspect, we plan in the near future to separate functional from non-functional evolution in the assistance algorithm. Indeed, these two imply different considerations. Non-functional (or quality) evolution have direct impact on existing decisions and quality, and the developer has some knowledge about the existing quality requirements. In the case of functional evolution, the developer has a different profile, and should be assisted differently.

On the tool and experimental aspect, we will conduct the creation of a catalog of predefined service-oriented architecture design decisions in order to help the developers in the initial documentation of their architectures. This will be based on existing works on patterns or quality models for service-oriented architectures, among others.

References

1. Bass, L., Bachmann, F., Klein, M.: Quality attribute design primitives and the attribute driven design method. In: Proceedings of the 4th International Conference on Product Family Engineering, pp. 169–186. Springer, Heidelberg (2001)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester (1996)
4. Capilla, R., Nava, F., Duenas, J.C.: Modeling and documenting the evolution of architectural design decisions. In: Proceeding of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI 2007). IEEE Computer Society, Los Alamitos (2007)
5. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Dordrecht (1999)
6. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures, Views and Beyond. Addison-Wesley, Reading (2003)
7. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures, Methods and Case Studies. Addison-Wesley, Reading (2002)
8. Cysneiros, L.M., Sampaio do Prado Leite, J.C.: Nonfunctional requirements: From elicitation to conceptual models. IEEE TSE 30(5), 328–350 (2004)
9. Erl, T.: SOA Design Patterns. Prentice Hall, Englewood Cliffs (2009)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Sofware. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., Reading (1995)
11. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proceedings of of the 5th IEEE/IFIP WICSA 2005 (2005)
12. Kim, S., Kim, D.-K., Lu, L., Park, S.: Quality-driven architecture development using architectural tactics. Elsevier JSS, 82(8), 211–1231 (2009)
13. Kruchten, P.: An ontology of architectural design decisions in software intensive systems. In: Proceedings of the 2nd Groningen Workshop Software Variability, pp. 54–61 (2004)
14. Lehman, M., Ramil, J.F.: Software evolution. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering, 2nd edn. Wiley, Chichester (2002)
15. Marew, T., Lee, J.-S., Bae, D.-H.: Tactics based approach for integrating non-functional requirements in object-oriented analysis and design. Journal of Systems and Software 82(10), 1642–1656 (2009)
16. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using nonfunctional requirements: A process-oriented approach. IEEE TSE 18(6), 483–497 (1992)
17. OASIS. Web services business process execution language version 2.0. Website of the Organization for the Advancement of Structured Information Standards (2006), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html
18. OMG. Objectconstraint language specification, version 2.0, document formal/2006-05-01. Object Management Group Web Site (2006), http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf
19. Tibermacine, C., Fleurquin, R., Sadou, S.: Nfrs-aware architectural evolution of component-based software. In: Proceedings of the 20th IEEE/ACM ASE 2005, Long Beach, California, USA, pp. 388–391. ACM Press, New York (2005)
20. Tyree, J., Akerman, A.: Architecture decisions: Demystifying architecture. IEEE Software 22(2), 19–27 (2005)

Towards Systematic Integration of Quality Requirements into Software Architecture*

Azadeh Alebrahim, Denis Hatebur, and Maritta Heisel

University Duisburg-Essen, Germany
{azadeh.alebrahim,denis.hatebur,maritta.heisel}@uni-duisburg-essen.de

Abstract. We present a model- and pattern-based approach that allows software engineers to take quality requirements into account right from the beginning of the software development process. The approach comprises requirements analysis as well as the software architecture design, in which quality requirements are reflected explicitly.

1 Introduction

Taking quality (or non-functional) requirements into account when developing a software architecture is a demanding task, for which satisfactory solutions are still sought for. In this paper, we want to contribute to improve this situation. We present a model- and pattern-based approach for architectural design that explicitly takes quality requirements (in particular, security and performance requirements) into account.

As a basis for requirements analysis, we use Jackson's problem frame approach [6]. We have carried over problem frames to UML [11] by defining a specific UML profile, and we have implemented a tool, called *UML4PF*[1] supporting requirements analysis and architectural design based on problem frames [4]. As a basis for architectural design, we use an method that we developed for deriving architectures based on functional requirements [2].

In the present paper, we extend our previous requirements analysis and architectural design methods by explicitly taking into account quality requirements. The analysis documents are extended by quality requirements that complement functional ones. For this purpose, we have extended the UML profile [5]. The so enhanced problem descriptions form the starting point for architectural design. To design the architecture, we apply appropriate security or performance patterns and mechanisms and define quality stereotypes that serve as hints for implementers.

The rest of the paper is organized as follows. We present the basics on which our approach builds in Sect. 2, namely problem frames and security and performance patterns and mechanisms. In Sect. 3, we present the UML profile we defined to carry over the problem frame approach to UML. Section 4 is devoted to describing our approach in more detail. Related work is discussed in Sect. 5, and conclusions and future work are given in Sect. 6.

[*] Part of this work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft - DFG) under grant number HE3322/4-1.

[1] Available under http://swe.uni-duisburg-essen.de/en/research/tool/

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 17–25, 2011.

2 Basic Concepts

In this section, we introduce the basic concepts our approach relies on.

2.1 Requirements Description Using Problem Frames

Problem frames are patterns to describe software development problems. They were proposed by Michael Jackson [6]. A problem frame basically consists of *domains*, *interfaces* between them, and a *requirement*. The task is to construct a *machine* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements.

Software development with problem frames proceeds as follows: first the environment in which the machine will operate is represented by a *context diagram*. A context diagram consists of machines, domains and interfaces. Then, the problem is decomposed into subproblems, which are represented by *problem diagrams* that can be instances of problem frames. A problem diagram consists of a submachine of the machine given in the context diagram, the relevant domains, the interfaces between these domains, and a requirement. Figure 1 shows a problem diagram in UML notation.

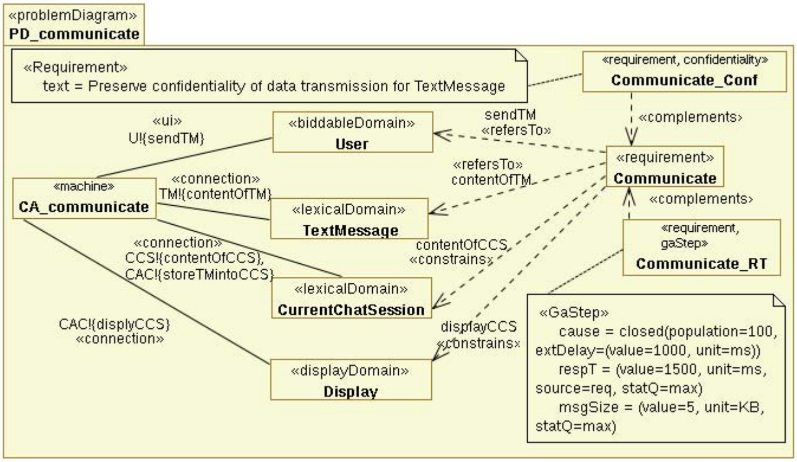


Fig. 1. Problem diagram for the requirement *Communicate*

2.2 Mechanisms and Patterns for Performance and Security

To satisfy performance and security requirements, different mechanisms – also called patterns – are available [3,10]. *Load Balancing* is such a mechanism that is used to distribute computational load evenly over two or more hardware components. The load balancing pattern consists of a component, which is called *Load Balancer*, and multiple hardware components that implement the same functionality. *Encryption* is an important means to achieve confidentiality. A plaintext is encrypted using a secret *key* and decrypted either using the same key (symmetric encryption) or a different key (asymmetric encryption).

3 Requirements Engineering

It is important that the results of the requirements analysis with problem frames can be easily re-used in later phases of the development process. Since UML is a widely used notation to express analysis and design artifacts in a software development process, we defined a new UML profile [4,2] that extends the UML meta-model to support problem-frame-based requirements analysis with UML. This profile can be used to create the diagrams for the problem frame approach. To address quality requirements in the requirement engineering process we enhance our UML profile with annotations for quality requirements as stereotypes.

3.1 UML Profile for Problem Frames

Using specialized stereotypes, our UML profile allows us to express the different diagrams occurring in the problem frame approach using UML diagrams.

A class with the stereotype ≪`machine`≫ represents the software to be developed (possibly complemented by some hardware). Jackson distinguishes the domain types biddable domains (represented by the stereotype ≪`BiddableDomain`≫) that are usually people, causal domains (represented by the stereotype ≪`CausalDomain`≫) that comply with some physical laws, and lexical domains (represented by the stereotype ≪`LexicalDomain`≫) that are data representations.

In problem diagrams, *interfaces* connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation *U!sendTM* (between *CA_communicate* and *User*) means that the phenomenon *sendTM* is controlled by the domain *User*. The interfaces are marked with specializations of the stereotype ≪`connection`≫, e.g., a user interface (≪`ui`≫) between *User* and *CA_communicate* machine in Fig. 1.

The stereotype ≪`requirement`≫ represents a functional or quality requirement. When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`refersTo`≫.

The problem diagram in Fig. 1 considers a chat application introduced in more detail in Sect. 4. It describes the requirement *Communicate*, e.g., it states that the *CA_communicate* machine can show to the *User* the *CurrentChatSession* on its *Display* (*CAC!*{*displayCCS*}). The requirement constrains the *CurrentChatSession* of the *User* and its *Display.* The requirement refers to the users and the text messages.

The problem frame approach substantially supports developers in analyzing problems to be solved. It points out what domains have to be considered, and what knowledge must be described and reasoned about when analyzing a problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge* are specified in *domain knowledge diagrams.*

3.2 Annotating Problem Descriptions with Quality Requirements

The problem frame approach proposed by Jackson provides a method that addresses functional requirements only. Quality requirements are not considered. We extended our UML profile for problem frames to complement functional requirements with security requirements [5]. Classes with stereotypes such as ≪`confidentiality`≫, ≪`integrity`≫ and corresponding attributes such as *attacker* or *stakeholder* address security requirements. The dependency from a quality requirement to a requirement is expressed with the stereotype ≪`complements`≫ (see Fig. 1). To provide support for annotating problem descriptions with performance requirements, we use the UML profile MARTE (Modeling and Analysis of Real-time and Embedded Systems) [12]. We focused on the GQAM package (Generic Quantitative Analysis Modeling) that contains basic concepts for modeling and analysis of domains based on software behavior, in particular performance. To define workload and behavior concerns we make use of the GQAM_Workload package by instantiating the appropriate attributes of this package. Each *BehaviorScenario* is composed of *Steps*, each of which can be refined as another *BehaviorScenario.* A behavior scenario is triggered by the *WorkloadEvent*, which may be generated by a stated *ArrivalPattern* such as the *ClosedPattern* that allows us to model a number of concurrent user and a think time (the time that the user waits between two requests) by instantiating the attributes *population* and *extDelay.* We define a *BehaviorScenario* composed of one *Step* for the requirement *Communicate_RT* (see Sect. 4.2), which is refined in three *BehaviorScenario* instances, each of which is composed of a single *Step.* The *Step* instances represent the requirements *Send_RT*, *Forward_RT* and *Receive_RT* that stand in the precedence relationship *Sequence* [12, p. 289].

4 Deriving Quality-Based Architectures

We now present our approach to derive software architectures, taking quality requirements into account. It comprises requirements analysis as well as the software architecture design. We illustrate our approach by a chat application, which allows a text-message-based communication via private I/O devices. Users should be able to communicate with other chat participants in a same chat room. We consider the *Communicate* functional requirement with the description "*Users can send text messages to a chat room, which should be shown to the users in that chat room in the current chat session in the correct temporal order on their displays*" and its corresponding quality requirement *Response Time* with the description "*The sent text message should be shown on the receiver's display in 1500 ms maximum*". Moreover, *Confidentiality* of the text messages should be preserved. Note that in order to specify performance and confidentiality requirements properly, more details have to be given.

4.1 Problem Diagrams

As described in Sect. 2.1, the first step in the software development process based on problem frames is to create a context diagram (not shown). We decompose the overall problem into subproblems represented by problem diagrams. Each problem diagram describes one subproblem with the corresponding requirement. We focus on the requirement *Communicate*. The corresponding problem diagram using our UML profile for problem frames is depicted in Fig. 1. It consists of the domains *User, TextMessage, CurrentChatSession* and *Display*. The requirement *Communicate* refers to the domains *User* and *TextMessage*, expressed by the stereotype ≪`refersTo`≫ and constrains the domains *CurrentChatSession* and *Display*, expressed by the stereotype ≪`constrains`≫.

4.2 Annotate Problem Diagrams with Quality Requirements

In this step, we address quality requirements by annotating problem diagrams with suitable stereotypes. The requirement *Communicate* is complemented by the confidentiality requirement *Communicate_Conf* that requires confidentiality of data transmission for *TextMessage* and the response time requirement *Communicate_RT* representing one *BehaviorScenario* composed of one *Step* described with the stereotype ≪`gaStep`≫ (see Fig. 1). The response time requirement is modeled by instantiating the relevant attributes of the *Step* class in the MARTE GQAM_Workload package described in Sect. 3.2. The *cause* attribute represents the triggered event, which is in our case a *ClosedPattern* with 100 concurrent users (*population*), each of which needs a think time of 1000 ms (*extDelay*). The *respT* attribute states that the required response time for sending text messages should be 1500 ms maximum. The *msgSize* attribute states that the sending text messages should be 5 KB maximum.

4.3 Choose Design Alternative and Create Architecture

We first create an initial architecture, where each machine domain in a problem diagram is mapped to a component. The initial architecture for the chat application (not shown) contains – among others – a component *CA_communicate* corresponding to the machine domain *CA_communicate* of Fig. 1.

The software architect then needs to take a design decision concerning the kind of distribution, e.g., client-server, peer-to-peer, or standalone. In the following, we describe the approach for a client-server architecture in more detail.

After having chosen a client-server architecture, we go back to the requirements description and **split the problem diagrams** in such a way that each subproblem is allocated to only one of the distributed components. This may lead us to introduce connection domains[2], e.g., networks. In our example, the problem diagram depicted in Fig. 1 is split into three problem diagrams, which address the problems of sending text messages to the server that belongs to the client (*Send*), forwarding text messages from the server to the receivers that belongs to the server (*Forward*), and receiving text messages that belongs to the client (*Receive*). For each of these three subproblems, we introduced the connection domain *Network* to achieve the distribution.

Analogously to splitting the problem diagrams, we also have to **split the corresponding quality requirements**. In case of a response time requirement, the response time should be divided so that all subproblems together satisfy the

[2] These are domains needed to establish a connection between other domains [6].

desired response time. The *Communicate* requirement states a response time of 1500 ms maximum. This must be achieved through the three subproblems *Send, Forward, Receive* and the time for data transmission over the network. We cannot meet the performance and specifically response time requirements, if we have no knowledge about the real circumstances in the environment. Therefore we specify knowledge about the network and the computational power of clients and server in a domain knowledge diagram. It contains specific knowledge about client and server, e.g., the number of processor cores, processor speed and memory. Additionally, we assume that the response time to transmit data over a network with 64 kb/s minimum is 400 ms.

To fulfill the confidentiality requirement for the problem *PD_communicate* (Fig. 1), we require confidentiality for each subproblem. Therefore, we annotate each subproblem with a corresponding refined confidentiality requirement. This requirement contains a *stakeholder* that is interested in preserving the confidentiality of data, and an *attacker* that the chat application should be protected against. The stakeholder in our case is the *User*, and the attacker is a *NetworkAttacker* who is able to attack the data transported over the network.

Concretized Quality Problem Diagrams describe solution approaches in terms of mechanisms and patterns. We elaborate the problem diagrams annotated with quality requirements from the previous step by introducing domains reflecting specific solution approaches.

For example, the problem diagram for the *Send* problem describes the problem of sending text messages with two additional quality requirements for security and performance, respectively. The requirement for performance states that sending a text message should be performed within 200 ms (allocated part of the 1500 ms). However, this requirement cannot be achieved by architectural means. Instead, it must be taken care of in the implementation. In such a case, we annotate the corresponding machine with a stereotype that serves as a hint to develop a particularly efficient implementation (≪`gaStep`≫) or an implementation that does not leak information (≪`confidentiality`≫).

The security requirement describes that a text message should be transmitted confidentially over an insecure network. To take this quality requirement into account, we specify the concretized quality problem diagram including an *Encryption* machine and domains for keys used for asymmetric encryption. This decision necessitates to also introduce new components on the receiver side, namely a new machine *Decryption* and a domain *ReceiverUserPrivateKey.*

In order to address the response time requirement in the *Forward* problem even under high load, we introduce a new machine *LoadBalancer* (see Fig. 2). It distributes the load from the network across several server components.

By now we have provided a suitable basis for quality-aware architectural design in the requirements analysis phase. To **design an architecture** that achieves the required level of performance and security, we make use of the split problem diagrams to allocate components to the client and to the server. Each machine in the split problem diagrams belongs to a component in the client or in the server according to functionality of that submachine. To design the architecture, we merge related components, apply design patterns (e.g., Facades), and use the solution domains for quality requirements (e.g.,*LoadBalancer*). The resulting software architecture for the chat application – represented as a UML composite structure diagram – is shown in Fig. 3.

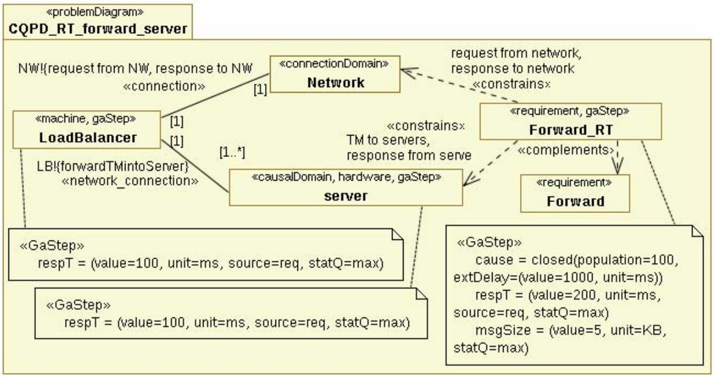


Fig. 2. Concretized quality problem diagram for the quality requirement *Forward_RT*

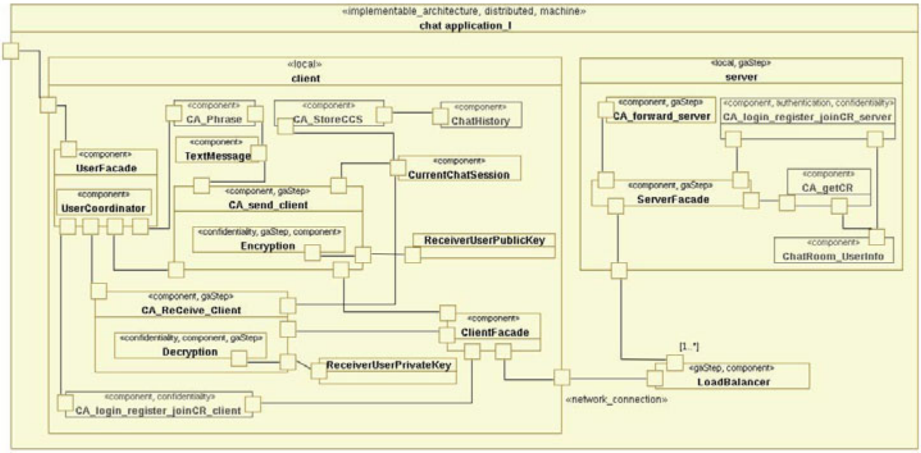


Fig. 3. Client-server architecture for the chat application

5 Related Work

Previous work often considers only one type of quality requirements during the software development process, e.g., security.

An approach to transform security requirements to design is provided by Mouratidis and Jürjens [9]. It starts with the goal-oriented security requirements engineering approach Secure Tropos [8], and connects it with a model-based security engineering approach, namely UMLsec [7].

Yskout et al. [14] present a semi-automated approach to support the transition from security requirements to architecture. They focus on delegation, authorization and auditing as security requirements. They presuppose an architecture that fulfills the functional requirements, and they apply security solutions to the functional architecture by transforming security requirements.

Attribute Driven Design (ADD) [13] is a method to design a conceptual architecture. It focuses on the high-level design of an architecture, and hence does not support detailed design. Identifying mechanisms to achieve quality attributes relies on the architect's expertise.

Q-ImPrESS [1] is a project that focuses on the generation and evaluation of architectures according to quality properties, in particular performance. The phases design and implementation of the software development process are particularly in focus. In contrast to our contribution, it does not use requirements descriptions as a starting point.

6 Conclusion

In this paper, we have presented a UML-based approach to design software architectures from requirements, taking quality requirements into account. We provide means to specify quality requirements thoroughly with problem diagrams, and we incorporate mechanisms or patterns addressing these requirements explicitly in the software architecture.

Our approach builds on established techniques such as problem frames, security and performance patterns. Its novelty lies in the fact that the different approaches are integrated and intertwined explicitly by an underlying methodology and a common notation. The notation as well as the methodology are open and can be developed further to enhance the power and breadth of the approach.

In the present work, we have not investigated possible conflicts between different quality requirements. We strive for a more systematic treatment of conflicting quality requirements. Moreover, we have concentrated on structural descriptions of software architectures. In the future, we will extend our approach to also support deriving behavioral descriptions for the developed architectures and automatically checking their coherence with the structural descriptions.

References

1. Becker, S., Dešić, S., Doppelhamer, J., Huljenić, D., Koziolek, H., Kruse, E., Masetti, M., Safonov, W., Skuliber, I., Stammel, J., Trifu, M., Tysiak, J., Weiss, R.: Q-ImPrESS Project Deliverable D1.1 – Requirements document. final version, Q-ImPrESS Consortium (2009)
2. Choppy, C., Hatebur, D., Heisel, M.: Systematic architectural design based on problem patterns. In: Avgeriou, P., Grundy, J., Hall, J., Lago, P., Mistrik, I. (eds.) Relating Software Requirements and Architectures, ch. 9. Springer, Heidelberg (to appear, 2011)
3. Ford, C., Gileadi, I., Purba, S., Moerman, M.: Patterns for Performance and Operability. Auerbach Publications (2008)
4. Hatebur, D., Heisel, M.: Making Pattern- and Model-Based Software Development More Rigorous. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 253–269. Springer, Heidelberg (2010)
5. Hatebur, D., Heisel, M.: A UML profile for requirements analysis of dependable software. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 317–331. Springer, Heidelberg (2010)
6. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
7. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)

8. Mouratidis, H.: A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People in England. PhD thesis, University of Sheffield, U.K (2004)
9. Mouratidis, H., Jürjens, J.: From goal-driven security requirements engineering to secure design. Int. J. Intell. Syst. 25, 813–840 (2010)
10. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. Wiley & Sons, Chichester (2005)
11. UML Revision Task Force. OMG Unified Modeling Language (UML), Superstructure, http://www.omg.org/spec/UML/2.3/Superstructure/PDF
12. UML Revision Task Force. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, http://www.omg.org/spec/MARTE/1.0/PDF
13. Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., Wood, B.: Attribute-Driven Design (ADD). Version 2.0, Software Engineering Institute (2006)
14. Yskout, K., Scandariato, R., Win, B.D., Joosen, W.: Transforming security requirements into architecture. In: Proc. of the 3rd Int. Conf. on Availability, Reliability and Security, pp. 1421–1428. IEEE Computer Society, Los Alamitos (2008)

Defining Architectural Viewpoints for Quality Concerns

Bedir Tekinerdogan[1] and Hasan Sözer[2]

[1] Bilkent University, Department of Computer Engineering
Bilkent 06800 Ankara, Turkey
bedir@cs.bilkent.edu.tr
[2] Ozyegin University, Department of Computer Engineering
Istanbul, Turkey
hasan.sozer@ozyegin.edu.tr

Abstract. A common practice in software architecture design is to apply architectural views to model the design decisions for the various stakeholder concerns. When dealing with quality concerns, however, it is more difficult to address these explicitly in the architectural views. This is because quality concerns do not easily match the architectural elements that seem to be primarily functional in nature. As a result, the communication and analysis of these quality concerns becomes more problematic in practice. We introduce a general and practical approach for supporting architects to model quality concerns by extending the architectural viewpoints of the so-called V&B approach. We illustrate the approach for defining recoverability and adaptability viewpoints for an open source software architecture.

Keywords: Software Architecture Modeling, Architectural Views, Quality Concerns.

1 Introduction

An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern [2]. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. Architectural views conform to viewpoints that represent the conventions for constructing and using a view. Because of the different concerns that need to be addressed for different systems, the current trend recognizes that the set of views should not be fixed but multiple viewpoints might be introduced instead. Certainly, existing multi-view approaches are important for representing the structure and functionality of the system and are necessary to document the architecture systematically. Yet, an analysis of the existing multi-view approaches reveals that they still appear to be incomplete when considering quality concerns. The ISO/IEC 42010 [4] standard intentionally does not define particular viewpoints to address the different concerns. In the V&B approach, quality concerns appear to be implicit in the different views but no specific viewpoints have been proposed to represent quality concerns. One could argue that for addressing quality concerns software architecture analysis approaches have been introduced. The difficulty here is that these

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 26–34, 2011.

approaches usually apply a separate quality model, such as queuing networks or process algebra, to analyze the quality properties. Although these models represent precise calculations they do not depict the decomposition of the architecture and an additional translation from the evaluation of the quality model needs to be performed. To represent quality concerns more explicitly, preferably an architectural view is required to model the decomposition of the architecture based on the required quality concern. In this context, we introduce an approach for defining architectural viewpoints for modeling quality concerns. We illustrate the approach for two different quality concerns; recoverability and adaptability. The approach is applied to the open source media player application, MPlayer.

The remainder of this paper is organized as follows. Section 2 introduces the case study and the problem statement in which we describe the need for architectural decomposition for quality concerns. Section 3 describes the concepts for modeling architectural viewpoints for quality concerns. Section 4 provides the related work. Finally, section 5 provides the conclusions.

2 Problem Statement

2.1 Case Study: MPlayer

MPlayer [6] is a media player, which supports many input formats, codecs and output drivers. It is available under the GNU General Public License. Figure 1 presents a simplified module view of the MPlayer software architecture with basic implementation units and direct dependencies among them.

Here *Stream* represents the module that reads the input media and provides buffering, seek and skip functions. *Demuxer* demultiplexes (separates) the input to audio and video channels, and reads them from buffered packages. *Mplayer* connects all the other modules, and maintains the synchronization of audio and video. *Libmpcodecs* embodies the set of available codecs. *Libvo* displays video frames. *Libao* controls the playing of audio. *Gui* provides the graphical user interface (GUI) of MPlayer.

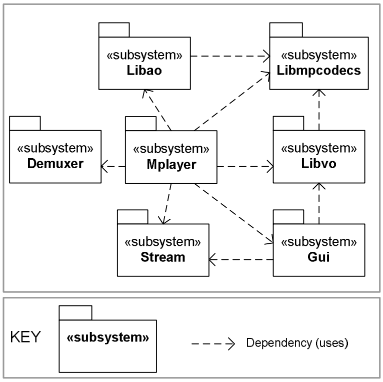
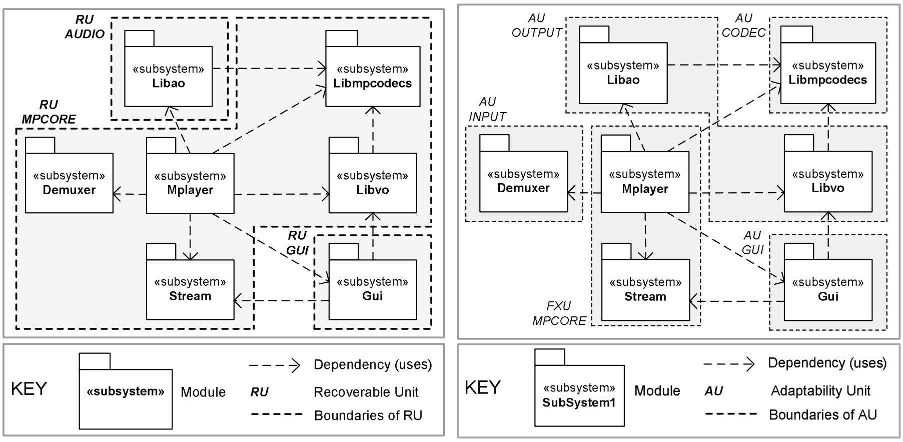


Fig. 1. Module View of the MPlayer Software Architecture

2.2 Architectural Decomposition for Quality Concerns

When designing the architecture for the MPlayer besides of the functional concerns also non- functional concerns have to be taken in to account. In the following, we will consider the recoverability and adaptability concerns for MPlayer.

Recoverability refers to the ability to recover from errors [1]. Recoverability has a separate impact on the system and is usually not always aligned with the individual components in the system. Figure 2a represents an example of the required decomposition of the architecture for recoverability in the MPlayer case. Here, a decomposition unit is called *recoverable unit (RU)*. Each RU should be independently recoverable. As we can see in Figure 2a, three recoverable units have been defined: *RU AUDIO, RU MPCORE*, and *RU GUI*. In fact, Figure 2a provides two views on top of each other, the module view and the view related to recoverability, which overlays the module view. Obviously many different decomposition alternatives are possible. Each design alternative will require a different impact on the system. Unfortunately, the architectural decomposition in Figure 1 is not sufficient to communicate design decisions about the recoverability. On the other hand, although Figure 2a provides the impact of recoverability it models two concerns at the same time and likewise it violates the separation of concerns principle. A more complicated decomposition for recovery would be harder to model, and in case more than one concern needs to be modeled the model becomes less useful for communication about the concerns. Both figures are also less suitable to support the analysis of recoverability and/or to guide the implementation of the system based on the architecture.



a) b)

Fig. 2. Required decomposition for MPlayer architecture for Recoverability (a) and Adaptability (b)

Adaptability is defined as the ease with which a system can change [5]. There are several types of adaptation techniques applied in practice. These techniques are applied at different phases (i.e., compile-time, run-time) and at different abstraction levels (e.g., source code, architecture description). Knowing the adaptability properties of architectural components early on is important to communicate and guide the system development. In addition, similar to recoverability, one may define different architecture design alternatives that behave differently with respect to adaptability properties. Figure 2b shows an example decomposition of the architecture that might be required for adaptability. Here, the decomposition unit is called adaptability unit (AU). Each AU shows whether the unit is adaptable or fixed, and should define the adaptability properties. As we can see in Figure 2b five adaptability units have been defined: AU INPUT, AU OUTPUT, AU GUI, AU CODEC, and FXU MPCORE (fixed unit). Again this provides two views on top of each other, the module view and the view related to adaptability which actually overlays the module view. Unfortunately, the architectural decomposition of Figure 1 alone is not suitable to support the communication, analysis and guidance of the implementation for adaptability.

When we consider other quality concerns the situation does not seem to be different than in the case for recoverability and adaptability. Reusability will require a different view on the architecture in which, for example, the reusable components need to be depicted. Performance will require, for example to view the elements of the system based on their influence on the performance, etc. We could try to visualize these quality concerns on the base view that we are working on (dominant decomposition), however this will clutter the module view and eventually will decrease the understandability of the architectural description. In fact, this would also not be in alignment with the overall strategy in architectural view modeling, i.e. define an architectural view for the relevant concerns. As such, we believe that the relevant quality concern should also be represented using the corresponding views.

3 Quality Viewpoints

In this section, we provide an approach for defining architectural viewpoints for quality concerns. The overall process is shown in Figure 3. The process starts with defining the stakeholders of the concerns. For each stakeholder the concerns are defined which are categorized as *functional concerns* and *quality concerns*.

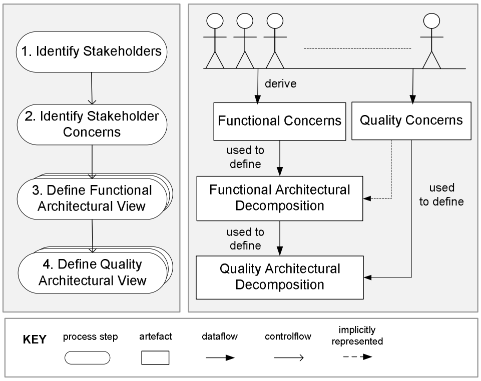

1. Identify Stakeholders
2. Identify Stakeholder Concerns
3. Define Functional Architectural View
4. Define Quality Architectural View
derive
Functional Concerns
Quality Concerns
used to define
Functional Architectural Decomposition
used to define
Quality Architectural Decomposition
used to define
KEY
process step
artefact
dataflow
controlflow
implicitly represented

Fig. 3. General Approach for Architectural View Modeling

The stakeholder concerns form an input for the architecture view modeling process. Hereby the architectural view that represents the *functional view* is described. On the other hand, quality concerns are modeled using *quality views*. For different quality concerns the architect might need to define a different architectural view. Similar to functional views, quality views will be based on architectural viewpoints. Defining a new architectural viewpoint implies writing a *viewpoint guide*. This is similar to the notion of *style guide* as defined in [2]. The *viewpoint guide* defines the vocabulary of the architectural element and relation types, and defines the rules for how that vocabulary can be used. For defining a viewpoint guide for a particular quality concern we apply the template as defined in Table 1. The viewpoint guide for quality concerns is largely the same as for the viewpoints that address functional concerns. The important difference here is that the architectural elements now are used to explicitly represent quality concerns in the architectural decomposition. Further, the quality view is applied to a functional view.

In the template of Table 1, this is defined by the field *Base View,* representing the view on which the quality view is applied. The base view could be for example the module view, component and connector view or deployment view. To make a distinction among these, the name of the viewpoint should be described accordingly, e.g. *Recoverability:Decomposition*, *Recoverability:Deployment*, *Adaptability: Process* etc. Here the symbol : refers to the mapping of the quality view on the functional view. In the following, we will give two distinct examples of the application of the viewpoint guide for quality concerns.

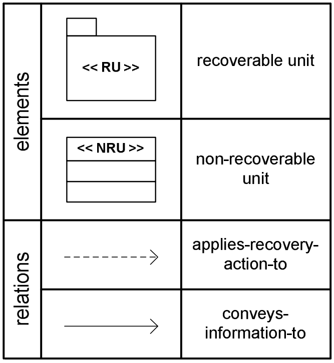
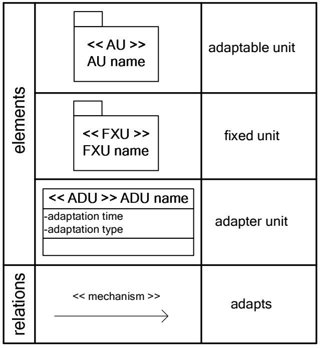
Table 1. Viewpoint Guide Template for Quality Concerns

Viewpoint Element	**Description**
Name	Unique name for the viewpoint concatenated with the view it overlays
Element Types	The architectural element types native to the viewpoint
Relation Types	The relation types among architectural elements
Properties of Elements	Additional information on the element types
Properties of Relations	Additional information on the relation types
Topology Constraints	The rules of composition of the elements and relations.
Notation	The adopted notation for the element types and relation types. The notation can be textual or visual.
Base View	The view that can be overlaid
Relation to other views/viewpoints	The relation to other viewpoints other than the base viewpoint

3.1 Example – Recoverability

Similar to the case where we separate the views for different concerns (e.g. deployment view separate from module view), we also provide a separate view for recoverability. To define the template for the recoverability view we introduce the recoverability viewpoint (Table 2) as an explicit viewpoint for depicting the architecture from the recoverability viewpoint. Unlike conventional analysis techniques that require different models, recoverability views directly represent the decomposition of the architecture and as such help to understand the structure of the system related to the recoverability concern. In essence, the recoverability viewpoint considers RUs as first class elements and represents the units of isolation, error containment and recovery control. The relation types define the relations for coordination and application of recovery actions.

Table 2. Recoverability Viewpoint (left) and Adaptability Viewpoint Guide (right)

Viewpoint Element	**Description**	
Name	Recoverability Viewpoint:Module View	Adaptability Viewpoint:Module View
Element Types	• Recovery Unit (RU) – represents a set of modules that can be recovered together, independently from other elements it is connected to. • Non-Recovery Unit (NRU) – an element that cannot be recovered independent of other RUs and NRUs.	• Adaptable Unit (AU): a set of modules that can be adapted independently from other modules of the system. • Fixed Unit (FXU): a set of modules that cannot be adapted. • Adapter Unit (ADU): an entity, which implements an adaptation mechanism.
Relation Types	• applies-recovery-action-to • conveys-information-to	• adapts
Properties of Elements	• RU: set of system modules, criticality, reliability, types of errors that can be detected, supported recovery actions, type of isolation. • NRU: types of errors that can be detected.	• AU: the set of modules, adapted properties • FXU: the set of modules • ADU: adaptation time, type of adaptation
Properties of Relations	• applies-recovery-action-to: type of communication, timing constraints. • conveys-information-to: type of communication, timing constraints.	• adapts: the type of mechanism used for adaptation.
Topology Constraints	• The target of an applies-action-to relation can only be a RU.	• the adapts relation can only be defined from an ADU to AU.
Notation	 elements << RU >> recoverable unit << NRU >> non-recoverable unit relations applies-recovery-action-to conveys-information-to	 elements << AU >> AU name adaptable unit << FXU >> FXU name fixed unit << ADU >> ADU name -adaptation time -adaptation type adapter unit relations << mechanism >> adapts
Base view	Module View	Module View
Relation to other views/viewpoints	• Dynamic views for depicting recovery scenarios.	• Deployment view for relating platform-specific adaptations.

We can document the viewpoint by using the viewpoint guide as defined in Table 1. An example application of the viewpoint guide to the MPlayer case is shown in Figure 4 (left). The figure represents the case as defined in Figure 2 but now we view the system solely from a recoverability concern perspective. The view includes three RUs and two non-recoverable units (NRUs) as first class abstractions. The relations represent the specific recovery mechanisms among the recovery units. Typically, this view can be used by reliability engineers to communicate about the reliability and fault tolerance of the system, to use this for guiding the implementation of recovery mechanisms in the corresponding units, and to analyze the different decompositions for recoverability.

3.2 Example – Adaptability

The adaptability viewpoint guide that we have defined is shown in the right column of Table 2. Here, we have identified three units and one relation. We have defined an *adaptable unit* and a *fixed unit* to differentiate software modules that are considered for adaptation from the ones that are not. We have defined an additional unit, *adapter unit*, which represents the implementer of the adaptation mechanism. This can be a part of the system or an external entity. Its attributes identify the time (compile-time or run-time) and type (manual or automatic) of adaptation implemented. The only relation defined is the *adapts* relation, which is defined from an adapter unit to an adaptable unit, emphasizing the mechanism used for adaptation. Focusing on this property of the system led us to define a decomposition (*Figure 4*) that comprises fixed and adaptable units and additional modules to support adaptability.

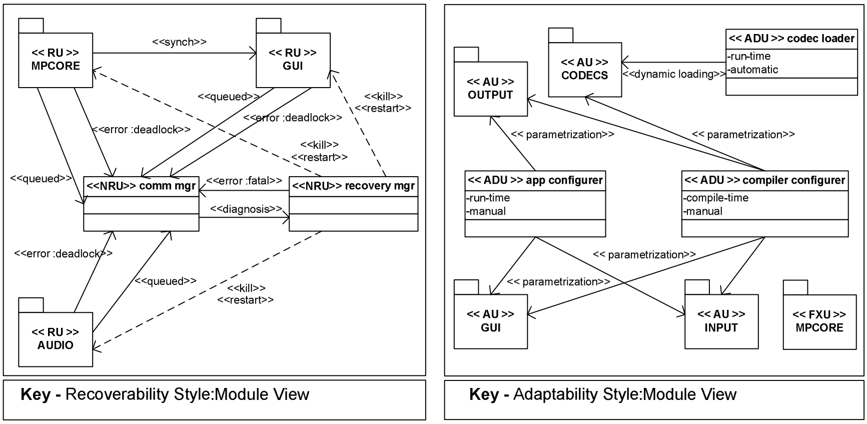


Fig. 4. Recoverability View (left) and Adaptability View (right) for MPlayer case

4 Related Work

Architectural Perspectives [8] are a collection of activities, tactics and guidelines to modify a set of existing views to document and analyze quality properties. Architectural perspectives as such are basically guidelines that work on multiple views together. An analysis of the Architectural Perspectives and our approach shows that the crosscutting nature of quality concerns can be both observed *within* an architectural view and *across* architectural views. Both approaches focus on providing a solution to the crosscutting problem. We have chosen for providing separate architectural viewpoints for quality concerns. It might be interesting to look at integrating the guidelines provided by the *Architectural Perspectives* and the definition/usage of the viewpoints developed by our approach. In that sense the approaches can also be considered as complimentary to each other.

Architectural tactics [1] aim at identifying architectural decisions related to a quality attribute requirement and composing these into an architecture design. Defining explicit viewpoints for quality concerns can help to model and reason about the application of architectural tactics.

Several software architecture analysis approaches have been introduced for addressing quality properties. The goal of these approaches is to assess whether or not a given architecture design satisfies desired concerns including quality requirements. The main aim of the viewpoint definitions in our approach, on the other hand, is to communicate and support the architectural design with respect to quality concerns. As such our work can directly support the architectural analysis to select feasible design alternatives.

5 Conclusion

The evolution of architectural view modeling can be characterized as a gradual shift from defining fixed set of multiple views to an understanding in which the set of views for architecture description is not bounded but open, dependent on the stakeholder concerns. Yet another step in the evolution of architectural view modeling is the focus on quality concerns in architectural views. From both our research activities and practical experiences in an industrial context [7] we can observe that quality concerns cannot be easily represented in current architectural views and tend to crosscut elements within an architectural view. We have proposed a solution to this problem by providing an approach for defining architectural viewpoints for quality concerns. From our experience, the explicit modeling of architectural viewpoint for quality concerns seemed to be a practical instrument [7]. Explicit viewpoints for quality concerns do not only improve the understanding and communication of the architecture but also support the analysis of these concerns. Our future work will include the analysis of quality concerns based on the architectural viewpoints that we have developed.

References

[1] Avizienis, A., et al.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. 1(1), 11–33 (2004)

[2] Clements, P., et al.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2002)

[3] Garlan, D., Barnes, J.M., Schmerl, B.R., Celiku, O.: Evolution styles: Foundations and tool support for software architecture evolution. In: Proc. of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2009), pp. 131–140 (2009)

[4] [ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010) (identical to ANSI/IEEE Std1471–2000) (July 2007)

[5] Kell, S.: A Survey of Practical Software Adaptation Techniques. Journal of Universal Computer Science 14(13), 2110–2157 (2008)

[6] MPlayer official website, `http://www.mplayerhq.hu/` (accessed March 2010)

[7] Sözer, H., Tekinerdogan, B., Akşit, M.: FLORA: A Framework for Decomposing Software Architecture to Introduce Local Recovery. Wiley Software Practice and Experience Journal 39(10), 869–889 (2009)

[8] Rozanski, N., Woods, E.: Software Systems Architecture – Working with Stakeholders using Viewpoints and Perspectives. Addison-Wesley, Reading (2005)

A Question-Based Method for Deriving Software Architectures*

Marco Müller, Benjamin Kersten, and Michael Goedicke

paluno - The Ruhr Institute for Software Technology, University of Duisburg–Essen
Gerlingstraße 16, 45127 Essen, Germany
{marco.mueller,benjamin.kersten,michael.goedicke}@paluno.uni-due.de

Abstract. Although several approaches exist for deriving architectures from requirements and environmental constraints, most solutions rely on experienced architects for proposing and choosing feasible architectural solutions. It is critical to develop architecture systematically and without strong dependencies on experienced architects, because the architecture has a deep impact on the quality of a system. This paper presents a question-based approach for efficiently finding architecture candidates using annotated pattern and style catalogues. Following this approach allows for a systematic development of architecture, that provides documented common experience.

1 Motivation

The development of software architecture is a challenge, even when requirements, environmental constraints, and the domain of a software system are clear. This is especially true, when a development team includes no experienced architect. For building upon common knowledge and best practices, the usage of catalogues containing architectural patterns and styles (e.g. [1, 2]) has shown to be valuable. These catalogues are subject to architectural design methods, that aim to be a guide for deriving architectures from requirements (e.g. [3–6]). In these methods, the catalogues are used as a reference to find solutions for an architectural problem by choosing applicable patterns and styles (called *solution candidates* in this document) from the catalogue.

Trying to find an applicable solution candidate is, however, not enough. Several questions should be answered during that selection: E.g. What are the criteria for a candidate to be applicable? Are there constraints (e.g. business or technical)? Is the candidate a good choice to meet the quality requirements? When quality requirements are contradictory, is the candidate a good tradeoff? These questions target the selection of candidates from a catalogue. Most existing approaches are imprecise or don't provide any aid for choosing good candidates from catalogues [7].

* The work reported herein was funded by the German Research Foundation under the grant no. GO 774/5-1.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 35–42, 2011.

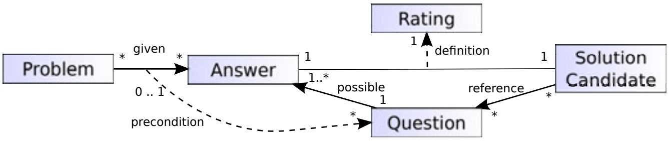


Fig. 1. Elements of the question-based method and their references

Sequentially evaluating the candidates of a catalogue for given requirements and constraints in a given domain (called *problem* in the remainder of this document) is not efficient. In this paper, we propose a process providing aid in this selection, by relating a specific problem to common experience. We extend the solution candidates in catalogues with rated questions. By answering these questions regarding a specific problem, a small set of candidates can quickly be identified as promising for that problem. The presented approach aims at helping to derive architectures that also meet quality requirements. To achieve this goal, the presented approach does not aim to elicit a single solution. Instead, it suggests solution candidates that are promising to meet the quality requirements. Before actually choosing a candidate, the proposed process provides for iteratively evaluating the most-promising candidates.

The remainder of this paper is structured as follows: Section 2 explains the extension of meta data for solution candidate catalogues. Section 3 describes the process using this meta data for efficiently selecting solutions. The provided tool support is introduced in section 4. Section 5 describes the current state of evaluation of the approach. Section 6 presents related work, before we conclude and present future work in section 7.

2 Annotating Architecture Patterns with Questions

In order to efficiently find solution candidates for problems, we annotate each candidate in a catalogue with questions and rated answers. The questions' goal is to find out how the solution candidates relate to a specific problem. Thus the question may target functional requirements, quality requirements, environmental constraints, and domain constraints.

The meta data for solution candidates is structured as shown in figure 1. Questions define a set of possible answers. These questions are referenced by the candidates. Thus different candidates can share the same questions. For each solution, a different rating may be defined for an answer. The rating describes whether a solution contributes positively $(0, 1.0]$ or negatively $[-1.0, 0]$ to a problem. The rating may also be *excludes*, which means that the solution candidate contradicts the problem, and is thus excluded. If a question is not answered (yet), its rating is 0 by default. In addition, specific answers can be preconditions for other questions to be asked. The elements are formally defined as follows:

- p is a problem definition,
- Q is a set of questions,
- a_q^p is an answer given to question q regarding problem p,
- $A_q, q \in Q$ is a set of possible answers for a question,
- A^p the set of given answers a_q^p for all questions $q \in Q$,
- S is a set of solution candidates,
- $r_s(a) \in [-1.0, +1.0] \cup \{\text{excludes}\}$ is the rating for the answer $a \in A_q$ of question $q \in Q$ for the solution candidate $s \in S$.

A solution candidate is a pattern description as found in pattern catalogues. For our purpose we define a solution candidate to be $s = \langle Q_s, \text{pre}_s \rangle$, with $Q_s \subseteq Q$ questions referenced by the solution candidate and $\text{pre}_s : Q_s \rightarrow 2^{A_{q_s}}, q_s \in Q_s$ being a function stating the answers that are preconditions for a question reference. The set of answers given for the questions referenced by s is A_s. A question reference q_s is called *enabled* for the problem p if: (1) it has not been answered for the problem in focus and, (2) it has either no precondition or (3) any precondition is a given answer:

$$\text{enabled}_{p,s}(q_s) := \nexists a_q^p \wedge (\text{pre}_s(q_s) = \emptyset \vee \exists a(a \in \text{pre}_s(q_s) \wedge a \in A^p))$$

Otherwise the reference is called disabled. A question is called enabled if it has any enabled question reference: $\text{enabled}_p(q) = \exists s(s \in S \wedge \text{enabled}_{p,s}(q))$. Analogously, a question is called disabled otherwise. At last, the rating r_s for a solution candidate s regarding all given answers A^p is the average of the single ratings or *excludes* if any rating is *excludes*:

$$r_s(A^p) := \begin{cases} \text{excludes} & \text{, if } \exists a(a \in A^p \wedge r_s(a) = \text{excludes}) \\ \frac{\sum\limits_{a \in A^p} r_s(a)}{|A_s|} & \text{, else} \end{cases}$$

3 A Process for Identifying Architectural Candidates from Requirements and Context

The elements described in section 2 can be used for efficiently identifying promising solution candidates for problems. In this section, we introduce the process for systematically deriving architecture candidates in an iterative top-down approach. This process is schematically depicted in figure 2. In each iteration promising candidates are found for the architecture. During the first iteration, the considered problem is the overall system, while in later iterations, the candidates are refined. Each iteration builds upon the results of the preceding iterations. Thus a tree of architecture candidates is spanned. The candidates are evaluated after each iteration. Due to this evaluation, branches of candidates that cannot fulfill the quality requirements are excluded early. The result of the process is a set of promising architecture candidates. The process is supposed to be executed by a person knowing the requirements, the context, and the domain of the system. This person is the reference for the process to the concrete problem.

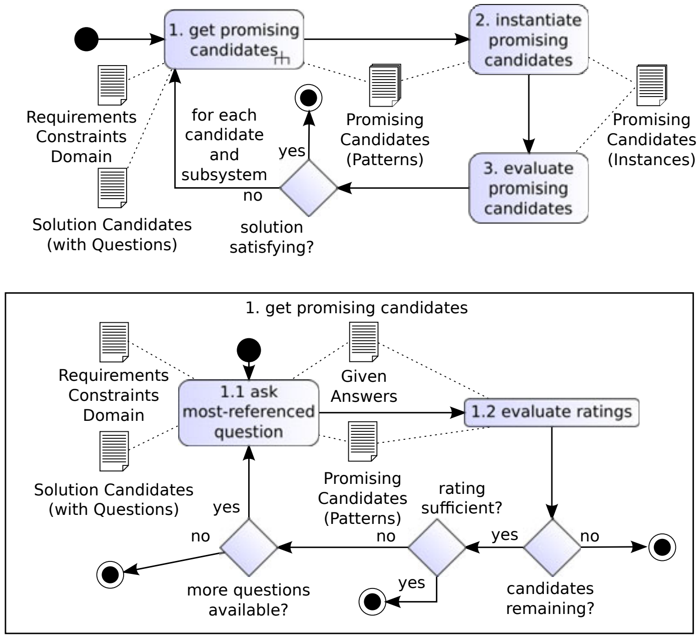

1. get promising candidates
2. instantiate promising candidates
3. evaluate promising candidates
Requirements Constraints Domain
Solution Candidates (with Questions)
for each candidate and subsystem
yes
no
solution satisfying?
Promising Candidates (Patterns)
Promising Candidates (Instances)
1. get promising candidates
1.1 ask most-referenced question
1.2 evaluate ratings
Requirements Constraints Domain
Solution Candidates (with Questions)
Given Answers
Promising Candidates (Patterns)
rating sufficient?
yes
no
yes
no
more questions available?
yes
no
candidates remaining?

Fig. 2. An overview of the process

In the following, the process is described in detail. Step (1) is to *get promising candidates* for the problem. The promising candidates is an ordered set $(S_{A^p}, \geq)$. S_{A^p} is the set of candidates which have not been excluded by one of the given answers A^p:

$$S_{A^p} = \{s | s \in S \wedge \forall a (a \in A^p \wedge r_s(a) \neq \text{excludes})\}.$$

The candidates are ordered by the ratings of the given answers:

$$s \geq s' \Leftrightarrow r_s(A^p) \geq r_{s'}(A^p).$$

To find these candidates, the first substep *ask most-referenced question* (1.1) is executed. In this substep, one of the questions that are referenced by the most remaining candidates are asked to the person knowing the problem p. The number of enabled references to a question is identified by

$$ref(q) = \sum_{s \in S} |\{q | q \in Q \wedge \text{enabled}_{p,s}(q)\}|.$$

Using this, the ordered set of enabled questions can be defined as $(\text{enabled}_p(q), \geq)$, with $q \geq q' \Leftrightarrow |ref(q)| \geq |ref(q')|$. In the next substep (1.2), the ratings are evaluated. Due to the definition of $\text{enabled}_{p,s}(q)$, the recently answered question is removed from the set of enabled questions. Analogously, solution candidates with $r_s(a_q^p) = \text{excludes}$ are removed from the set of promising candidates. Their question references are thus disabled.

At this point, no more candidates might be available. In this case, the catalogue of solution candidates does not provide an applicable solution candidate. If more questions are enabled, substep (1.1) is repeated with these questions. Otherwise step (1) is finished. The result is a set of patterns that represent promising candidates. The candidates with the highest rating are the most promising. Step (1) can be stopped after each evaluation, because the process does not require all questions to be answered. The rating is more specific with more answered questions.

The process aims at proposing candidates based on common knowledge. Thus the resulting candidates still need to be evaluated to confirm their feasibility and applicability. In the next step *instantiate & evaluate promising candidates* (2), the patterns resulting from step (1) are instantiated to model an architecture that fulfills the functional requirements and the constraints. The instantiated architectures are then evaluated e.g. using tradeoff analysis techniques. The process does not constrain the choice of methods or tools for evaluating architectures. When the evaluation shows that a candidate will most likely not meet the quality requirements, or is excluded due to a tradeoff analysis, it is removed from the list of candidates for the given process interation. If the instantiated architecture is detailed enough, the process can be stopped. Otherwise, for each candidate, each subsystem is taken as problem p for a next iteration. The set of questions and candidates are reset and the process starts a next iteration. Eventually, the iterations result in an architecture on the desired level of detail, or the process shows that no solution candidates are applicable. In the latter case, more patterns are necessary, or the requirements have to be adapted.

4 Tool Support

To support the approach, a tool was developed to store the description of solution candidates, including questions and ratings. The tool allows for easily modifying and querying the catalogue of solution candidates. We implemented this catalogue using a Semantic Media Wiki (SMW), a semantic extension for Media Wiki. The wiki allows for easily creating and modifying informal descriptions of candidates, that define relationships to each other for an easy understanding of the candidates and their environment (e.g. related patterns). The semantic extensions allow for defining typed relationships between pages, as well as properties for pages. Using these properties, relationships and pages are enriched with semantic information. The structure of the semantic properties is used as a meta model. Semantic wikis allow for defining complex queries over the modeled data.

In the semantic wiki used as solution candidate catalogue, the meta model was designed to define the structure necessary for the presented approach, as shown in figure 1. The elements in this figure represent data types (boxes) for wiki pages and their relationship types (arrows). The rating for an answer regarding a specific solution was realized using complex property types (the record type in SMW).

Furthermore, to support the structured description of solution candidates, we used semantic forms. They provide forms to guide editors of the wiki to reuse questions that are already defined, and to provide further semantic information, e.g. related patterns.

5 Example

The process was evaluated in internal software development projects. In this section, we show an excerpt of the process execution for designing a chat application, as it was used by students in a seminar. In this evaluation, a pattern catalogue with 10 patterns referencing 25 questions was used. The ratings were defined by experience. The functional and quality requirements for the application were given, as well as technical constraints: the framework to use was predefined due to the environment the software should be run.

In the first iteration, in step (1) of the process, the most-referenced questions were asked. An excerpt of the questions and given answers in this iteration is shown in table 1. The tables shows only the ratings for the given answers. As a result of step (1), the candidates *Client/Server* and *Simple Peer-to-Peer* were identified to be the most-promising candidates. The other candidates were excluded or had a significantly lower rating. In step (2), both candidates were instantiated. I.e. the styles were used as alternative designs for the application on the top-most abstraction level. In step (3), the instantiated architectures were evaluated by a performance-test, as the performance was one of the most important requirements. Both candidates passed the evaluation. Thus in the next iterations, each candidate was considered further when the architecture was refined. After five iterations, one solution (client/server-based) was considered detailed enough and satisfying.

Table 1. Rated answers for the first iteration of the chat application

Candidate \ Question	(1)	(2)	(3)	...	Average
Client/Server	1	0.5	0.9		0.19
Simple Peer-to-Peer	1	0	-0.1		0.22
Standalone	excludes	x	x		0.03
Shared State	-0.5	x	-0.8		0.07
Batch-Sequential	0.2	0	x		0.02
Pipes & Filters	0.2	0	x	...	0.0
Publish-Subscribe	0.2	0.4	-0.1		0.14
Event-Based	0.1	0.2	-0.3		0.13
Blackboard	x	0.6	-0.8		0.04
Layered	x	x	0.3		0.11

(1) Is the system necessarily distributed? → Yes

(2) Are there more clients expected than can be handled by a single node? → No

(3) Does the system conduct sensible / confidential data? → Yes

6 Related Work

Hofmeister et al. [8] compared five industrial software architecture design methods to extract a general design model from them. In their model, our approach can be used as the the Architectural Synthesis activity.

Related work is first of all found in different architecture derivation methods. These methods vary in their abstraction level and the development phases that are considered. For instance, Rational Unified Process (RUP)[9] is dealing with any software development process phase from early requirements to production and evolution. When it comes to the selection of patterns as solutions for a problem, RUP allows to integrate different architecture methods. It thus does not provide any guidance for this task.

Methods that are more specific in this point emphasize the design phase and describe how to select certain architecture styles and patterns for a given problem. For example, Attribute Driven Design (ADD) [3] is a method approximating this task. It defines a process to derive an architecture from requirements using patterns. However, ADD does not describe how to elicit a matching pattern from a large pattern catalogue, except for sequentially evaluating each element in the catalogue (cf. [10]). There are more architecture methods that are imprecise at this point such as Object Oriented Modeling and Design [4], Siemens 4 Views [5] or Architectural Separation of Concerns [6]. Therefore, our approach bridges the gap of pattern elicitation found in related work. Our approach is not designed to complement these methods, but it can be integrated to enrich them.

Zdun uses questions to select architecture patterns in [11]. In this approach, questions are directed to key characteristics of a group of patterns (e.g. "*How to realize asynchronous result handling?*"). The answers are patterns, which are related to criteria supporting the decision process. We believe that our approach is more suitable for less experienced teams, because of the are related to the system's requirements and context. This is, however, subject to validation.

Bode and Riebisch [7] relate solutions to quality requirements. Their work focuses on rating the impact of patterns (called solution instruments in this context) on quality goals. They first refine quality goals with subgoals. Then they group patterns to solution principles, e.g. modularization. A experience-based rating is then defined between solution principles and subgoals. In contrast to our work, Bode and Riebisch develop context-independent ratings. We are confident, that the specific requirements, the environmental constraints, and the domain influence these ratings.

7 Conclusion and Future Work

In this paper we presented a question-based approach for systematically proposing architecture candidates for software systems and subsystems. To select appropriate architecture patterns and styles we annotated each of these candidates within a catalogue with semantic meta data: questions, answers, and ratings. A developer who is familiar with the requiremens answers questions within a defined process to exclude irrelevant candidates and to guide the selection by rating the candidate regarding the given answers. To evaluate our approach, we evaluated the process in several internal software development processes.

As future work, we plan to further evaluate the process. For a deeper reflection, a larger base of solution candidates is necessary, as well as refined ratings for answers. We thus plan to extend the evaluation to larger projects with students and industrial partners. Furthermore, we are developing an interactive application on top of the wiki as already provided tool, to guide users through the process in a user-friendly way. We also plan to publish the semantic wiki as catalogue for general usage and for using the common experience of practitioners and researchers for extending the catalogue with more patterns and styles, and for refining the ratings. We also plan to find methods for refining the ratings by evaluating data collected from projects that use the presented process.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice, 1st edn. John Wiley & Sons, Chichester (2009)
2. Gamma, E., Helm, R., Johnson, R.E.: Design Patterns. Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Longman, Amsterdam (1994)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice (SEI Series in Software Engineering), 2nd edn. Addison-Wesley Longman, Amsterdam (2003)
4. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W.: Object-Oriented Modeling and Design, United states ed edn. Prentice-Hall, Englewood Cliffs (1991)
5. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture: A Practical Guide for Software Designers (Addison-Wesley Object Technology). Addison-Wesley Longman, Amsterdam (1999)
6. Jazayeri, M., Ran, A., van der Linden, F.: Software Architecture for Product Families. Addison-Wesley Longman, Amsterdam (2000)
7. Bode, S., Riebisch, M.: Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability. In: Babar, M., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 182–197. Springer, Heidelberg (2010)
8. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, J.H., Ran, A., America, P.: Generalizing a model of software architecture design from five industrial approaches. In: WICSA, pp. 77–88 (2005)
9. Kruchten, P.: The Rational Unified Process: An Introduction (Addison-Wesley Object Technology), 2nd sub edn. Addison-Wesley Longman, Amsterdam (2000)
10. Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., Wood, B.: Attribute-driven design (add), version 2.0. Technical report, Software Engineering Institute (2007)
11. Zdun, U.: Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis. Software: Practice and Experience 37(9), 983–1016 (2007)

Performance Simulation of Runtime Reconfigurable Component-Based Software Architectures

Robert von Massow, André van Hoorn, and Wilhelm Hasselbring

Software Engineering Group, University of Kiel, D-24098 Kiel, Germany

Abstract. Architectural runtime reconfiguration is a promising means for controlling the quality of service (QoS) of distributed software systems. Particularly self-adaptation approaches rely on runtime reconfiguration capabilities provided by the systems under control. For example, our online capacity management approach SLAstic employs changing component deployments and server allocations to control the performance and resource efficiency of component-based (C-B) software systems at runtime.

In this context, we developed a performance simulator for runtime configurable C-B software systems, called SLAstic.SIM. The system architectures to be simulated are specified as instances of the Palladio Component Model (PCM). The simulation is driven by external workload traces and reconfiguration plans which can be requested during simulation, based on continuously accessible monitoring data of the simulated systems. This paper demonstrates SLAstic.SIM including a quantitative evaluation of its performance.

1 Introduction

Self-adaptation approaches for software systems [1] rely on runtime reconfiguration capabilities provided by the controlled system. For example, our SLAstic [2,3] approach for increased resource efficiency of distributed component-based (C-B) software architectures changes the deployment of software components and the allocation of execution containers to control the system capacity in an elastic manner. For systems conforming to this architectural style, we developed the performance simulator SLAstic.SIM. SLAstic.SIM simulates instances of the Palladio Component Model (PCM) [4] and supports PCM-specific implementations of the runtime configuration operations employed by our SLAstic approach. Simulations can be driven by external workload traces which may have been generated or recorded prior to the simulation.

SLAstic.SIM is used for studying the performance impact of reconfiguration operations, as well as evaluating adaptation strategies and tactics based on realistic workload profiles—online and offline. This paper describes SLAstic.SIM and provides an evaluation of its features and performance.

The remainder of this paper is structured as follows. Sections 2 and 3 describe the underlying concepts of the Palladio Component Model and our SLAstic

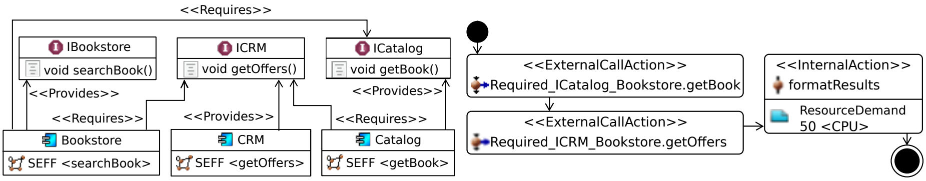
I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 43–58, 2011.

framework. SLAstic.SIM is described in Section 4 and its evaluation follows in Section 5. Related work is discussed in Section 6 before the conclusions are drawn in Section 7.

2 Palladio Component Model

The Palladio Component Model (PCM) [4] is a modeling language for architecture-based performance prediction of C-B software systems. A PCM instance consists of four complementary models providing architectural views to structural as well as performance-relevant behavioral aspects of a C-B software system: (1) a *repository* model, (2) a *system* model, (3) a *resource environment* model, and an (4) *allocation* model. Additionally, *usage* models allow to specify corresponding workloads. Transformations from PCM instances to analytic performance models and simulation models exist, allowing to derive performance indices of interest—e.g., statistical distributions of operation response times and resource utilization. The remainder of this section describes PCM's modeling concepts and related terminology required to understand the remaining parts of this paper. For further details, we refer to the publications on PCM, e.g. [4].

Repository. A PCM repository model contains the type-level specification of available interfaces and components. An interface constitutes a named set of service signatures, as known from object-oriented modeling. Components *provide* or *require* these interfaces. Figure 1(a) illustrates the PCM repository of a Bookstore application which is also used in the evaluation section of this paper.



(a) Repository diagram of the Bookstore (b) RDSEFF of the *searchBook* service

Fig. 1. PCM repository contents of the Bookstore example application

In order to use a PCM instance for performance prediction, the performance-relevant behavior of each service implementation provided by the components must be specified. In this paper, we will limit ourselves to one supported formalism—the *Resource Demanding Service Effect Specification* (RDSEFF). Similar to activity modeling employing the Unified Modeling Language (UML) [5], an RDSEFF specifies a service implementation as a control-flow of actions. PCM distinguishes between internal actions and external call actions—the former being a quantitative specification of the hardware and software resources used by the service; the latter denoting calls to required services. RDSEFFs provide additional features like probabilistic and guarded branches, loops, and operations on variables. Figure 1(b) illustrates the RDSEFF of the Bookstore's *searchBook* service.

System. A PCM system model provides a deployment-independent component-connector view of the system assembly. Components defined in the repository can be (potentially multiply) instantiated as so-called *assembly contexts* and inter-connected using so-called *assembly connectors*—constrained by the interface providing/requiring specification. The services provided and required by the system are delegated to/from the implementing assembly contexts. Figure 2 illustrates the Bookstore's system model.

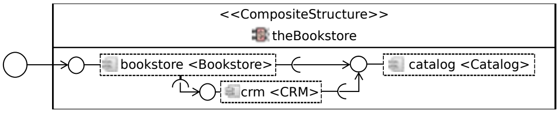


Fig. 2. PCM system diagram of the Bookstore application

Resource Environment. A PCM resource environment model specifies the available resource infrastructure and its performance-relevant characteristics. *Resource containers*, e.g., physical servers, are inter-connected by *linking resources*, e.g., network links. Each resource container is associated with the contained processing resources (e.g., CPU and HDD) which can be demanded in the RDSEFFs. For each resource, the resource environment model contains a specification of the performance-relevant properties of the resources—e.g., capacity, processing rates, throughput, and scheduling disciplines.

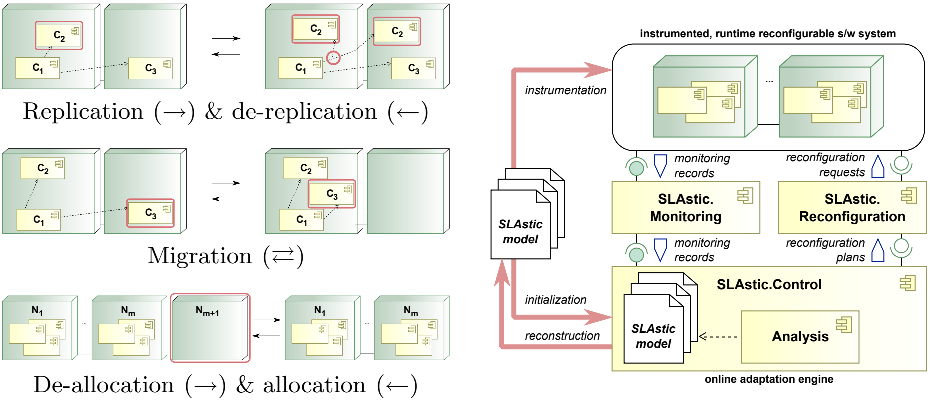
Allocation. A PCM allocation model specifies the deployment of the system's assembly contexts to resource containers. Each of these mappings is modeled as an *allocation context*.

Usage model. A PCM usage model allows to specify closed and open workloads. Probabilistic user behavior is described in an RDSEFF-like formalism including branches, loops, and calls to system-provided services. Closed workloads include the definition of population size and think time; open workloads include the definition of inter-arrival times.

3 SLAstic Approach

As a measure of a system's resource usage economy, resource efficiency is an important quality attribute of software systems. The capacity of software systems is often managed in a static and pessimistic way, causing temporarily underutilized resources, e.g., application servers, during medium or low workload periods.

Our SLAstic [2,3] self-adaptation approach for online capacity management aims to increase the resource efficiency of distributed C-B software systems employing architectural runtime reconfiguration. Architectural models specify the system assembly, deployment, instrumentation, reconfiguration capabilities, performance properties, etc. At runtime, these models are continuously updated and used for online quality-of-service evaluation, e.g., workload forecasting and performance prediction, in order to determine required adaptations and to select



(a) Reconfiguration operations

(b) Framework overview

Fig. 3. SLAstic reconfiguration operations (a) and self-adaptation framework (b)

appropriate reconfiguration plans. Architectural system modeling in SLAstic is based on the hierarchical modeling approach of PCM, as described in the previous Section 2—with a slightly different terminology. For this paper, we assume that *assembly components* are equivalent to the assembly contexts of PCM; *deployment components* to allocation contexts; and *execution containers* to resource containers. The following Sections 3.1 and 3.2 describe the supported architectural reconfiguration operations and the framework architecture.

3.1 Architectural Reconfiguration Operations

In principle, the SLAstic framework, which will be described in Section 3.2, allows arbitrary reconfiguration operations which are defined based on the architectural entities from the SLAstic meta-model. In this paper, we focus on the following five runtime reconfiguration operations that allow to control a system's performance and efficiency properties at runtime in an elastic way. Figure 3(a) illustrates these operations.

1./2. *Replication & de-replication of software components.* The replication operation creates an additional instance of a deployment component on another allocated execution container. Future requests to the services provided by the corresponding assembly component are distributed among the available deployment components. The inverse de-replication operation removes an existing deployment component; newly incoming requests are no longer dispatched to this deployment component instance.

3. *Migration of software components.* The migration operation removes a deployment component instance and creates a new instance of the same assembly component on an allocated execution container—possibly requiring a migration of state.

4./5. *De-allocation & allocation of execution containers.* The allocation operation makes an execution container available for component deployment. The reverse de-allocation operation removes an execution container from the set of allocated containers.

The replication and migration operations both allow to increase system capacity by deploying components to allocated but underutilized execution containers. The de-replication and migration operation can be used to shrink the system capacity by (re)moving deployment components. Operating costs—e.g., caused by power consumption or usage fees in cloud environments—can be saved by de-allocating execution containers.

3.2 Framework Architecture

Figure 3(b) depicts how the concurrently executing SLAstic components for monitoring (SLAstic.Monitoring), reconfiguration (SLAstic.Reconfiguration), as well as adaptation control (SLAstic.Control) are integrated and how they interact with the monitored software system.

The system is instrumented with monitoring probes which continuously collect measurement data from the running system [6]. The SLAstic.Monitoring component provides the monitoring infrastructure and passes the monitoring data to the SLAstic.Control component. The SLAstic.Control component analyzes the current architectural configuration with respect to the monitoring data and, if required, determines an adaptation plan consisting of a sequence of reconfiguration operations. The adaptation plan is communicated to the SLAstic.Reconfiguration component which is responsible for executing the actual reconfiguration operations.

Note, that the SLAstic.Control component takes an architectural, technology-independent view on the software system. The components SLAstic.Monitoring and SLAstic.Reconfiguration translate between architecture and technology. Thus, the SLAstic runtime reconfiguration operations described in the previous Section 3.1 are defined on the architectural entities. In Section 4.2, we present a PCM-specific implementation of these operations.

4 SLAstic.SIM

Section 4.1 gives an overview of SLAstic.SIM's architecture and its integration into the SLAstic framework described in the previous Section 3. In Section 4.2, we describe how the SLAstic reconfiguration operations (Section 3.1) are implemented within SLAstic.SIM using PCM. The execution of the simulation model is described in Section 4.3. Further details on the aspects presented in this section can be found in [7].

4.1 SLAstic.SIM Architecture and Framework Integration

SLAstic.SIM's conceptual architecture and its integration into the SLAstic framework are depicted in Figure 4. For the SLAstic.Control component, SLAstic.SIM

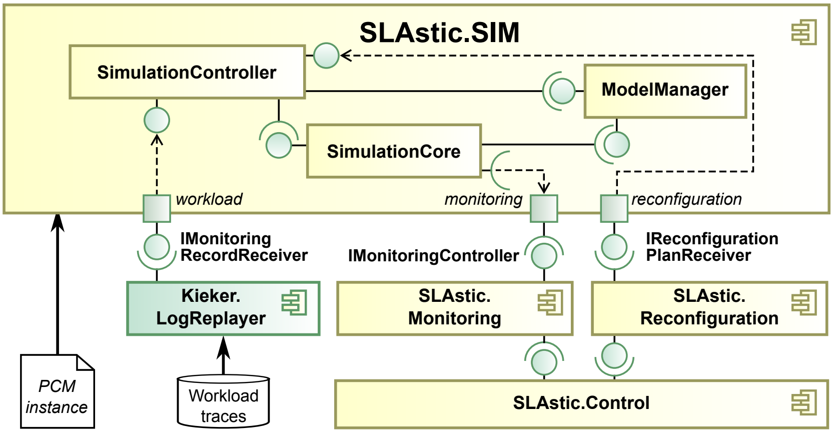


Fig. 4. High-level architecture and framework integration of SLAstic.SIM

emulates a real software system with runtime reconfiguration capabilities. The SimulationController is responsible for the simulation life-cycle and for handling external events. Initially, the input PCM instance is transformed into an internal representation used during simulation and maintained by the ModelManager. The ModelManager includes a ReconfigurationController and a controller for each PCM model, e.g., an AllocationController. The SimulationCore executes the simulation including the generation and execution of internal simulation events. SLAstic.SIM employs the Java-based discrete-event simulation framework Desmo-J[1] [8]. Communication with SLAstic.SIM is possible via the workload, monitoring, and reconfiguration ports. These ports allow to 1) input the workload driving the simulation, 2) receive the performance data generated during simulation, and 3) request reconfigurations to be executed by the simulator, as detailed below. Our monitoring and analysis framework Kieker[2] [6] is used for reading the workload traces and monitoring the simulation data.

Workload. Workload is received from a Kieker.LogReplayer component which reads workload traces from a monitoring log and passes them to registered plugins which implement the IMonitoringRecordReceiver interface—in this case SLAstic.SIM. As these logs typically contain complete control-flow traces and not just the top-level entry calls, the SimulationController filters the incoming workload and delegates it to the SimulationCore.

Monitoring. Currently, SLAstic.SIM includes probes for collecting the following information during simulation:

- *Executions.* Each simulated execution of external calls is monitored with the associated information on the service and assembly context, the resource container, the entry and exit times, as well as the control flow information.

[1] Desmo-J: `http://desmoj.sourceforge.net/`

[2] Kieker: `http://kieker.sourceforge.net/`

- *CPU utilization.* For each CPU of allocated resource containers, the utilization is measured in intervals of 0.5 simulated time units.
- *Active users.* If a call from outside of the system occurs, we increment the user count and write a monitoring record. Upon the return of a call the user count is decremented again and another record is written.

In each case, we defined a Kieker monitoring record type which allows to monitor, analyze, and visualize this data. The monitoring records are passed to the SLAstic.Monitoring component via Kieker's MonitoringController. The monitoring probes are injected using the Google Guice[3] dependency injection framework. This gives the possibility to enable or disable probes between different simulation runs by simply replacing a class's implementation. It is also possible to disable each of these probes separately or to add additional ones.

Reconfiguration. The IReconfigurationPlanReceiver interface makes it possible to send reconfiguration plans (see Figure 5) to the simulator. The plans will be received and checked by the SimulationController and then sent to the ModelManager, which translates them into reconfiguration events. These events will then be simulated by the SimulationCore. A reconfiguration plan consists of one or more reconfiguration operations. These operations are successively applied to the simulation model by the ReconfigurationController. Each operation of a reconfiguration plan is transformed into one or more events, which are scheduled and executed consecutively. If an event fails to execute, the current plan is aborted. The following Section 4.2 details the PCM-specific implementation and execution of the runtime reconfiguration operations.

4.2 PCM-Specific Runtime Reconfiguration Operations

Figure 5 shows the meta-model including the PCM-specific reconfiguration plan and operations.

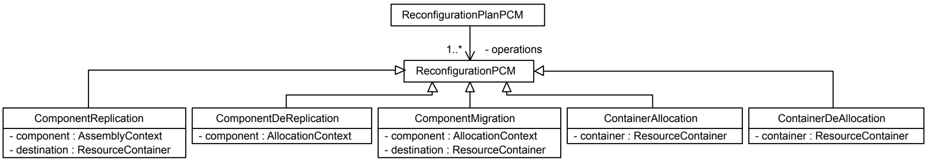

ReconfigurationPlanPCM
1..*
- operations
ReconfigurationPCM
ComponentReplication
- component : AssemblyContext
- destination : ResourceContainer
ComponentDeReplication
- component : AllocationContext
ComponentMigration
- component : AllocationContext
- destination : ResourceContainer
ContainerAllocation
- container : ResourceContainer
ContainerDeAllocation
- container : ResourceContainer

Fig. 5. PCM-specific reconfiguration plan and operations (cf. Figure 3(a))

1. *Component replication.* For the given assembly context, a new allocation context located on the destination container is created and added to the model. The destination container must be allocated prior to the call and must not contain an allocation context for this assembly context.

[3] Google Guice: `http://code.google.com/p/google-guice/`

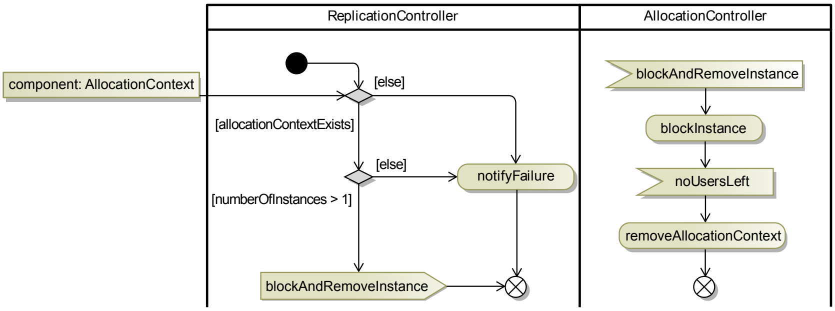


Fig. 6. Activity diagram for the component de-replication operation

2. *Component de-replication.* The existing allocation context is blocked, which means that no new calls are dispatched to this instance. As soon as all running transactions handled by the component are finished, the allocation context is removed from the model. Prior to the request, at least two allocation contexts must exist for the assembly context. The activity diagram in Figure 6 depicts the execution of a de-replication operation within SLAstic.SIM.
3. *Component migration.* The migration is implemented by executing a replication followed by a de-replication operation. Hence, the new allocation context immediately handles new calls while the old allocation context exists until all executing calls are finished.
4. *Container de-allocation.* The container is marked unavailable which means that it cannot be the target of migration or replication operations until it is allocated again. Prior to the request, the resource container to be de-allocated must be allocated and empty—i.e., it must not contain any allocation context.
5. *Container allocation.* The container is marked available which means that it can be the target of migration or replication operations. The operation can be executed if the resource container exists in the simulation model and is not allocated at that time. Upon completion, components can be replicated or migrated to it. Initially, exactly those resource containers from the resource environment being associated with at least one deployment context are marked as allocated.

4.3 Simulation

Desmo-J offers two styles of modeling [8]: process-based and event-based. We chose to use the event-based model as all our state changes in the simulation model are instantaneous and there would be no real life-cycle. Below, we give a brief overview of the generation and execution of control-flows.

Control-Flow Generation. On each external call from the input workload, the complete control-flow chain is generated. This is done by traversing and evaluating the corresponding RDSEFF. Call enter and return events are generated for each ExternalCallAction.[4] For each InternalAction, an internal action event is produced, containing the resource demands of the input InternalAction. BranchActions are evaluated by deciding which transition to take and traversing the transition's body. LoopActions are evaluated similarly by determining the iterations and then traversing the body for each iteration. The result is a list of Desmo-J events which are scheduled consecutively.

Execution of Control-Flow Chains. On occurrence of an external call, the allocation contexts for the corresponding assembly contexts are determined and one of these is selected based on the uniform probability distribution. The resource demands of internal actions are mapped to the corresponding resources of the current resource container. Each of these resources has a scheduler. Currently, we support hard drives scheduled by a first-come/first-served strategy and CPU usage by processor sharing. These components are also replaceable by implementing the corresponding interface.

5 Evaluation

Employing an example application, the evaluation in this section demonstrates SLAstic.SIM's performance and features by comparing the duration of simulation runs with the default PCM simulator SimuCom [9], as well as by two additional scenarios under varying workload intensity with and without the execution of reconfigurations. Section 5.1 describes the evaluation methodology. The evaluation scenarios follow in Sections 5.2–5.4. Details on SimuCom can be found in Section 6 (related work).

5.1 Methodology

The following Sections 5.1–5.1 describe the example system, the workload trace generation, and the hardware and software setup.

Example System. The Bookstore application, used in all evaluation scenarios, provides a single searchBook service which allows to search for books in a catalog (see also Section 2). The application consists of three software components—a front-end (Bookstore), a catalog (Catalog), and a customer-relationship management (CRM) component. A call to the searchBook service results in a single deterministic trace shown in the sequence diagram in Figure 7. The diagram was created employing Kieker, based on monitoring data from a simulation run. We created a PCM instance for the Bookstore application with the following models:

[4] ExternalCallAction, InternalAction, BranchActions, and LoopAction are classes in the PCM metamodel (see Figure 1(b)).

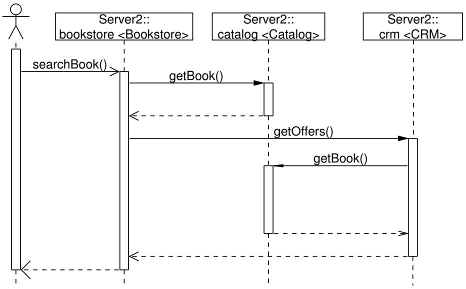


Fig. 7. Reconstructed sequence diagram of a Bookstore trace

1. *Repository.* The PCM repository model of the Bookstore and the RDSEFF describing the searchBook have already been shown in Figures 1(a) and 1(b). For each of the three components, we defined a corresponding interface in the repository. The RDSEFF of searchBook consists of external calls to the getBook and getOffers services, followed by a resource demand of 50 CPU units. The service getOffers consists of an external call to getBook, followed by a resource demand of 20 CPU units. The service getBook simply contains a resource demand of 15 CPU units. The resource demands have been chosen to yield a response time of 100 time units for a call to searchBook without resource contention.
2. *System.* The Bookstore system consists of three assembly contexts (see Figure 2)—one for each repository component. The only externally provided interface is the IBookstore interface. So the only service that is visible to users is searchBook.
3. *Resource environment.* The resource environment consists of two resource containers: Server1 and Server2. Each of them has a single CPU. The CPU's processing rate varies between the evaluation scenarios, as detailed in the respective sections.
4. *Allocation.* Initially, each assembly context is mapped to resource container Server 2. Server 1 is empty.

Generation of Workload Traces. For the scenarios, it was required to generate workload traces with constant and varying workload:

- *Constant Workload.* Constant workload was generated by a script writing Kieker monitoring logs in comma-separated value (CSV) file format based on a constant inter-arrival time.
- *Varying Workload.* Varying workload was generated using Apache JMeter.[5] We implemented a timer[6] that takes a function of time as its input. This allows to modulate inter-arrival times. The data was written to a CSV file and converted into a Kieker monitoring log in CSV format.

[5] Apache JMeter: `http://jakarta.apache.org/jmeter/`

[6] JMeter Function Timer: `http://code.google.com/p/delayfunction/`

Table 1. Hardware and software setup used to run the evaluation

CPU	Intel Core i5, hyper-threading enabled
RAM	4 GB
OS	Ubuntu Generic Linux kernel 2.6.32-22 SMP
Java	Sun Java Version 1.6.0_20
Heap space	1GB for SLAstic.SIM, 2GB for SimuCom 3.0

Hardware and Software Setup. The simulations were executed in the hardware and software environment listed in Table 1.

5.2 Scenario 1: Constant Workload Intensity

This scenario compares the duration of simulation runs executed with SLAstic.SIM and SimuCom.

Setting. The input workload for SimuCom was specified using the PCM workload specification. It was modeled as an open workload with an inter-arrival time of 0.1 units. The maximum simulation time was set to 1000 time units. The aim was to provide a workload which does not overload the system with an increasing number of running transactions. In order to produce a reasonable CPU utilization, the CPUs' processing rate was set to 1000 ticks per simulated time unit.

We only measured the duration of the simulation. Particularly, for SimuCom we excluded the time required for code generation and compilation, and for SLAstic.SIM we omitted the static initialization overhead.

Results. As expected, the simulated response time of the system was 0.1 for both, SLAstic.SIM and SimuCom.

Table 2 lists statistics for the duration in (milliseconds) of 50 simulation runs executed with SimuCom and SLAstic.SIM. Given this PCM instance, we can see that SLAstic.SIM and SimuCom are comparable regarding the overall duration of the simulation (SLAstic.SIM being slightly faster).

Table 2. Statistics for the duration (ms) of 50 simulation runs

	Min.	Median	Mean	Max.	Dev.
SimuCom	6434	7179	7199	7873	287.24
SLAstic.SIM	4864	5325	5333	5833	161.25

5.3 Scenario 2: Varying Workload without Reconfiguration

This scenario demonstrates the performance simulation driven by a trace of varying workload intensity without executing runtime reconfigurations.

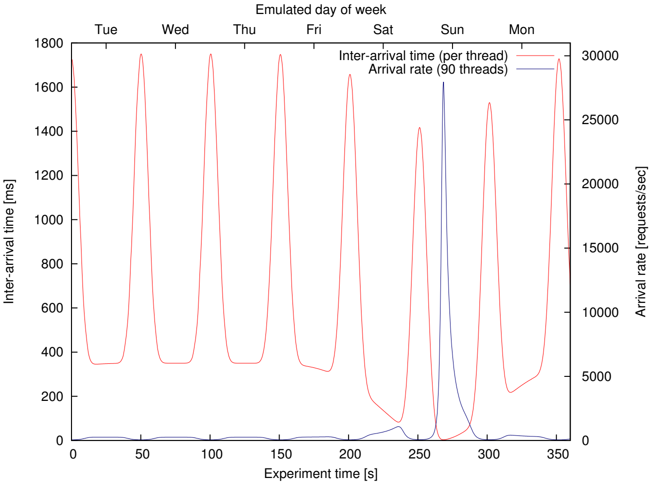
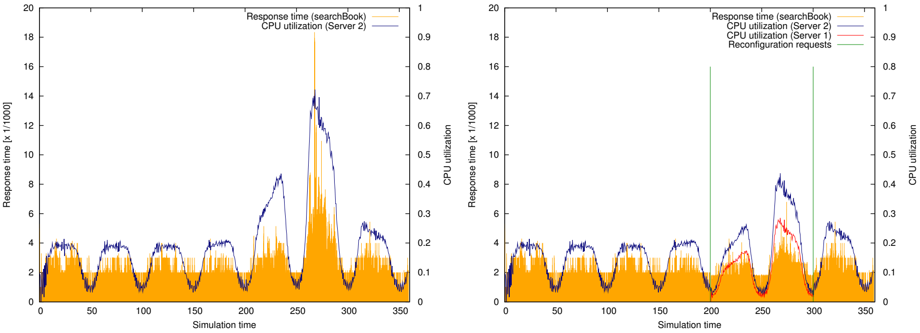


Fig. 8. Workload intensity (Scenarios 2&3)



(a) Scenario 2 (no reconfiguration)

(b) Scenario 3 (reconfiguration)

Fig. 9. Response times and CPU utilization (Scenarios 2 and 3)

Setting. The input workload function for this scenario was modeled to resemble one week of workload—with a peak intensity on the weekend. Such workload patterns can be observed in many real-world web-based systems. The function was sampled over 360 seconds by 90 JMeter threads, producing 68,653 calls. Figure 8 shows the inter-arrival time function (and the corresponding arrival rates) used as input. Opposed to the previous scenario we set the CPUs' processing rate to 100,000 ticks per simulated time unit.

Results. The simulated system behaved as expected. A plot of the simulation results is given in Figure 9(a). During periods with low workload intensity, the CPU utilization is between five and ten percent, and the response times are below 0.002 simulated time units. Increasing the CPU load also increases the response times of the service requests. A peak was reached at a simulation time of approximately 270 with a CPU load of 70% and a response time of nearly 0.018 time units. The average duration of 10 simulation runs was 18.6 seconds.

5.4 Scenario 3: Varying Workload with Reconfiguration

This scenario demonstrates the performance simulation driven by a trace of varying workload intensity including runtime reconfigurations.

Setting. We used the varying workload trace from Scenario 2 (see Section 5.3). During the time period with high workload intensity, we requested runtime reconfigurations in order to increase system capacity and improve responsiveness. We implemented SLAstic.Control components that requested the following two runtime reconfiguration plans at fixed simulation times:

1. The reconfiguration plan requested after 200 time units consists of: An allocation of Server1 followed by a subsequent replication and migration of the components CRM and Catalog respectively. Both the replication and migration have the newly allocated resource container Server1 as its destination.
2. The inverse reconfiguration plan requested after 300 time units consists of the migration of component Catalog back to Server2, the de-replication of component CRM, and the subsequent de-allocation of Server1.

The SLAstic.Control component maintains a runtime model of the PCM instance during the simulation run which is updated according to the executed system reconfigurations.

Results. A plot of the response times and CPU utilizations is shown in Figure 9(b). We can see that due to the reconfiguration both, response times and CPU utilizations, can be reduced on the simulated weekend. The dependency graph in Figure 10, which was generated from the monitoring data collected during the complete simulation run, shows that the calls are distributed among the two resource containers. The average duration of 10 simulation runs was 18.1 seconds.

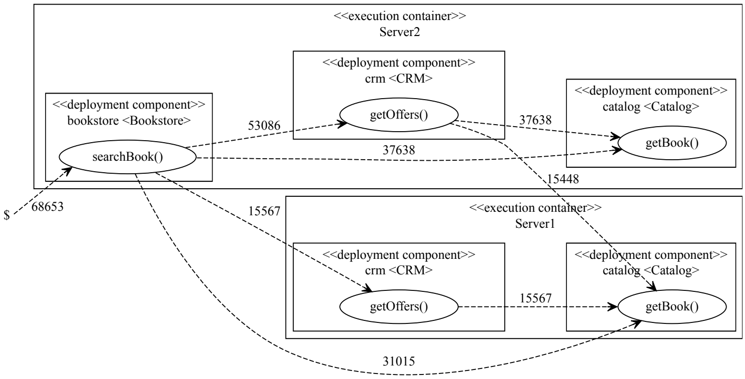


Fig. 10. Operation dependency graph with calling frequencies (Scenario 3)

6 Related Work

Performance evaluation of computer systems is a classical and well-studied domain for simulation, e.g., based on queueing (network) models [10,11,8]. For example, Java Modeling Tools (JMT) [12] is a tool suite for modeling and analyzing extended queueing networks. JMT includes the discrete-event simulator JSIMengine. In addition to probabilistic (multi-class) open and closed workloads, simulations can be driven by workload traces provided as log files. Like SLAstic.SIM, it is possible to use JSIMengine within external applications.

In our work, we focus on the performance simulation of software systems using performance meta-models. Simulation approaches exist for different kinds of architectural styles and corresponding models. Examples of approaches based on the UML SPT [13] profile for Schedulability, Performance, and Time are ArgoSPE [14], CB-SPE [15]. Cortellessa et al. [16] proposed an approach for the simulation-based performance analysis of UML 2 models. Bause et al. [17] proposed an approach for simulating models of service-oriented architectures (SOAs) using process chain models and the OMNeT++[7] network simulation framework.

Comparison to SimuCom. The work most related to SLAstic.SIM is SimuCom, the simulator for PCM instances of C-B software architectures without runtime reconfiguration capabilities. SimuCom is integrated into the PCM modeling environment SimuBench [4], developed as part of the Palladio research project[8]. In terms of simulation correctness and simulator performance—for simulations without reconfiguration and restricted to the PCM modeling features supported by SLAstic.SIM—we consider SimuCom the reference implementation. Simulations with SimuCom are driven by PCM usage models of closed or open workloads, as described in Section 2. SLAstic.SIM could be easily extended to allow these kinds of workload models. In Section 5.2, we have used a generated workload trace equivalent to a PCM open workload usage model. Currently, SLAstic.SIM does not support the following PCM features: Stochastic expressions (except for constants) and middleware models, as implemented by SimuCom.

Like SLAstic.SIM, SimuCom is implemented employing Desmo-J. The simulation code is completely generated from a PCM instance employing model-to-code (M2C) transformation prior to simulation start. This approach is well-suited for software architectures, which are not reconfigured during simulation. However, it is not trivial to extend SimuCom's M2C transformation by simulation support of runtime reconfigurable PCM instances. This was one of the main reasons for us to develop a new simulator for PCM models with runtime reconfiguration support, following an interpretive simulation approach. Another reason was that the SimuCom simulations are only executable in an OSGi[9] environment like Eclipse.

[7] OMNeT++ web site: `http://www.omnetpp.org/`

[8] Palladio project: `http://sdq.ipd.kit.edu/research/palladio_research_project/`

[9] OSGi: `http://www.osgi.org`

7 Conclusions

This paper presented SLAstic.SIM, a performance simulator for runtime reconfigurable, C-B software architectures. SLAstic.SIM is able to simulate instances of the Palladio Component Model (PCM) driven by external workload traces which may have been generated or recorded prior to the simulation. Additionally, it supports the proposed PCM-specific implementation of the SLAstic runtime reconfiguration operations aiming for increased resource efficiency: migration and (de-)replication of software components, as well as (de-)allocation of execution containers. The evaluation demonstrated SLAstic.SIM's performance and features employing a small sample application. In a simulation scenario under constant workload and without reconfiguration, SLAstic.SIM was slightly faster than SimuCom. Two additional scenarios showed the capability to drive simulations by varying workload intensity profiles and the possibility to simulate the afore-mentioned runtime reconfigurations.

In our future work, we will continue to improve and extend SLAstic.SIM's features and performance. The most important PCM feature to be implemented is support for the stochastic expressions allowing to model parametric resource demands, loops etc. Also, SLAstic.SIM currently implements an idealized view on the execution of reconfiguration operations: when executed, they consume no simulation time. For example, execution containers and replicated component instances are available without simulating delays. We plan to add the interpretation of corresponding model completions. Likewise, other QoS properties, such as the reliability of execution containers, may be modeled and simulated. We plan to implement alternative strategies for dispatching requests among replicated components, as well as additional runtime reconfiguration operations, e.g., replacing the implementation of component types. Moreover, we will use SLAstic.SIM to simulate more complex PCM instances with workloads derived from monitoring data of production systems. Also, the use of SLAstic.SIM for online simulation will be further investigated.

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4(2), 1–42 (2009)
2. van Hoorn, A., Rohr, M., Gul, A., Hasselbring, W.: An adaptation framework enabling resource-efficient operation of software systems. In: Proc. Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP 2009), pp. 41–44. ACM, New York (2009)
3. van Hoorn, A.: Online Capacity Management for Increased Resource Efficiency of Software Systems. PhD thesis, Dept. Comp. Sc., Univ. Oldenburg, Germany (2011) (work in progress)
4. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009)
5. Object Management Group: UML 2.3 Superstructure Specification (May 2010)

6. van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Dept. Comp. Sc., Univ. Kiel, Germany (November 2009)
7. von Massow, R.: Performance simulation of runtime reconfigurable software architectures, Diploma Thesis, Univ. Oldenburg, Germany (April 2010)
8. Page, B., Kreutzer, W. (eds.): The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java, 1st edn. Shaker Verlag, Aachen (2005)
9. Becker, S.: Coupled Model Transformations for QoS Enabled Component-Based Software Design. PhD thesis, Dept. Comp. Sc., Univ. Oldenburg, Germany (2008)
10. Jain, R.: The Art of Computer Systems Performance Analysis. Wiley & Sons, Chichester (1991)
11. Banks, J. (ed.): Handbook of Simulation: Modelling, Estimation and Control. Wiley & Sons, Chichester (1998)
12. Bertoli, M., Casale, G., Serazzi, G.: JMT: Performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. 36(4), 10–15 (2009)
13. Object Management Group: UML Profile for Schedulability, Performance, and Time (January 2005)
14. Gomez-Martinez, E., Merseguer, J.: A software performance engineering tool based on the UML-SPT. In: Proc. Int. Conf. on Quantitative Evaluation of Systems (QEST 2005), p. 247. IEEE, Los Alamitos (2005)
15. Bertolino, A., Mirandola, R.: CB-SPE tool: Putting component-based performance engineering into practice. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 233–248. Springer, Heidelberg (2004)
16. Cortellessa, V., Pierini, P., Spalazzese, R., Vianale, A.: MOSES: MOdeling software and platform architEcture in UML 2 for simulation-based performance analysis. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 86–102. Springer, Heidelberg (2008)
17. Bause, F., Buchholz, P., Kriege, J., Vastag, S.: A framework for simulation models of service-oriented architectures. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 208–227. Springer, Heidelberg (2008)

Aspect-Connectors to Support the Evolution of Component-Based Product Line Architectures: A Comparative Study

Leonardo P. Tizzei[*] and Cecília M.F. Rubira[**]

University of Campinas, Campinas, SP, Brazil
{tizzei,cmrubira}@ic.unicamp.br

Abstract. Software Product Line architects are concerned not only with traditional software architecture issues, but also with variation mechanisms that support diversity of products. A variation point may be scattered over various implementation elements (i.e. components and connectors) undermining product line architecture evolution. Aspect-connectors support the modularization of variation points by integrating aspects and components concepts. This work quantitatively evaluates to what extent aspect-connectors supports evolution of component-based product line architectures by means of a comparative study against a pure object-oriented component model. This study considered five evolution measures: scattering of variation points, scattering of features, tangling of features, change impact on components, and efferent coupling between components. The results have shown evidences that aspect-connectors can provide effective support for product line architecture evolution.

1 Introduction

Product Line Architectures (PLAs) are key assets to support Software Product Line (SPL) evolution [16]. A PLA should explicitly provide variation mechanisms to support software variability among software products. Software variability can be modeled using features, which are distinctive end-user-visible qualities or characteristics [10]. Ideally, feature variabilities should be explicitly mapped to PLA variabilities [16] thus achieving traceability between both models.

Software components (components for short) are a common way to design and implement PLAs [15]. Components promote encapsulation and explicit their services and dependencies thus supporting PLA evolution. However, they have limited capacity to handle variability. The integration of conditional compilation and software components increases support for variability, but it can lead to the scattering of architectural variation points which harms PLA evolution. The combined use of aspects and components arguably enhance separation of concerns in PLAs and, consequently, its evolvability. Dias et al. [7] proposed the use of aspect-connectors for coping with variability evolution. As variation points

[*] L.P. Tizzei is supported by Capes - Brazil.

[**] C.M.F. Rubira is partially supported by CNPQ - Brazil.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 59–66, 2011.

can be a crosscutting concern on PLAs, aspect-connectors uses aspect-oriented design mechanisms with the aim of modularizing variation points.

It is interesting for organizations to know which design and implementation technique supports PLA evolution, but to the best of our knowledge there is a lack of comparative studies. This work provides a comparative study that quantitatively assesses the effective support for evolution of PLAs of two design and implementation techniques: the integration of aspect-connectors and components against a pure (i.e. non-aspect-oriented) component approach. In order to perform this comparative study, we have adopted a representative component model, called COSMOS* [6], and its extension to support aspect-connectors, called COSMOS*-VP [7]. The support for evolution of both approaches were evaluated considering five measures: (i) scattering of variation points, (ii) scattering of features, (iii) tangling of features, (iv) change impact on components, and (iv) coupling between components. The results have shown evidences that the use of aspect-connectors can provide effective support for PLA evolution.

The remainder of this paper is organized as follows: Section 2 presents the two component models involved in this study, namely COSMOS* and COSMOS*-VP. Section 3 describes the empirical settings of our study, and Section 4 presents the analysis for all measures and points out study limitations. Section 5 presents some works related to this one. Finally, in Section 6 we draw the conclusions.

2 Background

COSMOS* Component Implementation Model. COSMOS* component implementation model [6] (COSMOS* model for short) specifies explicit provided and required interfaces, it can also be deployed independently, and is subject to composition by third parties. Thus, the COSMOS* model is compliant with Szyperski's definition of software component [15].

COSMOS* defines five sub-models, which address different perspectives of component-based systems: (i) the specification model specifies the external behavior of a component and is composed by required and provided interfaces; (ii) the implementation model explicitly separates the provided and required interfaces from the implementation; (iii) the connector model specifies the link between components using connectors; (iv) composite components model specifies high-granularity components; and (v) system model defines a software component which can be straightforwardly executed.

COSMOS*-VP Model. COSMOS*-VP model uses aspects to extend COSMOS* model. Aspect-Oriented Programming [11] is an approach that aims to modularize the crosscutting concerns. These concerns are widely-scoped properties and usually crosscut several modules in the software system. Aspects are the abstractions used to encapsulate otherwise crosscutting concerns. In this paper, the notion of concern is considered equivalent to feature.

The COSMOS*-VP model extends COSMOS* model in order to support separation of concerns and encapsulation of variation points. Components implemented using COSMOS*-VP are separated into two categories: base-level components and aspect-level components. Base-level components are similar to COSMOS* components, but they also expose joinpoints using crosscutting programming interfaces (XPIs) [9]. By means of XPIs, it is possible for an aspect-level component to change the behavior of a base-level component. Aspect-level components encapsulate crosscutting concerns and modify the behavior of at least one base-level component. Aspect-level components do not specify which base-level components they advise, because it would strongly couple them to the base-level components. Instead, aspect-connectors bind aspect-level and base-level components. An aspect-connector provides mechanisms to mediate the binding of aspect-level components to the XPIs of base-level. This binding is necessary in order to provide the non-mandatory features of aspect-level components to base-level components. We refer to Dias et al. [7] for more details on COSMOS*-VP.

3 Empirical Settings

The objective of this comparative study is to assess quantitatively to what extent aspect-connectors support PLA evolution. In this study, we compare two design and implementation techniques: (i) a pure component-based model; and (ii) the use aspect-connectors, a hybrid approach using components and aspects.

Based on this objective, we state the hypothesis we need to evaluate:

> H_0: *there is no difference in PLA evolution with respect to the design and implementation technique.*
> H_1: *the aspect-component approach supports PLA evolution better than the pure-component technique.*

In order to evaluate these hypotheses, we have analyzed a target SPL during evolution and collected five measures of evolution [4]: (i) scattering of variation points [13], (ii) scattering of features [13], (iii) tangling of features [13], (iv) change impact on components [17], and (v) efferent coupling between components [5]. If one technique has significant better results than the other for at least one metric, then it will be possible to reject H_0. However, if aspect-component approach does not succeed in achieving the best results for all metrics, then it will not be possible to accept H_1. Regarding the design of this study, as we intended to compare two techniques, pure-component and aspect-component, in a pair-wise way, thus the Wilcoxon signed rank test [14] was suitable.

The original OO and AO implementations of the target SPL were the input for our study. The pure-component implementation was refactored from the original OO implementation, and aspect-component implementation was refactored from the original AO implementation. We refer to the hybrid approach that uses components, aspects, and aspect-connectors as aspect-component for short, and we refer to component-based approach that does not use aspects as pure-component. We have adopted COSMOS* model to implement the pure-component SPL and COSMOS*-VP model to implement aspect-component SPL.

After refactoring the original implementations, the following steps were executed:

- *Step 1.* Evolve the refactored pure-component and aspect-component implementations according to evolution scenarios (source code available on [1]).
- *Step 2.* Collect evolution metrics, namely scattering of variation points, scattering of feature, feature tangling, change impact, and efferent coupling between components, for eight pure-component and aspect-component releases;
- *Step 3.* Compare the results of aspect-component against pure-component implementation.

In order to exemplify and evaluate our solution, we present a software application, called MobileMedia [8], which is a SPL for mobile applications that manipulates photo, music, and video on mobile devices, such as mobile phones. The system uses various technologies based on the Java ME platform. It has two implementations with the same functionalities, but implemented with different approaches: one uses AO programming and has approximately 12 KLOC and the other uses only OO programming and has 11 KLOC. MobileMedia endured seven evolution scenarios, which led to eight releases. The scenarios comprise different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. We refer to MobileMedia study webpage [1] for more details on the evolution scenarios.

During the execution of this study, some design and implementation decisions may have influenced the results. For instance, all SPLs (i.e. pure-component and aspect-component MobileMedia) follow the same architecture pattern, namely the Model-View-Controller (MVC). Regarding the feature implementation of the aspect-component SPL, optional and alternative features are encapsulated by a component and its implementation relies on aspects and auxiliary classes. Thus, every time an optional or an alternative feature is included, at least one component is created in the aspect-component implementation. The implementation of mandatory features in the aspect-component does not use aspects and is realized by multiple modules. In pure-component SPLs, the implementation of all types of features is realized by multiple modules.

Regarding variability programming mechanisms, we have used conditional compilation and aspects. On pure-component implementations, conditional compilation defined which parts of code should be compiled or not, based on the value of preprocessor variables. All the above-mentioned design and implementation decisions were made following the original MobileMedia study [8].

During the execution of *Step 2*, data collection was supported by tools and the execution of *Step 3* was supported by a statistical analysis tool to double-check the correctness of our calculations.

4 MobileMedia Study

Scattering of Features. Concerns are considered equivalent to features in this study. Figure 1(a) presents the scattering of feature on aspect-component and pure-component implementations. Aspect-component approach has the best results for scattering of features, because AOP provides mechanisms to modify the behavior of components without changing them. The main difference regarding

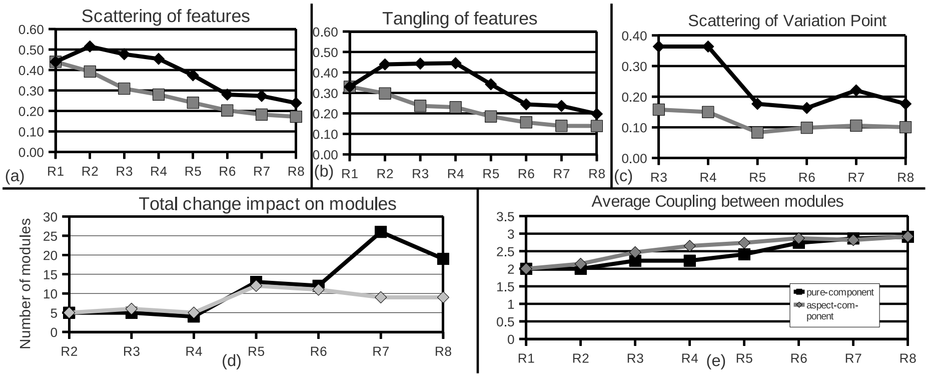


Fig. 1. Number of affected modules during PLA evolution

scattering of feature of aspect-component and pure-component implementations is how non-mandatory features were implemented, because mandatory features were implemented likewise in both implementations. On aspect-component implementation, non-mandatory features were implemented in separate aspect-level components, which encapsulate feature implementation. On pure-component implementation, non-mandatory features were implemented in multiple components and connectors by means of conditional compilation.

Tangling of Features. Figure 1(b) shows the tangling of feature on aspect-component and pure-component implementations. Aspect-component approach achieved to untangle features better than pure-component, because, on aspect-component implementation, non-mandatory features were implemented by separated aspect-level components, whereas on pure-component implementation these features were implemented on existing components.

Scattering of Variation Points. Figure 1(c) shows the scattering of variation points for each release of both aspect-component and pure-component implementations of MobileMedia SPL. The overall results show that aspect-component achieved to minimize the scattering of variation points when compared to pure-component approach. All variation points on aspect-component PLAs were implemented on aspect-connectors, because aspect mechanisms, such as XPIs, allow to intercept decision points on components and encapsulate the implementation of these decisions on aspect-connectors. On pure-component PLAs these decisions were implemented on multiple components and connectors.

Change Impact Analysis. The greater the number of modules affected (i.e. changed, added or removed), the greater is the impact on the architecture. The change impact metrics were collected comparing each release to its previous one (e.g. comparing R2 to R1).

Figure 1(d) shows the results for total change impact on modules. The inclusion of mandatory features (R2 and R3) and the inclusion of optional features (R4-R6) on aspect-component PLAs caused the addition of new modules that

Table 1. Results from the Wilcoxon paired test for all metrics

Metric	N	W^+	p-value	Best approach	Statistically significant
Scattering of variation points	6	21	0.03	aspect-component	yes
Scattering of feature	7	28	0.02	aspect-component	yes
Tangling of feature	7	28	0.02	aspect-component	yes
Change impact on components	6	11	0.28	aspect-component	no
Efferent coupling between components	7	2	0.05	pure-component	no

were added to implement the included features. On pure-component PLAs the change impact was mostly due to modifications on existing modules to implement the included features. In the last two releases (R7 and R8), aspect-component approach achieved a much lower number of modules impacted. Aspect-connectors facilitated the evolution of aspect-component PLA by isolating from components the implementation that supports the variability decisions of the variation points.

Efferent Coupling Analysis. Figure 1(e) illustrates the results for efferent coupling between modules (just coupling, for short). The use of XPIs helped to decouple aspect-components, but it was not enough to support loosely coupled modules as pure-component approach. In this study, an aspect-component implements a non-mandatory feature by creating a new component which encapsulates the implementation of this feature and then, via aspect-connectors, it changes the behavior of other components. As it has to change the behavior of other components, the semantic dependency between components is strong and may become a syntactical dependency. On pure-component implementation, although features are implemented in multiple components, each component has its role in the implementation of feature (for instance, `PresentationMgr` component displays functions to end user, `PersistenceMgr` component persists metadata related to media) and the dependency among them is not strong.

Testing Hypothesis. Table 1 describes the results of the Wilcoxon test for all metrics. Due to limitations of space, we refer to MobileMedia study webpage [1] for details on the calculations. Based on results presented in Table 1, we can test the null hypothesis (H_0). Three out five measures show statistically significant differences between aspect-component and pure-component approaches, which provide us evidence to reject the null hypothesis. As two out five metrics are not statistically significant, and one of them suggests that pure-component approach is better than aspect-component approach, we do not have enough evidence to either accept or reject H_1. Further studies are necessary to test H_1.

Study Limitations. We have identified the following threats to validity: (i) bias while refactoring the original MobileMedia and evolving them;(ii) MobileMedia might not be representative of industrial SPLs. Regarding risk (i), we were strict in following exactly the same design and implementation decisions of the original MobileMedia study [8]. In addition, all MobileMedia implementations provide the same functionalities and were similarly tested. Regarding risk (ii), even though MobileMedia is a small SPL, it is heavily-based on industry-strength technologies, and it is a complex SPL as it can derive 200 products. Furthermore, it has been extensively used and evaluated in previous research (e.g. [7,8]).

5 Related Work

Figueiredo et al. [8] presented a case study which quantitatively compares the positive and negative impact of using AOP to support SPL evolution. Two SPLs were involved in their study: MobileMedia and BestLap. Both were implemented using OO and AOP. The authors evaluate the support for evolution of OO and AOP considering cohesion, coupling, change impact, and feature dependency metrics. They conclude that AOP supports the implementation of optional and alternative features. However, AOP also increases coupling and cohesion. In contrast to their study, we evaluated two component-based approaches. Furthermore, we have used a different metric suite.

Apel et al. [2] compared two aspect-oriented approaches, namely aspect languages (e.g. AspectJ, AspectC++) and collaboration languages (e.g. Classbox/J, Jiazzi) and their impact on crosscutting modularity. The authors quantitatively compared the use of each approach and based on this comparison, they established guidelines for programmers on when to use each approach. In contrast to our study, the authors do not focus on PLA evolution.

Kvale et al. [12] presented a case study that investigated if AOP could help to build more easy-to-change COTS-based systems than OO. They compare how much effort is necessary to: (i) integrate a COTS-based system; and (ii) replace COTS after integration. They concluded that fewer classes need to be changed when adding and replacing COTS using AOP. They evaluated the benefits of using aspects to implement the glue-code between COTS and the system, while we evaluated the application of AOP in a SPL context.

Baniassad et al. [3] presented an inquisitive study in which they identified kinds of crosscutting concerns that impact on software developers the strategies developers use to cope with these concerns. Furthermore, the study compared whether the use aspect-oriented approaches enable developers to better work with crosscutting concerns. In order to perform this study, the authors tracked eight software developers from academia and industry which were developing a software system. Each developer was evolving a different software system.

6 Conclusions and Future Work

The main contribution of this paper is a study which compares an aspect-component approach against a pure component-based approach. The results have shown evidences that the aspect-component approach has positive influence in supporting PLA evolution. The aspect-component approach achieved significant better measures for scattering of variation points, feature scattering, and tangling of feature. It also achieved better measures for change impact, not statistically significant though. Results for coupling metrics were similar to pure-component approach, but slightly worse.

References

1. MobileMedia comparative study webpage, http://www.ic.unicamp.br/~tizzei/mobilemedia/ecsa2011/ecsa2011.html
2. Apel, S., Kästner, C., Kuhlemann, M., Leich, T.: Pointcuts, advice, refinements, and collaborations: similarities, differences, and synergies. Innovations in System and Software Engineering 3, 281–289 (2007)
3. Baniassad, E.L.A., Murphy, G.C., Schwanninger, C., Kircher, M.: Managing crosscutting concerns during software evolution tasks: an inquisitive study. In: AOSD 2002: Proc. of the 1st Intl. Conf. on Aspect-Oriented Software Development, pp. 120–126. ACM, New York (2002)
4. Brcina, R., Bode, S., Riebisch, M.: Optimisation process for maintaining evolvability during software evolution. In: ECBS 2009: Proc. of the IEEE Intl. Conf. and Workshop on the Engineering of Computer Based Systems, pp. 196–205. IEEE Computer Society, Los Alamitos (2009)
5. Chidamber, S., Kemerer, C.: A metrics suite for OO design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
6. da Silva Jr., M.C., de Castro Guerra, P.A., Rubira, C.M.F.: A java component model for evolving software systems. In: Intl. Conf. on Automated Software Engineering, p. 327. IEEE Computer Society, Los Alamitos (2003)
7. Dias, M., Tizzei, L., Rubira, C.M.F., Garcia, A., Lee, J.: Leveraging aspect-connectors to improve stability of product line variabilities. In: 4th Intl. Workshop on Variability Modelling of Software-Intensive Systems, pp. 21–28 (2010)
8. Figueiredo, E., Camacho, N., Monteiro, C.S.M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: Proc. of the Intl. Conf. of Software Engineering (2008)
9. Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular software design with crosscutting interfaces. IEEE Softw. 23(1), 51–60 (2006)
10. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis. Technical Report CMU/SEI-90-TR-21, CMU/SEI (1990)
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
12. Kvale, A.A., Li, J., Conradi, R.: A case study on building cots-based system using aspect-oriented programming. In: Proc. of the 2005 ACM Symposium on Applied Computing, pp. 1491–1498. ACM, NY (2005)
13. Riebisch, M., Brcina, R.: Optimizing design for variability using traceability links. In: ECBS 2008: Proc. of the 15th Annual IEEE Intl. Conf. and Workshop on the Engineering of Computer Based Systems, pp. 235–244. IEEE Computer Society, Washington, DC, USA (2008)
14. Siegel, S., Castellan Jr., N.J.: Nonparametric statistics for the behavioral sciences, 2nd edn. McGraw-Hill, New York (1988)
15. Szyperski, C.: Component Software. Addison-Wesley, Reading (2002)
16. Thiel, S., Hein, A.: Systematic integration of variability into product line architecture design. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 130–153. Springer, Heidelberg (2002)
17. Yau, S., Collofello, J.: Design stability measures for software maintenance. IEEE TSE 11(9), 849–856 (1985)

Verifying Composite Service Transactional Behavior with EVENT-B

Lazhar Hamel[1], Mohamed Graiet[1], Mourad Kmimech[1],
Mohamed Tahar Bhiri[1], and Walid Gaaloul[2]

[1] MIRACL, ISIMS, TUNISIA
lazhar.hamel@gmail.com, mohamed.graiet@imag.fr,
mkmimech@gmail.com, Tahar_bhiri@yahoo.fr
[2] Computer Science Department Télécom SudParis
walid.gaaloul@it-sudparis.eu

Abstract. A key challenge of Web Service (WS) composition is how to ensure reliable execution. Due to their inherent autonomy and heterogeneity, it is difficult to reason about the behavior of service compositions especially in case of failures. Therefore, there is a growing interest for verification techniques which help to prevent service composition execution failures. In this paper, we present a proof and refinement based approach for the formal representation, verification and validation of Web Services transactional compositions using the Event-B method.

Keywords: web service composition, Event-B, transactional web service, proof, refinement, verification.

1 Introduction

Web services are emergent and promising technologies for the development, deployment and integration of applications on the internet. One interesting feature is the possibility to dynamically create a new added value service by composing existing web services, eventually offered by several companies. Due to the inherent autonomy and heterogeneity of web services, the guarantee of correct composite services executions remains a fundamental problem issue. An execution is correct if it reaches its objectives or fails properly according to the designer's requirement or users needs. The problem, which we are interested in, is how to ensure reliable Web services compositions. By reliable, we mean a composition which all executions are correct.

Our work deal with the formal verification of the transactional behavior of web services composition. In this paper, we propose to address this issue using proof and refinement based techniques, in particular the Event-B method [1] used in the RODIN platform [2]. Our approach consists on a formalism based on Event-B for specifying composite service (CS) failure handling policies. This formal specification is used to formally validate the consistency of the transactional behavior of the composite service model at design time, according to users' needs. We propose to formally specify with Event-B the transactional service patterns. These patterns formally

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 67–74, 2011.

specified as events and invariants rule to check and ensure the transactional consistency of composite service at design time. Most previous work is based on the model checking technique and does not support the full description of transactional web services. Refinement and proof techniques offered by Event-B method are used to explore it and in section 5 we discuss this approach.

This paper is organized as follows. In Section 2 we introduce a motivating example. Section 3 presents the Event B method, its formal semantics and its proof procedure and introduces our transactional CS model. In Section 4, we present how we specify a pattern-based of the transactional behavior using the Event-B. An overview of the validation methodology is given in Section 5.

2 Motivating Example

In this section, we present a scenario to illustrate our approach we consider a travel agency scenario (Fig.1).the client specifies its requirement in terms of destinations and hotels via the activity “Specification of Client Needs” (SCN). After SCN termination, the application launches simultaneously two tasks “Flight Booking” (FB) and “Hotel Reservation” (HR) according to customer's choice. Once booked, the “Online Payment” (OP) allows customers to make payments. Finally travel documents (air ticket and hotel reservations are sent to the client via one of the services “Sending Document by Fedex” (SDF) ,”Sending Document by DHL” (SDD) or “ Sending Document by TNT” (SDT).

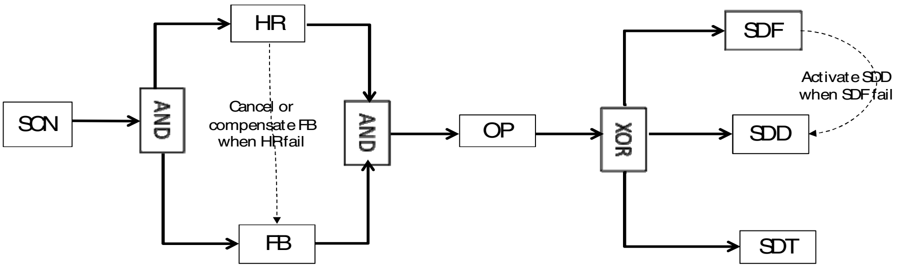

HR
SCN
AND
Cancel or compensate FB when HR fail
AND
OP
XOR
SDF
Activate SDD when SDF fail
SDD
FB
SDT

Fig. 1. Motivating example

To guarantee outstanding reliability of the service the designers specify that services FB, OP and SDT will terminate with success. Whereas on failure of the HR service, we must cancel or compensate the FB service (according to his current state) and in case of failure of the SDF, we have to activate the SDD service as an alternative. The problem that arises at this level is how to check / ensure that the specification of a composite service ensures reliable execution in accordance with the designer’s requirements.

3 Formalizing Transactional Composite Service Using Event-B

To better express the behavior of web services we have enriched the description of web services with transactional properties. Then we developed a model of Web services composition. In our model, a service describes both a coordination aspect and a transactional aspect. On the one hand it can be considered as a workflow services. On the other hand, it can be considered as a structured transaction when the services components are sub-transactions and interactions are transactional dependencies. The originality of our approach is the flexibility that we provide to the designers to specify their requirements in terms of structure of control and correction. We show how we combine a set of transactional service to formally specify the transactional CS model in EVENT-B.

3.1 Event-B

B is a formal method based on he theory of sets, enabling incremental development of software through sequential refinement. Event-B is a variant of B method introduced by Abrial to deal with reactive system. An event-B model contains the complete mathematical development of a discrete system. A model uses two types of entities to describe a system: machines and contexts. A machine represents the dynamic parts of a model. Machine may contain variables, invariants, theorems, variants and events whereas contexts represent the static parts of a model .It may contain carrier sets, constants, axioms and theorems.

The concept of refinement is the main feature of Event-B. it allows incremental design of systems. In any level of abstraction we introduce a detail of the system modeled. Correctness of Event-B machines is ensured by proving proof obligations (POs); they are generated by RODIN to check the consistency of the model. For example: the initialization should establish the invariant, each event should be feasible (FIS), each given event should maintain the invariant of its machine (INV), and the system should ensure deadlock freeness (DLKF). Proof obligations are produced from events in order to state that the invariant condition is preserved.

3.2 Transactional Web Service Model

By Web service we mean a self-contained modular program that can be discovered and invoked across the Internet. Each service can be associated to a life cycle or a statechart. A set of states (*initial, active, cancelled, failed, compensated, completed*) and a set of transitions (*activate(), cancel (), fail(), compensate (), complete()*) are used to describe the service status and the service behavior. A service ts is said to be *retriable*(r) if it is sure to complete after finite number of activations. ts is said to be *compensatable*(cp) if it offers compensation policies to semantically undo its effects. ts is said to be *pivot*(p) if once it successfully completes, its effects remain and cannot be semantically undone. Naturally, a service can combine properties, and the set of all possible combinations is {r; cp; p; (r; cp); (r; p)}[3].

CONTEXT ServiceContext SETS *SWT* *STATES*	AXIOMS Axm1: STATES= {active, initial, aborted, cancelled, failed, completed, compensated}

The initial model includes the context *ServiceContext* and the machine *ServiceMachine*. The context *ServiceContext* describes the concepts *SWT* which represents all transactional web services and *STATES* represents all the states of a given *SWT*. These states are expressed as constants.

The service state which is represented by a functional relation *service_state* defined in VARIABLES clause gives the current state of such a service. The transactional behavior of a transactional web service is modeled by a machine. *Inv1* specifies that *service_state* is a total function, and that each service has a state. In our model, transitions are described by the event. For instance the event *compensate* enables to compensate semantically the work of a service and pass it from *completed* status to *compensate*.

```
MACHINE  ServiceMachine
SEES  ServiceContext
VARIABLES
Service_state
SWT_C
SWT_P
SWT_R
INVARIANTS
Inv1:  service_state∈SWT→STATES
Inv2: SWT_C ⊆ SWT
Inv3: SWT_R ⊆ SWT
Inv4: SWT_P ⊆ SWT
```

```
Activate □ ANY  s  WHERE
grd1   :    s∈SWT
grd2   :    service_state (s) =initial
THEN
act1   :    service_state (s):=active
END
Compensate □ ANY  s  WHERE
grd1   :    s∈SWT_C
grd2   :    service_state (s) =completed
THEN
act1   :    service_state (s):=compensated
END
```

3.3 Transactional Composite Service Model

A composite service is a conglomeration of existing Web services working in tandem to offer a new value-added service [4]. It orchestrates a set of services, as a composite service to achieve a common goal. A transactional composite (Web) service (TCS) is a composite service composed of transactional services. Such a service takes advantage of the transactional properties of component services to specify failure handling and recovery mechanisms. Concretely, a TCS implies several transactional services and describes the order of their invocation, and the conditions under which these services are invoked. To formally specify in Event-B the orchestration we introduced a new context *CompositionContext* which extends the context *ServiceContext* that we have previously introduced. The first refinement includes the context *CompositionContext* and the machine *CompositionMachine* which refine the machine introduced at the initial model. In this section we show how formally the interactions between CS are modeled. We introduce the concept of dependencies. Dependencies are specified using Relations concept. It is simply a set of couples of services. For example, *depA* ($depA \in SWT \leftrightarrow SWT$) represents the set of couples of services that have an activation dependency. These dependencies express how services are coupled and how the behavior of certain services influences the behavior of other services. We distinguish between "normal" execution dependencies and "exceptional" or "transactional" execution dependencies which express the control flow and the transactional flow respectively. The control flow defines a partial services activations order within a composite service instance where all services are executed without failing cancelled or suspended. Activation dependencies express a succession relationship between two services $s_1$and s_2. At this level the refinement of the compensate event is a strengthening of the event guard to take into consideration the condition of compensation of a service when a service will be compensated.

```
Compensate  REFINES Compensate
…
grd4:∃S0·S0∈SWT∧S0↦s∈depCOMP⇒((service_state(S0)=failed)∨
(service_state(S0)=compensated))
THEN act1   :   service_state (s) ≔compensated
END
```

The guard *grd4* in the compensate event in expresses that the compensation of a service s is triggered when a service S0 failed or was compensated and there is a compensation dependency from s to S0. Therefore compensate allows to compensate the work of a service after its termination, the dependency defines the mechanism for backward recovery by compensation, the condition added as a guard specifies when the service will be compensated.

4 Transactional Service Patterns

The use The use of workflow patterns [5] appears to be an interesting idea to compose Web services. However, current workflow patterns do not take into account the transactional properties (except the very simple cancellation patterns category). It is now well established that the transactional management is needed for both composition and coordination of Web services. That is the reason why the original workflow patterns were augmented with transactional dependencies, in order to provide a reliable composition [6]. In this section, we use workflow patterns to describe TCS's control flow model as a composition pattern. Afterwards, we extend them in order to specify TCS's transactional flow, in addition to the control flow they are considering by default. Indeed, the transactional flow is tightly related to the control flow. The recovery mechanisms (defined by the transactional flow) depend on the execution process logic (defined by the control flow). The use of the recovery mechanisms described throw the transactional behavior varies from one pattern to another. Thus, the transactional behavior flow should respect some consistency rules(INVARIANT) given a pattern. These rules describe the appropriate way to apply the recovery mechanisms within the specified patterns. In the following we formally specify these patterns and related transactional consistency rules using Event-B. To extend these patterns we introduce new events that can describe them. For example, to extend the pattern AND-split the machine introduces a new event *AND-split* which defines the pattern AND-split. Due to the lack of space, we put emphasis on the following patterns AND-split and XOR-split to explain and illustrate our approach, but the concepts presented here can be applied to other patterns.

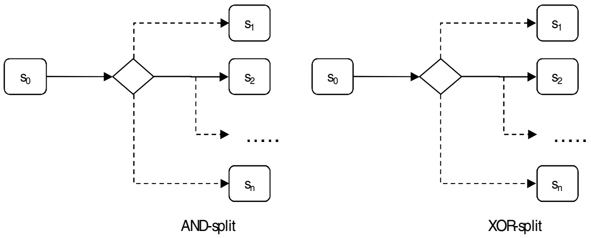


Fig. 2. Studied patterns

An AND-split pattern defines a point in the process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing services to be executed simultaneously or in any order. To verify the transactional consistency of these patterns we add predicates in the INVARIANTS clauses. These invariants ensure transactional consistency according to the context of use. These rules are inspired from [7] which specifies and proves the potential transactional dependencies of workflow patterns. Our example illustrates the application of AND-split pattern to the set of services (SCN, HR, FB) and specifies that exist a dependency of compensation from HR to FB and a cancellation dependency also from HR to FB. The transactional consistency rules of the AND-split pattern support only compensation dependencies from *SWToutside* (Inv23). The compensation dependencies can be applied only over already activated services. The transactional consistency rules supports only cancellation dependencies between only the concurrent services. Any other cancellation or alternative or compensation dependencies between the pattern's services (Inv 11, 12) are forbidden.

- Inv 23: $\forall s.s \in SWToutside \Rightarrow sAS \mapsto s \notin depCOMP$
- Inv 11: $\forall s.s \in SWT\_AS \Rightarrow s \mapsto sAS \notin depANL$
- Inv12: $\forall s, s1.s \in SWT\_AS \wedge s1 \in SWT\_AS \Rightarrow s \mapsto s1 \notin depAL$

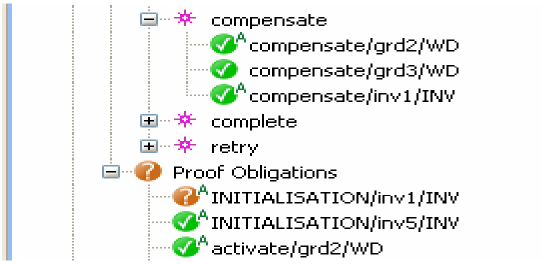
AND-split ≙ ANY S0 SWToutside WHERE grd1 : SWToutside⊆SWT_AS grd2 : S0∈ SWT_AS∖SWToutside grd3 : service_state(S0)=complete THEN act1 : etatSWTout≔activated END	XOR-split ≙ ANY S0 SWToutside sw WHERE grd1 : SWToutside⊆SWT_XS grd2 : S0∈SWT_XS∖SWToutside grd3 : service_state(S0)=complete grd4 : sw∈SWToutside THEN act1 : service_state(sw)≔active END

An XOR-split pattern defines a point in the process where, based on a decision or control data, one of several branches is chosen. To extend the pattern XOR-split, the machine introduces a new event *XOR-split* which defines the pattern XOR-split. Our example illustrates the application of XOR-split pattern to the set of services (OP, SDD, SDF, SDF) and specifies that exist an alternative dependency from HR to FB. The XOR-split pattern supports alternative dependencies between only the services SWToutside, as the alternative dependencies can exist only between parallel and non concurrent flows. The XOR-split pattern support also compensation dependencies from *SWToutside* to *sXS*. Any other cancellation or alternative or compensation dependencies between the pattern's services are forbidden.

- Inv18: $\forall s.s \in SWT\_XS \setminus \{sXS\} \Rightarrow s \mapsto sXS \in depCOMP$
- Inv15: $\forall s.s \in SWT\_XS \Rightarrow s \mapsto sXS \notin depAL$
- Inv22: $\forall s.s \in SWT\_XS \setminus \{sXS\} \Rightarrow sXS \mapsto s \notin depCOMP$

5 Validation

In the previous section, we showed how to formally specify a TCS using Event-B. The objective of this section is to show how we verify and validate our model using proof and ProB animator. We find many proof obligations. Each of them has got a compound name for example, "evt / inv / INV" . A green logo situated on the left of the proof obligation name states that it has been proved (an A means it has been proved automatically). In our case shown in Fig.3 RODIN generates the following proof obligations "compensate / inv1 / INV" . This proof obligation rule ensures that the invariant *inv1* in the *CompositionMachine* is preserved by the *compensate* event.



```
⊟ compensate
    ✓A compensate/grd2/WD
    ✓  compensate/grd3/WD
    ✓A compensate/inv1/INV
⊞ complete
⊞ retry
⊟ ? Proof Obligations
    ?A INITIALISATION/inv1/INV
    ✓A INITIALISATION/inv5/INV
    ✓A activate/grd2/WD
```

Fig. 3. Proof Obligations

Our work is proof oriented and covers the transactional web services. All the Event-B models presented in this paper have been checked within the RODIN platform. The proof based approaches do not suffer from the growing number of explored states. However, the proof obligations produced by the Event-B provers could require an interactive proof instead of automatic proofs. The Event-B formalization of our motivating example defines a cancellation dependency and compensation dependency from HR to FB. For example, by checking the compensation dependency between SCN and HR RODIN mentioned that the proof obligations has not been discharged (Fig. 3). As HR is executed after, it can not exist a compensation dependency from SCN to HR. A red logo with a ”?” appear in the proof tree and it means that is not discharged. To repair this error we can refer to the initialization of the machine and verify the compensation dependencies.

In the development of our model some proof obligations are not discharged but the specifications is correct according to our work in [8] which is specified and validated using Event Calculus. To do so, we use ProB animator [9] to verify our specification of transactional web services. This case study has shown that the animation and model-checking are complementary to the proof, essential to the validation of Event-

B models. In other case, many proved models (proof obligations are discharged) still contain behavioral faults, which are identified with the animators. The main advantage of Event-B develop that can repair errors during the development. It allows the backward to correct specification.

6 Conclusion

The paper addresses the formal specification, verification and validation of the transactional behavior of services compositions within a refinement and proof based approach. The described work uses Event-B method, refinement for establishing proprieties. This paper presents our model of web service enriched by transactional properties to better express the transactional behavior of web services and to ensure reliable compositions. Then we describe how we combine a set of services to establish transactional composite service by specifying the order of execution of composed services and recovery mechanisms in case of failure. Finally we introduced the concept of composition pattern and how we uses it to specify a transactional composite service.

In our future works we are considering the following perspective:

- Using automation approach of MDE type to verify transactional behavior of services compositions.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for event-B. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
3. Mehrotra, S., Rastogi, R., Korth, H.F., Silberschatz, A.: A transaction model for multidatabase systems. In: ICDCS, pp. 56–63 (1992)
4. Medjahed, B., Benatallah, B., Bouguettaya, A., Ngu, A.H.H., Elmagarmid, A.K.: Business-to-business interactions: issues and enabling technologies. The VLDB Journal 12(1), 59–85 (2003)
5. Van Der Aalst, W.M.P., Barros, A.P., Ter Hofstede, A.H.M., Kiepuszewski, B.: Advanced Workflow Patterns. In: Etzionand, O., Scheuermann, P. (eds.) 5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS 2000). LNCS, vol. 1901, pp. 18–29. Springer, Eilat (2000)
6. Bhiri, S., Godart, C., Perrin, O.: Transactional patterns for reliable web services compositions. In: Wolber, D., Calder, N., Brooks, C., Ginige, A. (eds.) ICWE, pp. 137–144. ACM, New York (2006)
7. Bhiri, S., Perrin, O., Godart, C.: Extending workflow patterns with transactional dependencies to define reliable composite web services. In: AICT/ICIW, p. 145. IEEE Computer Society, Los Alamitos (2006)
8. Gaaloul, W., Bhiri, S., Rouached, M.: Event-Based Design and Runtime Verification of Composite Service Transactional Behavior. IEEE Transactions on Services Computing (February 02, 2010)
9. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)

A Constructive Approach to Compositional Architecture Design

Constanze Deiters and Andreas Rausch

Department of Informatics – Software Systems Engineering
Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
{constanze.deiters,andreas.rausch}@tu-clausthal.de
http://sse.in.tu-clausthal.de

Abstract. Most of today's software systems are large-scaled and have to manage manifold demands. To ease their development, reusable and proven architectural building blocks, for example architectural patterns, are often composed to the desired architecture. Building blocks are specified by their structure and behaviour. Additionally, each architectural building block has specific properties which are interpreted as assurances. Keeping assurances also valid during composition of different architectural building blocks is essential for software quality.

This paper introduces an approach which constructs software architectures by composing architectural building blocks and which also assures architectural properties of these compositions. Aiming at a sound approach, a proper description of the different architectural building blocks and their properties is required. Furthermore, this paper presents how to compose architectural building blocks and how to check their assurances.

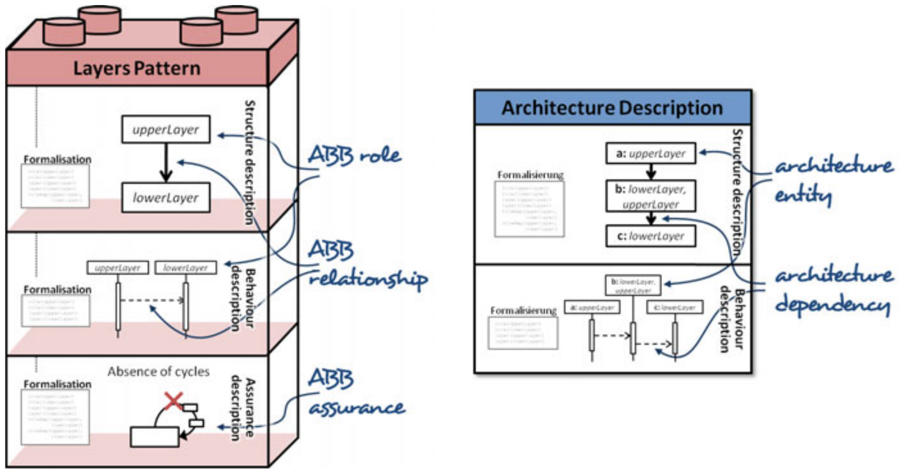
Keywords: Software Architecture, Architecture Composition, Architectural Building Blocks, Architectural Patterns.

1 Introduction

A software architecture defines the basic organization of a system by structuring different architectural elements and relationships between them [1]. With increasing size of a software system also its architecture's size and complexity increase. To handle this complexity, software architectures are composed of ***architectural building blocks*** (ABBs), which summarize related architectural elements and their relationships under abstract entities [2]. Such ABBs are approved templates like architectural patterns [3], architectural principles [4] and reference architectures [1].

Each ABB has properties, for example the absence of cycles in the case of the layers pattern. According to how well their properties suite to the needs of the intended architecture, ABBs are chosen from the set of different ABBs [5]. But selected ABBs could be in conflict, for example, if they have contradictory properties, and thus could lead to defects in the resulting architecture. To avoid

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 75–82, 2011.



(a) (b)

Fig. 1. Concept of an architectural building block (ABB) with description separation into structure, behaviour and assurance (a); concept of an architecture description with separation into structure and behaviour (b)

defective architectures, resulting compositions need to be checked against used ABBs and their properties. Thereby, it can be assured that ABB properties are preserved within the desired architecture.

Hence, the aim of this work is to provide a seamless approach for composing software architectures from ABBs, which assures the preservation of properties owned by used ABBs. One step towards this approach is a description technique to formalize ABBs and software architectures. Additionally, it has to be defined what it means to use an ABB and how to perform the composition of ABBs.

The remaining paper is structured as follows: Section 2 introduces the approach of composing ABBs with assuring their properties. Structure, behaviour and assurances of ABBs are described based on a simple metamodel, which is exemplified in Section 3. The problem described above and its solution touch different fields of related work over which Section 4 gives an overview. Section 5 concludes this paper and gives an outlook on further work.

2 The Overall Approach

In this approach, an ABB is represented by a formalised tripartite description: ***Structure description*** and ***behaviour description*** specify structure and behaviour of an ABB, respectively, and an ***assurance description*** details properties the ABB guarantees (see Fig. 1a). To describe these parts the basic notation elements ***ABB role*** and ***ABB relationship*** are used. For example, considering the ABB *Layers Pattern* the two layers named *upperLayer* and *lowerLayer* are ABB roles and the use relationship from *upperLayer* to *lowerLayer* is an

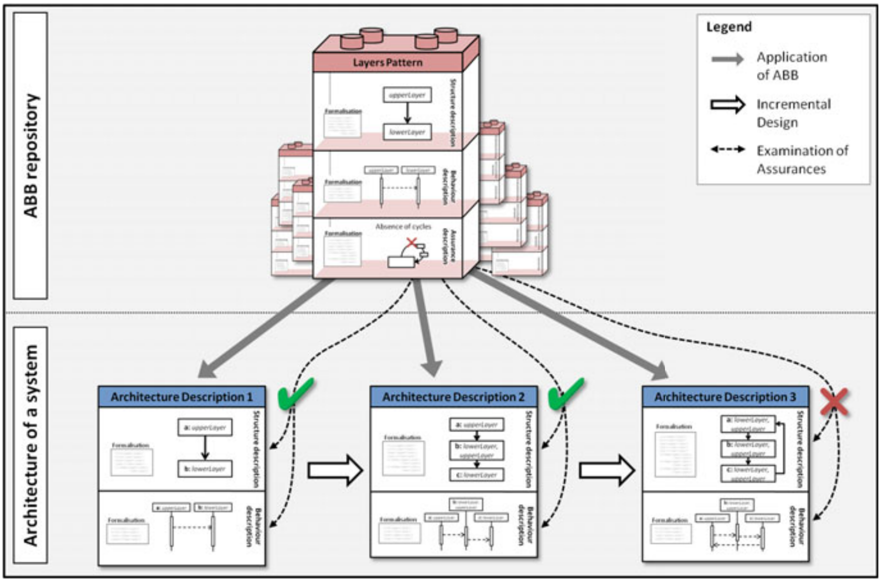


Fig. 2. Instance composition of ABBs and examination of assurances

ABB relationship. A conceivable assurance of this ABB is the property *absence of cycles between layers*.

A concrete architecture is represented by a formalised ***structure description*** as well as a formalised ***behaviour description***. Similar to an ABB description an ***architecture description*** consists of different notation elements: ***architecture entities*** (e.g., the layers a, b and c) and ***architecture dependencies*** (e.g., between the layers a and b or b and c) (see Fig. 1b). Software architectures are now created stepwise by applying ABB by ABB. ABB roles and relationships are assumed by architecture entities and dependencies, respectively. During this process, called ***ABB instantiation***, the structure and behaviour an ABB defines is mapped to the corresponding architecture. An architecture entity can at the same time assume roles from different ABBs and thus belongs to different ABB instances. As a result, the affected ABBs are not only instantiated but also composed (***instance composition***). Figure 1b shows the result of instantiating ABB *Layers Pattern* twice in such a way that the architecture entity b is instance of the ABB roles *lowerLayer* as well as *upperLayer*.

ABBs can also be used to describe architectures at different levels of detail by applying ***hierarchical composition***. This means, that an instance of one ABB or an instance composition of several ABBs specifies the interior of an already existing architecture entity.

Composing ABBs could lead to conflicts in the resulting architecture, for example, because of contradictory properties of applied ABBs. In our approach [6], we consider ABB properties as assurances. To discover conflicts, assurances of all applied ABBs are examined after each composition step (see Fig. 2). These

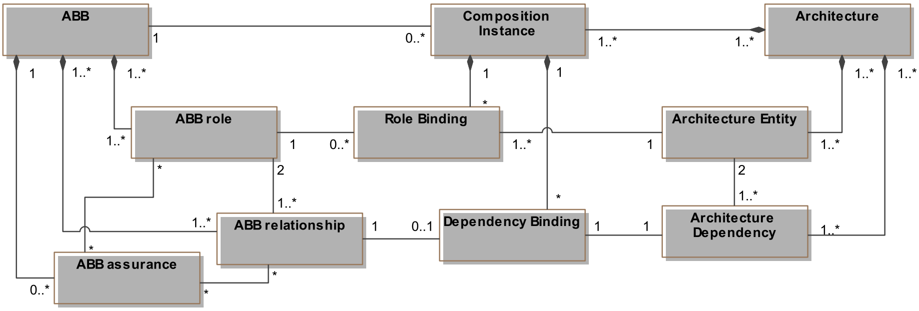


Fig. 3. A description metamodel to formalize ABBs (left part), architectures (right part) and instance relations (middle part) between ABBs and the resulting architecture

examinations exploit the formalised descriptions of ABBs, their instantiations and their assurances. Detected violations of assurances can be presented to the architect, who can now decide how to handle them.

3 Describing ABBs and Architectures

3.1 Description Metamodel

Textual and informal descriptions of ABBs (e.g., in pattern collections [3]) could lead to inexactnesses and ambiguous interpretations of their usage. A concise and uniform formalization of ABBs is thus necessary for a sound approach. Such a formalization needs to be powerful enough to describe the elements of ABBs as well as of architecture descriptions and, according to the approach introduced in this paper, needs to offer notation elements to relate ABB elements to architecture elements.

The metamodel in Figure 3 shows the different notation elements as already introduced in Section 2. An ABB contains at least one ABB role and one ABB relationship. An ABB relationship connects two ABB roles, and each ABB role has to be connected to at least one other ABB role. Additionally, ABBs can be constrained by ABB assurances which are defined over ABB roles and ABB relationships. Architecture descriptions contain at least one Architecture Entity and one Architecture Dependency, which are connected with each other similar to roles and relationships of ABB descriptions. During instantiation, ABB roles and ABB relationships are bound to Architecture Entities and Architecture Dependencies, respectively. These bindings are represented by Role Bindings and Dependency Bindings with every binding being part of exactly one Composition Instance.

To check the defined assurances, ABB as well as architecture descriptions based on this metamodel are formally represented by first-order logic. Elements and associations of the metamodel are mapped to logical predicates with equal or abbreviated names. Unary predicates represent instances of the metamodel elements (Fig. 5, left column), like `ABB/1` stating that an element used as parameter

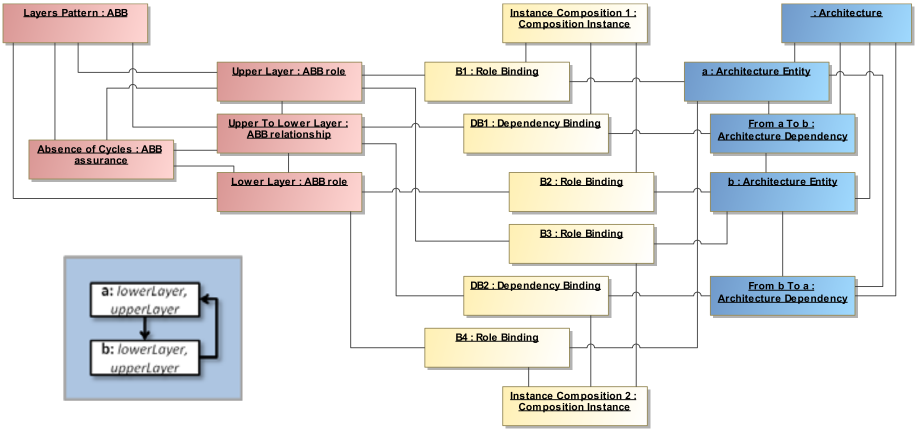

Layers Pattern : ABB
Instance Composition 1 : Composition Instance
: Architecture
Upper Layer : ABB role
B1 : Role Binding
a : Architecture Entity
Upper To Lower Layer : ABB relationship
DB1 : Dependency Binding
From a To b : Architecture Dependency
Absence of Cycles : ABB assurance
Lower Layer : ABB role
B2 : Role Binding
b : Architecture Entity
B3 : Role Binding
a: lowerLayer, upperLayer
b: lowerLayer, upperLayer
DB2 : Dependency Binding
From b To a : Architecture Dependency
B4 : Role Binding
Instance Composition 2 : Composition Instance

Fig. 4. Applying the ABB *Layers Pattern* twice so that it results in a cycle between the two layers *a* and *b* (architecture structure in the lower left corner)

is an ABB. Associations between elements are represented by n-ary predicates (Fig. 5, middle column), like `ABBrelFromTo/3` defined with appropriate parameters expresses that an ABB relation connects two ABB roles.

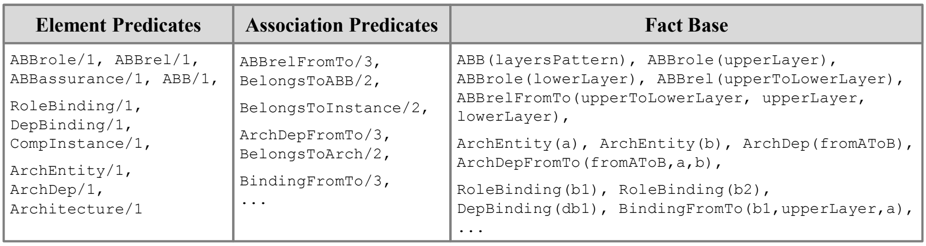
3.2 Applying the Description Metamodel - Example

Exemplary, the ABB *Layers Pattern* and its instance composition are considered. The object diagram in Figure 4 illustrates the ABB and architecture descriptions and the bindings created during two instance compositions. A logical fact base represents the elements of this diagram using the defined predicates (Fig. 5, right column); for the sake of brevity, some statements are omitted in comparison to the object diagram.

In the upper left part of the diagram ABB *Layers Pattern* is represented. Its two roles UpperLayer and LowerLayer are typed as ABBrole and its relationship UpperToLowerLayer connecting both roles is typed as ABBrelationship. Additionally, this ABB owns an assurance AbsenceOfCycles typed as ABBassurance. The ABB description is similarly mapped to logical facts. For example, `ABB(layersPattern)` and `ABBrole(upperLayer)` defines that `layersPattern` is an ABB and `upperLayer` is an ABB role, respectively. The logical fact `ABBrelFromTo(upperToLowerLayer,upperLayer,lowerLayer)` expresses that `upperToLowerLayer` connects `upperLayer` and `lowerLayer`.

In the same way, the architecture description is represented in the right part of the object diagram and as logical facts in Figure 5. For example, a is typed as ArchitectureEntity in the diagram and expressed as logical fact `ArchEntity(a)`.

Instance compositions create bindings from ABB roles and relationships to architecture entities and dependencies, respectively. These bindings and their corresponding composition instances are depicted in the middle part of the object diagram. For example, the binding element B1 typed as RoleBinding binds ABB



Element Predicates	Association Predicates	Fact Base
ABBrole/1, ABBrel/1, ABBassurance/1, ABB/1, RoleBinding/1, DepBinding/1, CompInstance/1, ArchEntity/1, ArchDep/1, Architecture/1	ABBrelFromTo/3, BelongsToABB/2, BelongsToInstance/2, ArchDepFromTo/3, BelongsToArch/2, BindingFromTo/3, ...	ABB(layersPattern), ABBrole(upperLayer), ABBrole(lowerLayer), ABBrel(upperToLowerLayer), ABBrelFromTo(upperToLowerLayer, upperLayer, lowerLayer), ArchEntity(a), ArchEntity(b), ArchDep(fromAToB), ArchDepFromTo(fromAToB,a,b), RoleBinding(b1), RoleBinding(b2), DepBinding(db1), BindingFromTo(b1,upperLayer,a), ...

Fig. 5. Expressing the different descriptions with first-order logic

role UpperLayer to architecture entity a. Formalised as logical fact, this binding is expressed as BindingFromTo(b1,upperLayer,a), whereas RoleBinding(b1) states that b1 is a role binding.

ABB *Layers Pattern* owns the assurance *Absence of Cycles*. This assurance is expressed by the following logical rule using the defined predicates:

$$\nexists\ \texttt{X : ArchDep(X)} \wedge \texttt{BindingFromTo(DB,upperToLowerLayer,X)}$$
$$\wedge\ \texttt{DepBinding(DB)} \wedge \texttt{ArchDepFromTo(X,A,B)} \wedge \texttt{ArchDepFromTo+(\_,B,A)}$$

The rule expresses that no element X with the following properties exists: X is an architecture dependency and bounded via a dependency binding represented by DB to the ABB relationship upperToLowerLayer, and X connects two architecture entities represented by A and B (see ArchDepFromTo(X,A,B)). A cycle occurs, if there exists also a transitive closure of architecture dependencies from B to A (see ArchDepFromTo+(_,B,A)).

Instance compositions applied as depicted in Figure 4 and Figure 5 result in a cycle between architecture entities. The above defined logical rule can be used to detect this assurance violation. Executing the negated rule as query on the logical fact base provides architecture dependencies causing the violation.

4 Related Work

An architecture is characterised by its structural elements and also by the behaviour of its parts. Therefore, an adequate description language needs to handle both. Common architecture description languages (ADLs) usually define at least the three primary concepts component, connector and configuration to describe structure as well as behaviour [7]. But ADLs vary in their intended purpose, for example, focusing on distinct system categories or following certain programming languages and their paradigms [8,7,9].

Important for the approach introduced in this paper is the ability to describe an ABB as a kind of template and to instantiate these templates. But, most ADLs do not explicit support template mechanisms or even support only distinct architectural patterns [10,11]. Only few ADLs, like WRIGHT [12] or Alloy [13], offer an explicit template mechanisms but with the limitation that an architecture can base on only one template. This requires combining ABBs first on an abstract level, for example, by defining new merged ABB elements. Then, the new ABBs can be instantiated. The same procedure is followed by [14]. Other approaches treat applied ABBs as separate parts of an architecture whose elements are coupled by connectors beyond the parts' borders [15,16,14]. Considering only this kind of composition does not allow overlapping ABB elements, which does not seem to represent reality in software architectures. Comparably restrictive as the aforementioned strategy it is to compose ABBs by handling them as own subsystems [15,16]. This leads to unnecessary architectural decomposition levels, where an overlapping of ABB elements is impossible and architecture entities are not connected directly. Description approaches considering ABB elements as roles [17,18] prevent the mentioned shortcomings. They allow architectural entities to belong to instances of different ABBs by assuming roles and allow instantiated ABBs to be intertwined.

Furthermore, ABBs also specify assurances which constrain their usage. An appropriate formalism has to include structure and behaviour elements and has to provide sufficient expressive power. ADL constraints are usually focused on behavioural aspects like a kind of communication protocol [7,11]. Other approaches use constraints to limit the entity types that can bound to a role [14].

5 Conclusion and Further Work

In this paper, we introduced an approach to construct software architectures from architectural building blocks (ABBs) assuring architectural properties of composed ABBs. For this purpose, we divided the descriptions of ABBs and architectures into the three parts structure, behaviour and assurances. Instance bindings relate ABB elements to architecture elements.

The presented metamodel, used to express ABBs and architecture descriptions, also provides facilities to represent the bindings of ABB compositions. Properties, ABBs own in addition to structure and behaviour, are expressed as assurances using first-order logic, which provides high expressive power and flexibility. Formalising ABB and architecture descriptions also using first-order logic enables to check the ABB assurances against them.

In our approach, ABB entities are represented as roles. This allows to relate roles of different ABBs to one and the same architecture entity, which hence assumes all duties the several roles specify.

We exemplified the approach as well as the metamodel using only one specific ABB, namely the *Layers Pattern*. It is undenied that there exist more ABBs that must be considered to prove the practicability of this approach. They need to be summarised, analysed for their elements and properties and formalised. Then, it is possible to build and study larger examples of composed ABBs.

Furthermore, an extension of the metamodel is planned to detail descriptions and to support hierarchical composition of ABBs. For example, adjacent layers could contain other architecture entities, which are interconnected over the layer boundaries and reverse to the relationship the layers pattern implies. Since this structure results in a cycle the ABB assurance *Absence of cycles* is violated.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Longman, Amsterdam (2003)
2. The Open Group, (ed.): The Open Group Architecture Framework (TOGAF), 8th edn. Van Haren Publishing (2007)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns, 1st edn. Software Design Patterns, vol. 1. Wiley & Sons, Chichester (1996)
4. Bergner, K., Rausch, A., Sihling, M., Vilbig, A.: Putting the Parts Together - Concepts, Description Techniques,and Development Process for Componentware. In: Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS 2000). IEEE Computer Society, Washington DC, USA (2000)
5. Herold, S., Metzger, A., Rausch, A., Stallbaum, H.: Towards Bridging the Gap between Goal-Oriented Requirements Engineering and Compositional Architecture Development. In: Proceedings of the 2nd SHARK-ADI Workshop at ICSE 2007, IEEE Computer Society, Washington DC, USA (2007)
6. Deiters, C., Rausch, A.: Assuring architectural properties during compositional architecture design. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 141–148. Springer, Heidelberg (2011)
7. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
8. Kamal, A.W., Avgeriou, P.: An Evaluation of ADLs on Modeling Patterns for Software Architecture. In: Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007). LNCS. Springer, Heidelberg (2007)
9. Vestal, S.: A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center (1993)
10. Clements, P.C.: A Survey of Architecture Description Languages. In: Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD 1996). IEEE Computer Society, Los Alamitos (1996)
11. Di Nitto, E., Rosenblum, D.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In: Proceedings of the 21st ICSE, pp. 13–22. ACM, Los Angeles (1999)
12. Allen, R.J.: A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University (May 1997)
13. Kim, J.S., Garlan, D.: Analyzing Architectural Styles. Journal of Systems and Software 83, 1216–1235 (2010)
14. Hammouda, I., Koskimies, K.: An approach for structural pattern composition. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 252–265. Springer, Heidelberg (2007)
15. Abd-Allah, A.A.: Composing Heterogeneous Software Architectures. Ph.D. Thesis, University of Southern California (1996)
16. Gacek, C.: Detecting Architectural Mismatches During Systems Composition. Ph.D. thesis, University of Southern California (1998)
17. Riehle, D.: Describing and Composing Patterns Using Role Diagrams. In: Proceedings of the 1996 Ubilab Conference, pp. 137–152. Universitätsverlag Konstanz (1996)
18. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. In: Proceedings of the 20th Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), vol. 40, pp. 133–146. ACM, New York (2005)

Capturing Architecture Evolution with Maps of Architectural Decisions 2.0

Andrzej Zalewski, Szymon Kijas, and Dorota Sokołowska

Warsaw University of Technology,
Institute of Automatic Control and Computational Engineering
a.zalewski@ia.pw.edu.pl, s.kijas@elka.pw.edu.pl

Abstract. Modern IT systems evolve being re-architected throughout their entire lifetime. Existing architecture decision-making approaches are oriented towards systems design, rather than systems evolution. However, real-life architecture evolution is substantially different to initial architectural design. It is a disorderly process, in most cases unrepeatable, and therefore difficult to be put into a predefined rut as most approaches try to do. MAD 2.0 model has been developed to support architect-practitioners working on systems evolution. It does not impose any predefined classification or hierarchy of architectural decisions and assumes a limited number of kinds of relations between architectural decisions. This makes a model of the decision process intuitive and easy to comprehend. To explain the choices made and capture their rationale, the entire decision situation is presented, including: the decision topic, considered design options, relevant requirements, and the advantages and disadvantages of every considered option. The proposed models and approach, supported by an appropriate modelling tool, has been validated in the real life conditions of one of the telecom companies.

Keywords: architectural decisions, architectural knowledge, diagrammatic representation.

1 Introduction

Recent developments in systems architectures, especially service-oriented architectures, concepts of business process management or enterprise service bus promote transition from design-oriented to evolution-oriented engineering. Evolution is an intensive process nowadays, dominating a system's lifetime.

We argue in section 2 that re-architecting substantially differs from initial design, and so it requires an approach to architecture decision-making tailored to its specificity. The purpose of this paper is to present an approach to support and capture architecture decision making during system evolution, which has been validated in practice.

The notation MAD 2.0, used here as a model of architecture decisions and decision making process, is an extension of our former work [9]. MAD 2.0 was inspired by the outcomes of a workshop held with architects-practitioners – see section 2.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 83–96, 2011.

The rest of the paper is organised as follows: the MAD 2.0 model and its components are presented in section 3, tool supporting the notation is characterised in section 4, the practical application of MAD 2.0 is shown in section 5, the results are widely discussed in section 6 against the background of existing architecture decision-making models and approaches, with a paper summary and further research prospects comprising section 7.

2 Inspiration

The development of the MAD 2.0 notation and modelling approach was directly inspired by observations made during a workshop held with 22 IT architects working for one of the largest telecom firms in Poland. Architects focus mainly on the evolution of this entire complex system-of-systems, consisting of more than 100 systems integrated with a BPM solution based on SOA. The workshop was aimed at developing architects' skills by teaching and practicing concepts from the area of architecture decision-making, e.g. architecture decision modelling, with text records [3], decisions classifications [4], [5], identifying and classifying relations between the architectural decisions as well as architecture decision-making approach presented in [4].

The survey performed at the end of the workshop showed that:

- almost 80% of architects found the presented concepts and models as overcomplicated,
- about 65% found them not adequate to their everyday jobs,
- about 90% stated they always work under time pressure and have very little time to document architectural knowledge,
- almost 75% assessed the repeatability of their work as low, and another 15% saw their work as unrepeatable,
- almost 80% lack important architectural knowledge on existing systems.

These results confirm the concerns as to whether architecture decision-making really helps to manage complexity, as presented in [10]. However, there are also deeper, fundamental reasons for such perception. Telecoms belong to the class of "*emergent organisations*", whose systems are "*subject to constant urgent change*" (comp. [12]). Their evolution is rather a random than a highly predictable, carefully deliberated process following an earlier established path as in [13].

We analysed 25 cases of changes made to the systems. All of them were driven by an unexpected and unforeseeable change or emergence of business needs (e.g. changes to tariffs, development of new tariffs or new product support, esp. support for product bundles). In such conditions architects make architectural decisions rather disorderly, trying to achieve solutions based on reuse and adaptation of existing systems as well as purchase of new ones. Such decisions do not match any particular predefined abstraction levels or classifications like [4], [5]. They usually cannot be reapplied as a predefined solution to other problems. Existing architecture decision making models and methods are more oriented on well-structured architecture development than on its rapid, chaotic changes.

The survey and subsequent analysis of change cases indicated that to extend the architect's everyday practice with architectural knowledge capturing and documenting [1]:

- architectural knowledge has to be captured as it emerges – together with the resolution of architectural problems;
- the overhang connected with capturing architectural knowledge has to be minimised;
- decision-making concepts have to be easy to comprehend.

This led us back to the idea of mind-mapping architectural decisions sketched in our earlier paper [9]. The Maps of Architectural Decisions notation was redesigned with consultation with architect-practitioners and tool supporting architecture decision-making was developed to validate the whole concept in practice.

3 MAD 2.0 Notation and Modelling Approach

MAD 2.0 was crafted to assist architects in architecture decision-making, without enforcing any particular architecting approach or decision-making order. MAD 2.0 works similarly to popular mind maps used to graphically present a problem structure. The model consists of two diagrams Architecture Decision Relationship Diagrams (ADRD) – section 3.1, and Architecture Decision Problem Maps (ADPM) – section 3.2). ADRD represents the logic of the decision-making process – the diagram can be developed gradually, while ADPM models the internal structure of a single decision problem. The notation's syntax and validity rules have been presented in section 3.3.

3.1 Architecture Decisions Relationship Diagram (ADRD)

The Architecture Decisions Relationship Diagram is built out of just two basic elements (fig. 1):

- **Decision problem** – represents the architectural issue being considered;

Attributes: problem name, problem description, status, creation date, resolution date, extended solution rationale.

States: *defined* – indicates a newly defined project, *being solved* – an ADPM for the problem has been created, but the problem has not been resolved yet, *resolved*, *requires reassessment* – indicates that solution, or the occurrence of other problem, requires reconsidering an already resolved problem.

- **Connector** – in its basic form shows just that one problem led architect to the one indicated by an arrow, in the form with a hexagon it indicates that the solution of a given problem constrains the possible solutions of the pointed problem ("constrains relation"). Two Decision problems can be connected only once.

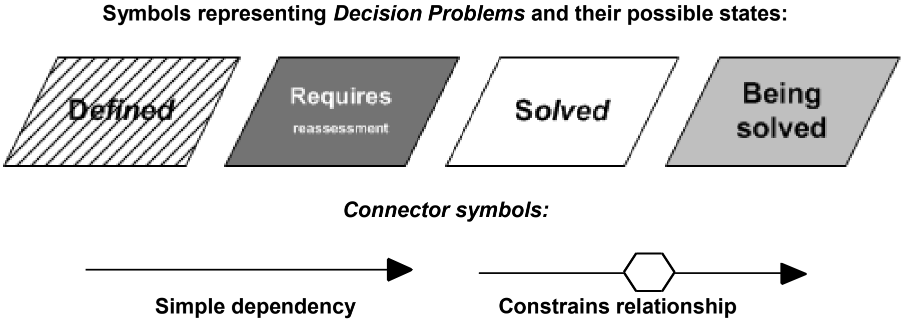


Fig. 1. The Elements of an Architectural Decisions Relationship Diagram

Problems can group on the ADRD by surrounding some of them with a solid line – compare fig. 5. Such a group is treated like a single problem. This way architects can indicate that problems concern a closely-related issue (e.g. define domain solution).

3.2 Architecture Decision Problem Map (ADPM)

The Architecture Decision Problem Map is where architectural decisions are actually captured. The diagram was constructed to show the structure of a given architectural problem (issue) consisting of: Decision-maker, set of relevant Requirements, considered Solutions together with their Pros and Cons. If a given solution meets certain requirements, it is considered as a Pro, otherwise it is a Con. The latter is a key change made to the diagram in comparison with its version presented in [9].

The elements of the ADPM diagram have been summarised in fig. 2. The central element of a diagram is a single decision problem symbol (as in fig. 1) representing the architectural issue (problem) being analysed. The other symbols are:

- **Solution** – represents a single solution to the architectural problem considered

  Attributes: name, state, description, generated problems.

  States: *defined* – assigned immediately after creating an element; *feasible* – indicates a solution meeting all the requirements, *infeasible* – indicates a solution that does not meet one of the requirements (the two former states are assigned automatically), *chosen* – indicates the finally selected solution (assigned by the decision-maker).

- **Requirements** – represents a requirement relevant to a given architectural problem

  Attributes: *name*, *description*

- **Decision-maker** – represents a person or a group of people responsible for the resolution of a related architectural problem.

  Attributes: name, remarks

- **Pro or Con** – represents a single advantage or disadvantage of a given solution

  Attributes: name, description, state, related requirement (met or unmet requirement if the given element represents such a case)

State: *defined* – assigned immediately after creating a given Pro or Con element; *minor*, *medium*, *major* – declares the importance of a given advantage or disadvantage of a given solution.

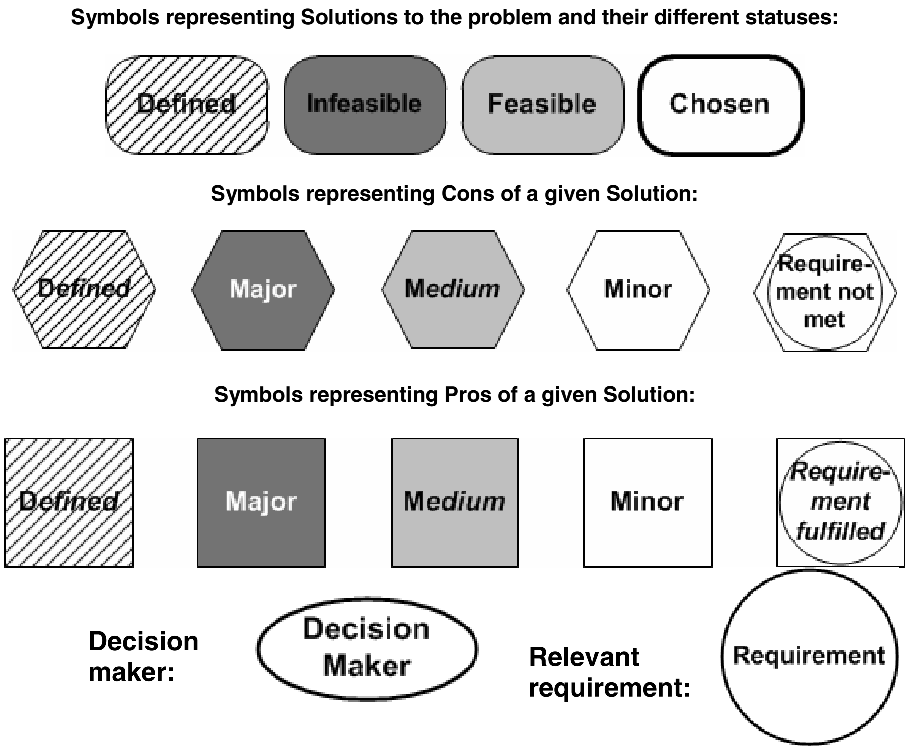


Fig. 2. The Elements of ADPM Diagram

3.3 Model Syntax and Validity Rules

The syntax of the MAD 2.0 model is quite intuitive; here is a summary of the rules:

- Rule 1. Two decisions represented on ADRD may be unconnected or connected with just a single connector.
- Rule 2. *Decision-maker*, *Requirement* and *Solution* symbol on ADPM can only be attached to a *Decision problem* symbol.
- Rule 3. *Pros* and *Cons* symbols on ADPM can only be attached to a solution symbol.

Although, MAD 2.0 models are semiformal by nature, the syntax imposes a certain structure of the information representing architectural decisions enhancing the potential of model analysis.

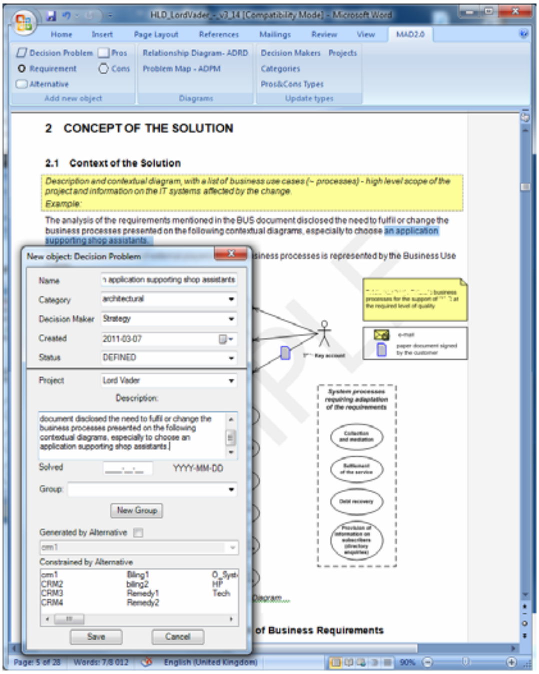
- Rule 1. Every decision problem has to be resolved, i.e. one of the solutions chosen.
- Rule 2. Every solution has to be assessed in the context of every requirement relevant to the given problem. All these requirements have to be finally classified as either Pros (requirement met) or Cons (requirement not met) of a given solution.

- Rule 3. Every Pro (Advantage) or Con (Disadvantage) may be attached to a given solution only once.
- Rule 4. Only a single solution to a given problem can be in the "Chosen" state.
- Rule 5. Pros and Cons connected with a given solution cannot be mutually contradictory.
- Rule 6. Two solutions to a problem cannot designate virtually the same resolution to the same architectural problem.

Rules 1-4 can be verified automatically, while rules 5-6 can be verified with a model walkthrough.

4 A Modelling Tool for MAD 2.0

We developed a software tool supporting MAD 2.0 to validate its concepts in practice. The tool is designed as a diagram editor being an extension to MS Word. Architects and analysts often create documentation with MS Word (despite the availability of more advanced specification and modelling tools). Therefore, to capture the knowledge as it emerges, it seems to be a good idea to bring the architecture decision modelling tool as close as possible to the general documentation tool. This makes it possible to connect decisions to the relevant fragments of system documentation.



2 CONCEPT OF THE SOLUTION

2.1 Context of the Solution

Description and contextual diagram, with a list of business use cases (~ processes) - high level scope of the project and information on the IT systems affected by the change.

Example:

The analysis of the requirements mentioned in the BUS document disclosed the need to fulfil or change the business processes presented on the following contextual diagrams, especially to choose an application supporting shop assistants.

...siness processes is represented by the Business Use

Name: n application supporting shop assistants

Category: architectural

Decision Maker: Strategy

Created: 2011-03-07

Status: DEFINED

Project: Lord Vader

Description:

document disclosed the need to fulfil or change the business processes presented on the following contextual diagrams, especially to choose an application supporting shop assistants.

Solved: ____-__-__ YYYY-MM-DD

Group:

Generated by Alternative □

crm1

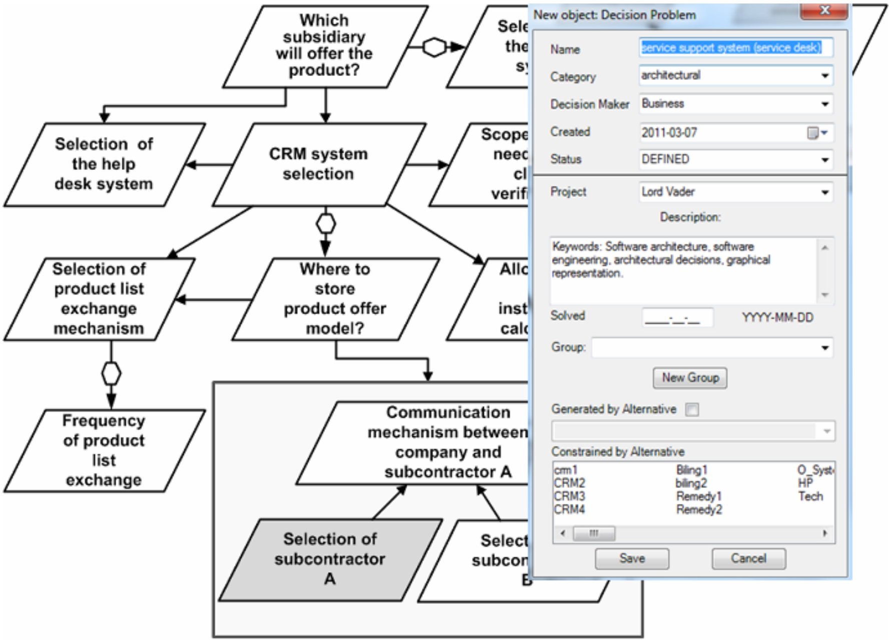
Constrained by Alternative

crm1	Biling1	O_Syst
CRM2	biling2	HP
CRM3	Remedy1	Tech
CRM4	Remedy2	

...of Business Requirements

Fig. 3. Example – creating a new decision problem

The tool provides a diagram editor for ADRD and ADPM, which can be invoked from the MS Word menu. It imposes model syntax and provides for model verification as described in section 3.3. Some screenshots are shown in fig. 3 and 4.


Which subsidiary will offer the product?
Sele the s
Selection of the help desk system
CRM system selection
Scope neec cl verif
Selection of product list exchange mechanism
Where to store product offer model?
Allc inst calc
Frequency of product list exchange
Communication mechanism between company and subcontractor A
Selection of subcontractor A
Select subcon B
New Group
Save
Cancel

New object: Decision Problem

Name: service support system (service desk)
Category: architectural
Decision Maker: Business
Created: 2011-03-07
Status: DEFINED
Project: Lord Vader
Description:
Keywords: Software architecture, software engineering, architectural decisions, graphical representation.
Solved: ____-__-__ YYYY-MM-DD
Group:
Generated by Alternative □
Constrained by Alternative

crm1	Biling1	O_Syst
CRM2	biling2	HP
CRM3	Remedy1	Tech
CRM4	Remedy2	

Fig. 4. Editing ADRD diagram

5 A Case Study

MAD 2.0 was validated on a real world system evolution case study. This was done with the same telecom company we had a workshop with a couple of months earlier. This company offers various telecommunication services (called products). The dealers use a set of various specialised applications to serve their clients. These are mainly CRM applications supporting the selling process at all its stages: client registration or identification, product offer, product support and modification.

The company wants to offer a new product composed of the installation and maintenance of fibre optic networks services and the delivery of IT equipment being sold together as a product bundle. The product can be sold by two subsidiaries of the telecom company under their own brand. There are various financing options planned for such a product: wire transfer, credit instalment payments or even leasing.

The product components will be provided by two independent external subcontractors: "*subcontractor A*" will deliver IT equipment, "*subcontractor B*" will install fibre-optic networks in the clients' premises and will configure the delivered equipment so that both elements operate together.

The overall structure of the architecting task has been depicted in fig. 5 with ADRD, which comprises the following architectural decisions:

1. *CRM system selection* – product sales are supposed to be supported with a CRM system. However, as a result of a number of mergers/acquisitions there are a number of CRM systems used originally by the merged companies. None of them contains a complete client database; they support different product business models.
2. *Selection of the billing system* – There are two such systems, though each of them contains only a part of the entire client dataset. Moreover, these parts overlap substantially.
3. *Selection of the help desk system* – as such services have not been offered to clients yet, the help desk system used so far for internal purposes is one option, and the development of a newer system is another one.
4. *Selection of subcontractor B* – This is both a technical and business decision.
5. *Selection of subcontractor A* – This is both a technical and business decision
6. *Which subsidiary will offer the product?* – there are two subsidiaries of the telecom company that can offer the product under their own brands; however, they differ a lot in terms of their own IT systems and IT infrastructure, while it is necessary to interface appropriate systems of telecom company with those used by subsidiary or at least to enable subsidiary to access some systems of mother company.
7. *Should lease and loan options be processed separately?* – the accounting department uses different systems for these two options, while from the vendor's point of view it is sensible to use a single system for both.
8. *Where to store product offer model* – The product list of the company should be accessible to other systems, so they have to support the same product business model. It is supposed to be synchronised with the product list of subcontractor A.
9. *Scope of data needed for client verification* – describes the set of data needed to assess a client's financial credibility;
10. *Selection of credit rating system* – there are a number of systems (internal and external) that can be used to assess the client's financial credibility;
11. *Allocation of instalment calculator* – the question is where to allocate the calculator's functionality – should it be extending one of the applications used by product sellers, or should it be a separate system;
12. *Communication mechanism between company and subcontractor A?* – the communications necessary to keep the vendor's data (product list, warehouse, client order processing support etc.) up to date;
13. *Selection of product list exchange mechanism* – the product lists of subcontractor A and the telecom company have to be periodically synchronised;
14. *Frequency of product list exchange* – to achieve this, various database synchronisation mechanisms may be needed.

Fig. 5 shows that the resolution of problem No 6 leads to problems Nos 1, 2, 3; the resolution of issue No 8 leads to a group of problems concerning organising systems interaction necessary to support product delivery.

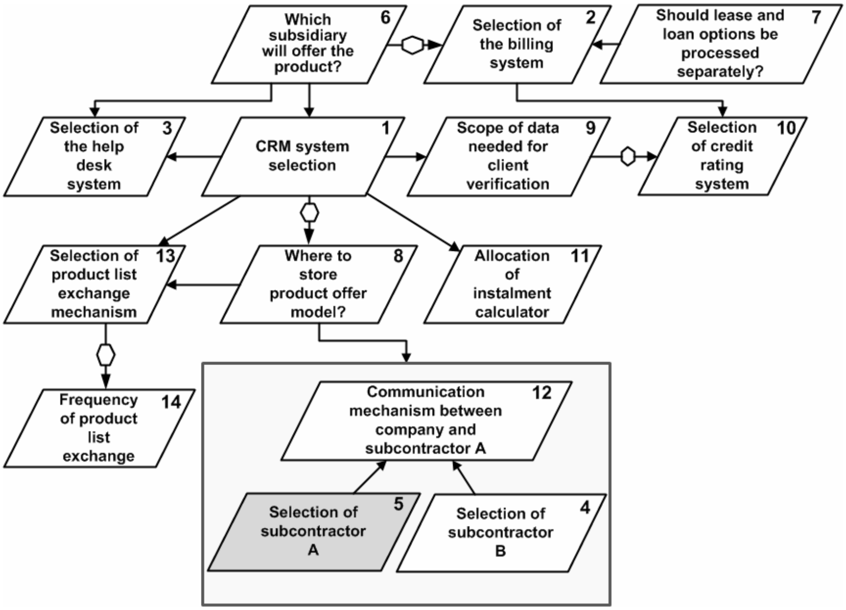

Which subsidiary will offer the product? 6
Selection of the billing system 2
Should lease and loan options be processed separately? 7
Selection of the help desk system 3
CRM system selection 1
Scope of data needed for client verification 9
Selection of credit rating system 10
Selection of product list exchange mechanism 13
Where to store product offer model? 8
Allocation of instalment calculator 11
Frequency of product list exchange 14
Communication mechanism between company and subcontractor A 12
Selection of subcontractor A 5
Selection of subcontractor B 4

Fig. 5. ADPM developed for the case study

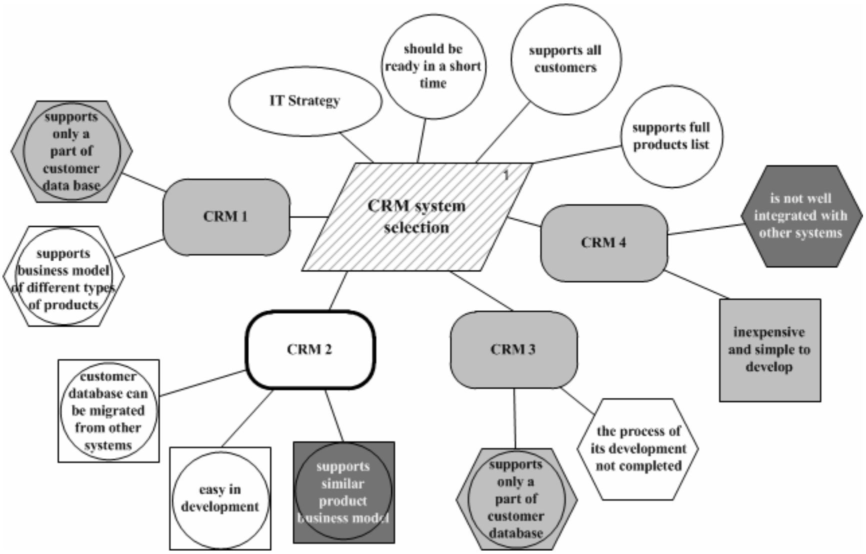

IT Strategy
should be ready in a short time
supports all customers
supports full products list
supports only a part of customer data base
CRM 1
supports business model of different types of products
1
CRM system selection
CRM 4
is not well integrated with other systems
inexpensive and simple to develop
CRM 2
CRM 3
customer database can be migrated from other systems
easy in development
supports similar product business model
supports only a part of customer database
the process of its development not completed

Fig. 6. ADPM for the problem of CRM selection

ADPMs were developed for all the architectural problems shown in fig. 5. We show only two out of fourteen diagrams modelling architectural problems and decision-making. Fig. 6 shows the structure of problem No 1 concerning the selection of an appropriate CRM system to support the new offer. The CRM system should be developable or customisable in a short time, it should support: all the customers (complete client database), frequent offer changes, provide client data and support full product list including the new product. Four existing or developed CRM systems are candidate solutions; the architect has to choose only one of them. In this case the solution is very easy to identify – only CRM 2 meets all the requirements. Obviously in many cases the situation will not be as clear cut as this time, and may require an individual judgement.

Problem no. 3 concerning the selection of the helpdesk system has been presented in fig. 7. ADPM shows that there are three possible choices here. In this case as well, only one of the candidates meets the prescribed requirement of supporting the hardware help desk.

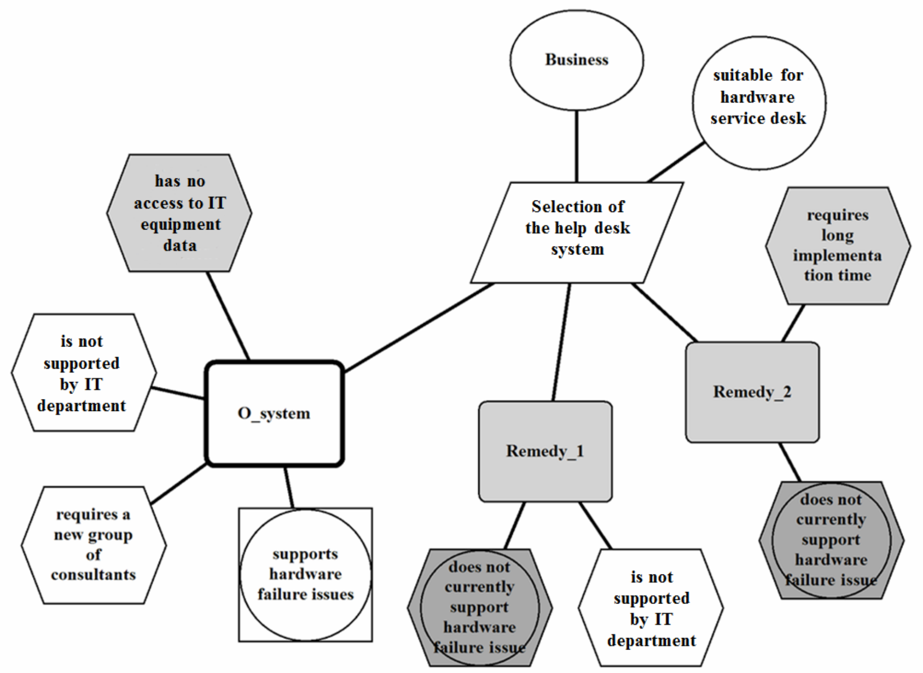

Business
suitable for hardware service desk
has no access to IT equipment data
Selection of the help desk system
requires long implementa tion time
is not supported by IT department
O_system
Remedy_2
Remedy_1
requires a new group of consultants
supports hardware failure issues
does not currently support hardware failure issue
is not supported by IT department
does not currently support hardware failure issue

Fig. 7. ADPM for the problem of service desk support system selection

6 Discussion: Related Work

Developing models supporting architectural decision-making, we face the classical dilemma between model usability (i.e. simplicity, legibility, comprehensibility, etc.) and its complexity defining its expressiveness (i.e. information capacity, level of formalisation) and analysability. The more complicated the model, the less usable it is, and vice versa: we can increase usability by decreasing model's complexity, and by the same its expressiveness and the range of analyses it enables. Over-complicated architecture decision-making models create complexity of their own, adding to the complexity of the overall systems construction instead of supporting complexity control [10].

In such a discussion we find semiformal diagrams a rational choice. Semiformal diagrammatic modelling has turned out to provide the right balance between model usability and complexity. For this reason, semiformal diagrams have been dominating software engineering for at least the past thirty years, in the form of structured and object-oriented methods.

MAD 2.0 models have been crafted to assist system architects similarly to create mind maps. This helps to capture architectural knowledge as it gradually comes to light while elaborating the architecture (i.e. decision-making). The model is certainly not overloaded with information, though it is still possible to verify some consistency/completeness rules (section 3.3). In terms of the level of formalism, MAD 2.0 stands between text models e.g. [3], [11], and partially formalised models [4]. An extended comparison of these models contains table 1. Although, the range of information concerning architectural decisions and decision making process has been limited, the most important components of architectural knowledge are still preserved, i.e. decision's rationale, considered solutions and their pros and cons.

ADPM is similar to the rationale model of [8]. An assessment of every possible problem solution by indicating the pros and cons thereof seems intuitive and comes from the early works of Bosch [2]. It can also be found in the most modern developments like [7].

The classification of ADs and top-down decision-making as proposed in [4] is perfectly suited to developing a system from scratch. It more closely resembles the configuration of a predefined recurring solution rather than the resolution of a unique problem that requires creative thinking.

It is difficult to apply categories defined in [4] or [5] to unstructured problems resulting from the evolution of systems of systems. If one considers the architectural problems listed in section 5, it will turn out that they are generally difficult to classify, with most of them belonging to the executive problem category. Architecting systems of systems mainly involves reusing existing systems, where the internal structure usually remains unchanged. Therefore, lower level architectural decisions (e.g. technology, vendor) play a less important role.

MAD 2.0 assumes no predefined classification of architectural decisions, which is a reasonable decision in the above context. The other drawbacks of decision classification have been investigated in [10].

Only two kinds of relations between ADs are available in MAD 2.0. This provides for a smooth, uninterrupted flow of the architecting process, as architects do not need to worry about which of at least several kinds of relations to chose, which often becomes a separate challenge. This makes advanced analyses similar to that presented in [4] impossible, which is the price for higher model usability.

The MAD 2.0 tool concept is similar to Knowledge Architect [6] – architectural decision models are linked to appropriate parts of requirement specifications.

Although MAD 2.0 has been motivated by the rapid, random changes typical for the evolution of systems supporting emergent organisations it can also be used as a kind of light-weight architecture decision making model for the initial architecture development.

Table 1. Models supporting architectural decision making – a comparison

	Text models ([3], [11])	**MAD 2.0**	**Graph/text model proposed in [4]**
Model form	Text records	Diagrams, additional information stored in attributes of diagram elements	Graphs, certain elements accompanied with text attributes.
Level of formalism	Basic information structuring (fields of text records).	Syntax defined, simple consistency / completeness checking.	Syntax defined, extended completeness / consistency checking, decision-making consistency based on relations between ADs.
Information content	Issue, Decision, Status, Group, Assumptions, Constraints, Positions, Argument, Implications, Related decisions, Related requirements, Notes [3]	Decisions, two kinds of relations between ADs, problems (issues), possible solutions, pros and cons of every solution, chosen solution indicated, rationale.	Classification of architectural problems, possible solutions, pros and cons of every solution, chosen solution indicated, rationale.
Classifi-cation of ADs	No classification assumed, decisions can be put into groups according to the architects' needs.	No classification assumed; decisions can be put into groups according to the architects' needs.	Problems assigned to one of the levels: Executive, Conceptual, Technology, Vendor Asset. Each problem classified with topic groups mechanism.
Relations between ADs	Does not assume any particular types relations.	Only „lead to" and "constraints" relations.	Influences, refined by, decomposes into, forces, is incompatible with, is compatible with, triggers, has outcome.
Rationale modelling	Textual.	Diagrammatic, when necessary supported by additional textual explanations.	Textual.
Model analysis and verification	Manual walkthroughs only.	Limited to syntax enforcement, consistency / completeness with automated or manual walkthroughs.	Automatic verification of decision-making consistency, completeness and consistency checking.

7 Summary: Future Work

The presented architecture decisions model MAD 2.0 has been tailored to the specific conditions of the evolution of systems of systems subject to constant urgent change typical. In such conditions, evolution turns out to be a highly creative, disordered process, performed under time pressure. MAD 2.0 can assist architecting activities, which helps capture architectural knowledge when it is created. It eliminates, or at least minimises, the need to document architectural decisions *ex post*. It also provides for basic automated verification as well as for model walkthroughs.

MAD 2.0, together with modelling, has been validated in the real life conditions of a large enterprise on the same group of architects. Their perception has changed considerably: they were satisfied with such an evolution-oriented approach – the survey indicated that about 85% found it easy to learn and use. About 70% found it useful for their job.

Future work will include:

- Further empirical evaluation;
- Analysing a larger number of evolution cases to develop a deeper insight in the architectural problems concerning the evolution of systems of systems;
- Developing a classification of architectural problems connected with systems evolution;
- Extending the information content of MAD 2.0 to widen the range of possible analyses;
- Providing a view mechanism for managing large sets of MAD 2.0 models, where users define a subset of model components that should be presented to them extending the concepts from [14] onto MAD 2.0;
- Integration of MAD 2.0 with other models of systems architecture, e.g. UML, BPMN.

Acknowledgement. This work was sponsored by the Polish Ministry of Science and Higher Education under grant number 5321/B/T02/2010/39.

References

1. Ali Babar, M., et al.: Architecture knowledge management. In: Theory and Practice. Springer, Heidelberg (2009)
2. Bosch, J., Jansen, A.: Software Architecture as a Set of Architectural Design Decisions. In: 5thWorking IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 109–120. IEEE Computer Society, Los Alamitos (2005)
3. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22(2), 19–27 (2005)
4. Zimmermann, O., et al.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. Journal of Systems and Software 82(8), 1249–1267 (2009)
5. Kruchten, P.: An Ontology of Architectural Design Decisions. In: 2nd Groningen Workshop on Software Variability Management, pp. 54–61. Rijksuniversiteit Groningen (October 2004)

6. Jansen, A., Avgeriou, P., van der Ven, J.: Enriching Software Architecture Documentation. Journal of Systems and Software 82(8), 1232–1248 (2009)
7. Zimmermann, O.: Architectural Decisions as Reusable Design Assets. IEEE Software 28(1), 64–69 (2011)
8. Mojtaba Shahin, M., Liang, P., Reza Khayyambashi, M.: Improving understandability of architecture design through visualization of architectural design decision. In: SHARK 2010 Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge. ACM, New York (2010)
9. Zalewski, A., Ludzia, M.: Diagrammatic Modeling of Architectural Decisions. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 350–353. Springer, Heidelberg (2008)
10. Zalewski, A., Kijas, S.: Architecture Decision-Making in Support of Complexity Control. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 501–504. Springer, Heidelberg (2010)
11. Harrison, N.B., Avgeriou, P., Zdun, U.: Using Patterns to Capture Architectural Decisions. IEEE Software 24(4), 38–45 (2007)
12. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering (ICSE 2000), pp. 73–87. ACM, New York (2000)
13. Garlan, D., Barnes, J.M., Schmerl, B., Celiku, O.: Evolution styles: Foundations and tool support for software architecture evolution. In: Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture, WICSA/ECSA 2009 September 14–17, pp.131–140 (2009)
14. Chen, L., Babar, M.A., Liang, H.: Model-Centered Customizable Architectural Design Decisions Management. In: 2010 21st Australian Software Engineering Conference (ASWEC), April 6-9, pp. 23–32 (2010)

Resource Management in the Air Traffic Domain

Guglielmo Lulli[1], Raffaela Mirandola[2], Pasqualina Potena[3], and Claudia Raibulet[1]

[1] Università degli Studi di Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione, Viale Sarca 336, Edificio U14, 20126, Milan, Italy
[2] Politecnico di Milano, Dipartimento di Elettronica ed Informazione, Piazza Leonardo da Vinci, 32, 20133, Milan, Italy
[3] Università degli Studi di Bergamo, Dipartimento dell'Informazione e Metodi Matematici, Viale Marconi, 5, 24024, Dalmine (BG), Italy
lulli@disco.unimib.it, mirandola@elet.polimi.it,
pasqualina.potena@unibg.it, raibulet@disco.unimib.it

Abstract. Nowadays, with the increasing need of traveling and flying, the air traffic system is highly capacity constrained due to the limited availability of resources. These resources are shared among the various domain actors. In this paper, we propose an architectural perspective for resource management in the aeronautic domain, which is based on resource allocation, trading, and adaptation to reduce the congestion phenomenon in the air traffic system.

Keywords: Resource management, resource allocation, resource trading, resource adaptation, air traffic domain.

1 Introduction

The air traffic system is facing congestion phenomena almost on a daily basis, due to the availability of scarce resources both on the ground and in en-route airspace which pose restricting capacity constraints. Capacity at airports is limited by the runway systems and the terminal airspace around them. The capacity of en-route airspace sectors is limited by the maximum workload acceptable for air traffic controllers (measured as the average number of aircraft which are permitted to fly an en-route sector in a specific period of time) [7]. Due to these capacity constraints, imbalances between demand and capacity may occur at key times and points of the air transportation network. These local overloads create delays which propagate to other parts of the air network, amplifying congestion as increasing number of local capacity constraints come into play. Last year, more than 20% of US flights were delayed or cancelled (according to the US Bureau of Transportation Statistics). Similar statistics have been reported by European authorities.

Complying with the SESAR JU Initiative in Europe and the NGATS in the US, we are developing a research project named *Enhancing the European Air Transportation System* whose goal is to design resource management schemes for the air traffic system capacity at European level. Whenever there is competition for a limited number of resources, the air traffic management has to take critical resource allocation [12] decisions. In this context, our objective is to develop an architectural

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 97–104, 2011.

solution which ensures the efficient allocation and use of resources. This solution embodies mathematical models and procedures for air traffic management, which are consistent with the philosophy of collaborative decision making (initiative aimed at improving air traffic management through increased information exchange among the various parties in the aviation community [3]). Approaches might vary from system-wide optimization to market-based mechanisms.

The proposed solution aims to address the variability of resources through architectural mechanisms, by defining appropriate components for the dynamic allocation, trading, and adaption of resources [4]. Our resource management approach aims to solve, as a downstream effect, the planning problem locally by the parties involved (airlines) instead of centrally by a single decision maker (the Air Navigation Service Provider (ANSP)). The management mechanisms enable a multi-step trading process in order to satisfy all the capacity constraints, assuring the safety of flights (claimed by the Eurocontrol, i.e., the European agency devoted to the air traffic management) and minimizing delays (claimed by the airlines). In the proposed approach we consider multiple adaptations. A first one is structural. It concerns the ANSP and is achieved by airspace reconfiguration, see e.g., [6], in order to satisfy the air traffic demand. A second one is a process adaptation which is achieved by changing dynamically the optimization models and methods based on the current external and internal conditions of the air traffic management system.

The proposed approach is new. In fact, currently, no resource trading scheme is implemented in the considered application domain. Indeed, a simple exchange of resources is only performed at the airport level based on the exchange of updated information between airlines and the ANSP. Airlines release airport resources (arrival time slots) in return of a priority on possibly new released resources which are treated according to a first-in-first-served strategy. Furthermore, the research efforts have been devoted either to support the air traffic flow management (e.g., [1]), airspace configuration as a measure to alleviate air traffic congestion (e.g., [6]), or market based mechanism to improve the efficiency of resource allocation [11]. However, to the best of our knowledge, no integrated approach has been proposed for the application domain herein considered.

The rest of the paper is organized as follows. Section 2 provides an overview on the air traffic management domain. Section 3 introduces our solution to resource management. Conclusions and further work are dealt in Section 4.

2 The Application Domain

The main actors of the aeronautical application domain herein considered are the airlines, on one side, and the Air Navigation Service Provider (ANSP), on the other side. ANSP performs the Air Traffic Flow Management (ATFM) and the Air Traffic Control (ATC) functions. In execution the ATFM function, the ANSP requires a global vision of the air traffic and its main task is to adjust the air traffic flows in order to meet all the system capacity constraints. As a by-product of its functions, the ANSP ensures that the available resources are used efficiently. At the European level the ATFM function is carried out by Eurocontrol. ATC guarantees that aircraft are separated to prevent collisions, and provides information and other support for pilots.

In the rest of the paper, we focus on the ATFM function. To fulfill this function the ANSP gathers weather information and establishes the capacity of the system i.e., the number of aircraft which can feasibly fly in each sector, and the allowed number of departures and arrivals for each airport. Furthermore, ANSP receives as input, information about the status of flights from the airlines, and tries to address airlines' requests based on a first-in-first-served policy.

The main objective of airlines (besides the safety of the flights) is to be cost-effective. Costs are directly influenced by delays of the airline flights. Airlines to achieve their individual objectives try to collect the resources they need.

The current solution for resource management, which -at the moment- consists exclusively of the allocation mechanism, (see Figure 1) has the following limitations:

- The airspace configuration and thus the capacity of the system is established statically and it is independent of the system's conditions. Our scope is to enable a dynamic approach which reconfigure the airspace whenever the conditions of the air traffic system change;
- The resource allocation is performed as follows: ANSP receives the requests from airlines and tries to accommodate their request based on a first-in-first-served strategy. However, this approach lacks of a complete knowledge of airlines objectives. As opposed, the approach herein proposed enables a trading process which involve actively airlines in the resource allocation process;
- Airlines may only "exchange" (airlines release resources (arrival time slots) in return of a priority on possibly new available time slots) time slots only at a single airport level. In our approach, we enable airlines to globally (e.g., European level – entire air traffic system) trade resources (time slots for any flight phase, e.g., take off, cruise and/or landing) among them.

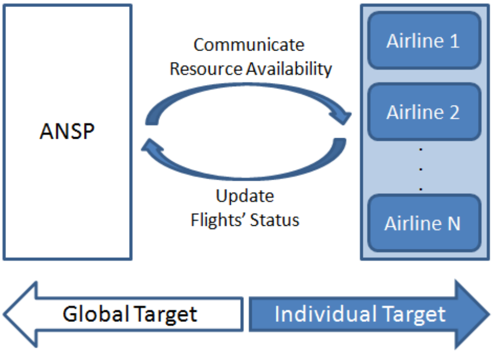


Fig. 1. The Current Available Resource Management Solutions in the Air Traffic Domain

3 Our Solution

The architectural solution we propose for resource management in the air traffic domain is displayed in Figure 2. It implements the following six main steps:

Step 0: ANSP retrieves the initial scheduling of flights, which is obtained by the official timetable planned long time in advance.

Step 1: ANSP collects the current weather conditions of the airspace, and based on this information establishes the capacity of the system (sectors and airports).

Step 2: Based on the current system capacity and the last updated schedule of flights, the ANSP can either approve the schedule or suggest modifications to the schedule. ANSP communicates the system capacity and possible schedule modifications to airlines.

Step 3: Airlines trade the available resources and communicate to the ANSP the attained exchanges and modifications of the flight schedule.

Step 4: Based on the requests from the airlines (e.g., requests for new allocations or the results of the trading process), the ANSP tries to adapt the resources in order to meet the airlines' requests.

Step 5: If a time interval greater then X has passed from the last weather condition check, continue with Step 1; otherwise continue with Step 2.

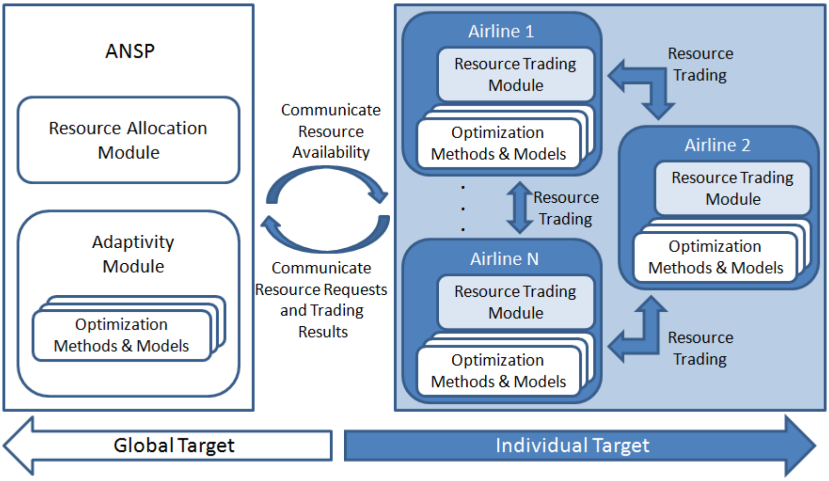

ANSP
Resource Allocation Module
Adaptivity Module
Optimization Methods & Models
Communicate Resource Availability
Communicate Resource Requests and Trading Results
Airline 1
Resource Trading Module
Optimization Methods & Models
Resource Trading
Airline 2
Resource Trading Module
Optimization Methods & Models
Resource Trading
Airline N
Resource Trading Module
Optimization Methods & Models
Resource Trading
Global Target
Individual Target

Fig. 2. The Main Architectural Modules of Our Resource Management Solution

The details of the resource allocation, trading and adaptation modules are described in the rest of this section.

3.1 The Air Traffic Resource Allocation Module

The Resource Allocation (RA) module assigns the available resources (e.g., time slots) to the airlines based on global objectives (e.g., the efficient usage of the resources, minimization of the total delay, safety). To execute this task, RA has to know the system capacity to be allocated and the pool of airlines (see Figure 3). In addition to this information, RA receives in input the current flight schedule, and the capacity constraints. The RA module is equipped with a supervision capability on the use of resources thus guaranteeing a feasible and efficient allocation of resources. This functionality requires in input the results of the trading process and the new requests for resource allocations, as well as the results of the adaptation process (e.g., system capacity modifications). To provide these functionalities, RA uses assignment optimization models and methods [1, 5] customized for the air traffic domain.

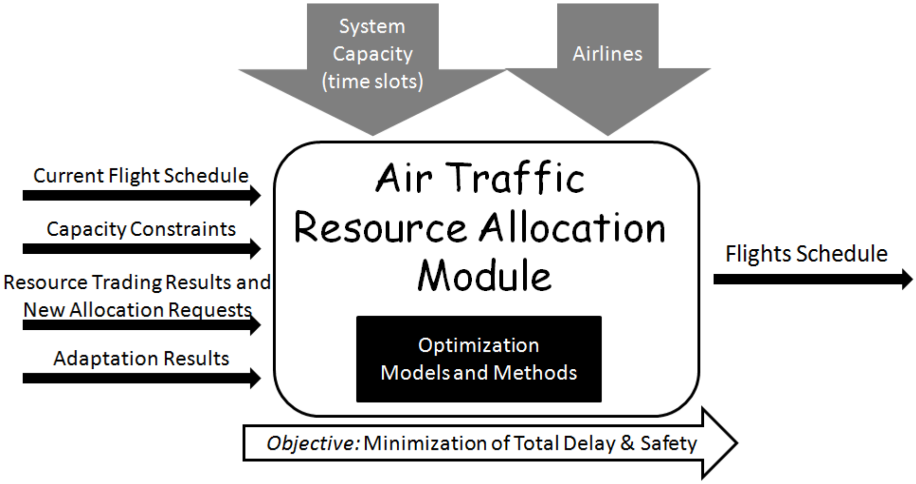

System Capacity (time slots)
Airlines
Current Flight Schedule
Capacity Constraints
Resource Trading Results and New Allocation Requests
Adaptation Results
Air Traffic Resource Allocation Module
Optimization Models and Methods
Flights Schedule
Objective: Minimization of Total Delay & Safety

Fig. 3. The Air Traffic Resource Allocation Module

3.2 The Air Traffic Resource Trading Module

The airlines receive from the ANSP the input for executing a new resource trading. ANSP suggests an initial solution and the possible bottlenecks in the air traffic system due to imbalances between available capacity and the airline requests (flights demand).

The Resource Trading (RT) module, depicted in Figure 4, is used by the airlines to exchange resources with other airlines and to communicate with ANSP. Each airline has an RT module. The communication with ANSP includes resource allocation/modification/usage/cancellation requests, which may or may not be a result of a trading process. The communication with the other airlines concern exchange requests, thus implying a trading process. Each RT receives in input the current flights schedule, if any, the capacity constraints, and the resource trading requests from other airlines. The airline decides on exchanging resources with other airlines based on its individual objectives (e.g., costs minimization) and exploiting customized optimization models and methods, i.e., optimization models include specific constraints and objectives of the airline. Therefore, different airlines may have different optimization models [2, 13]. An exchange is performed if it is convenient for both parties.

Airlines may be equipped with several optimization models and methods which can be interchanged at runtime to trade resources, hence they can adapt themselves dynamically according to the cost of their solution.

There are cases in which an airline does not find an appropriate solution. This means that the costs generated by the expected delays are too high to be afforded and the airline may decide to cancel one or more flights. Even if this solution does not lead to the achievement of the individual target, it may be beneficial for the overall target at the global level. Coherently with the collaborative decision making philosophy, the airline which cancels one or more of its flights can get a priority on new released resources.

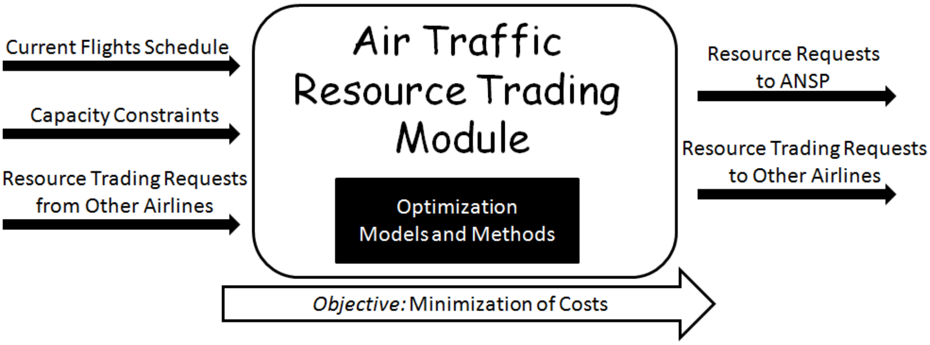

Current Flights Schedule
Capacity Constraints
Resource Trading Requests from Other Airlines
Air Traffic Resource Trading Module
Optimization Models and Methods
Resource Requests to ANSP
Resource Trading Requests to Other Airlines
Objective: Minimization of Costs

Fig. 4. The Air Traffic Resource Trading Module

3.3 The Air Traffic Adaptivity Module

ANSP tries to accommodate the current schedule, verifying its feasibility by estimating the use of available resources and the overall delays of the flights. If this verification has a positive answer, then ANSP approves the schedule.

Otherwise, it may consider two possible alternatives determined by a distance measure (*d*) between the current scheduled solution (*S*) and the current capacity of the system (*C*). If this distance is lower than a given threshold $d(S, C) < \delta$, then the ANSP tries to structurally adapt itself (by airspace reconfiguration) to accommodate the current solution which can be either the initial schedule or the outcome of the resource trading process. If $d(S, C) \geq \delta$ then the ANSP is not able to satisfy the current schedule of flights by a structural adaptation. In this case, it requests to the airlines a new trade of resources to reach a new feasible schedule of flights. The ANSP may suggest a feasible schedule of flights according to a first-in-first-served strategy and may highlight possible shortages of capacity. The adaptivity module is depicted in Figure 5.

We here describe in more details what we mean for structural adaptivity. The adaptation consists in the reconfiguration of the airspace. For example, if the European airspace is structured as in Figure 6a, after the application of the adaptive strategies, its structure may look like as in Figure 6b.

To better understand the possible benefits attainable by airspace reconfiguration, let’s consider the following example. The capacity of each sector (air traffic controller workload) is equal to 10 flights. Suppose sectors 1, 2, 4, and 5 have 5 flights equally distributed in the sector, while sectors 3 and 6 have 20 flights per sector. Under configuration 6a, we have an imbalance between capacity and demand, but reconfiguring the airspace as in Figure 6b the demand of each sector meets the available capacity of the sector by supervising 10 flights each. Sectors 1, 2, 4, and 5 have been extended to cover 10 flights, while sectors 3 and 6 have been reduced to cover the same number of flights as the other sectors.

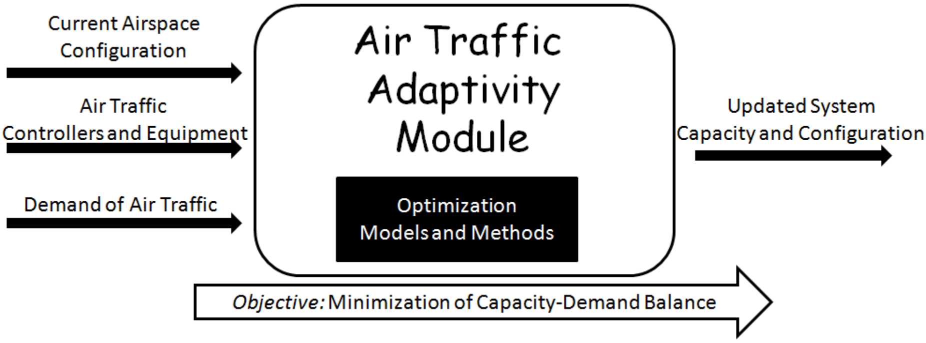
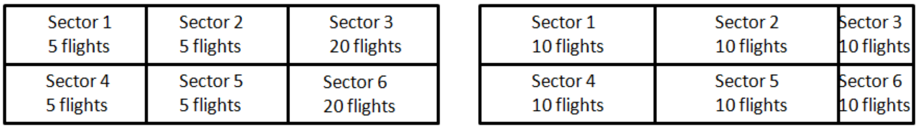

Current Airspace Configuration
Air Traffic Controllers and Equipment
Demand of Air Traffic
Air Traffic Adaptivity Module
Optimization Models and Methods
Updated System Capacity and Configuration
Objective: Minimization of Capacity-Demand Balance

Fig. 5. The Air Traffic Adaptation Module

To achieve this structural adaptivity the ANSP has be enriched/equipped with various optimization methods and models for example by combining existing approaches as described in [6, 10]. We consider that the ANSP is equipped with several models and optimization methods which differ for the computational time required to compute the solution, and for the quality of the computed solution. The ANSP should be able to decide dynamically which of these models and methods to apply based on either the time available to propose a solution, the distance measure, or both. This is translated into an adaptation process of the ANSP.



Sector 1 5 flights	Sector 2 5 flights	Sector 3 20 flights
Sector 4 5 flights	Sector 5 5 flights	Sector 6 20 flights

a.

Sector 1 10 flights	Sector 2 10 flights	Sector 3 10 flights
Sector 4 10 flights	Sector 5 10 flights	Sector 6 10 flights

b.

Fig. 6. Air-Space Adaptation Re-Configuration

4 Conclusions and Further Work

The solution described in this paper for the resource management in the air traffic application domain is characterized by at least two main advantages. First, it actively involves the various actors of the system. Hence, it becomes a collaborative and distributed approach which enables each actor (airline) to achieve its own objectives. As a consequence, through this approach we may reach a better performance of the air traffic management system: it is not a minimization of the total delay, but it is a minimization of the total delay cost, which depends on the costs structure and the operations of each airline. Second, the solution is based both on a top-down approach because of the global perspective of ANSP on the system, as well as a bottom-up approach because of the individual perspective of the airline companies.

Our solution for resource management in the air traffic domain may be generalized and applied to further application domains characterized by a limited number of resources which should be exploited by different stakeholders.

One of the further developments is related to the extension of the resource management process also to the ground-side resources. This extension introduces further actors in the system (e.g., handling services providers), which have their own objectives that may be considered orthogonal to the previous ones. Integrating also the ground-side resources will enable us to have a uniform and complete solution for the resource trading in the aeronautical application domain.

A prototype of the solution described in this paper is currently under development at the University of Milano-Bicocca. It exploits adaptation mechanisms similar to the ones we have developed for [8, 9].

Acknowledgments. This work is partially supported by the *Enhancing the European Air Transportation System* (EATS) research project funded by the Italian Government within the PRIN program.

References

1. Bertsimas, D., Lulli, G., Odoni, A.: An Integer Optimization Approach to Large-Scale Air Traffic Flow management. Operations Research 59(1), 211–227 (2011)
2. Castelli, L., Pesenti, R.: Allocating Air Traffic Flow Management Slots. Department of Applied Mathematics, University of Venice, Woking Paper No.191 (2009)
3. Chang, K., Howard, K., Oiesen, R., Shisler, L., Tanino, M., Wambsganss, M.C.: Enhancements to the FAA Ground-Delay Program under Collaborative Decision Making. Interfaces 31(1), 57–76 (2001)
4. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
5. Dell'Olmo, P., Lulli, G.: A Dynamic Programming Approach for the Airport Capacity Allocation Problem. IMA Journal of Management Mathematics 14, 235–249 (2003)
6. Gianazza, D.: Forecasting Workload and Airspace Configuration with Neural Networks and Tree Search Methods. Journal of Artificial Intelligence 174(7-8), 530–549 (2010)
7. Hansman, R.J., Odoni, A.: Air Traffic Control in the Global Airline Industry. In: Belobaba, P., Odoni, A., Barnhart, C. (eds.), pp. 377–403. Wiley, Chichester (2009)
8. Raibulet, C., Arcelli, F., Mussino, S., Riva, M., Tisato, F., Ubezio, L.: Components in an Adaptive and QoS-based Architecture. In: Proc. of the ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 65–71. IEEE Press, Los Alamitos (2006)
9. Mirandola, R., Potena, P.: Self-Adaptation of Service based Systems based on Cost/Quality Attributes Tradeoffs. In: Proc. of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 493–501 (2010)
10. Raibulet, C.: Facets of adaptivity. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 342–345. Springer, Heidelberg (2008)
11. Waslander, S., Roy, K., Johari, R., Tomlin, C.: Lump-Sum Markets for Air Traffic Flow Control with Competitive Airlines. Proc. of the IEEE 96(12), 2113–2130 (2008)
12. Webb, A., Sarkani, S., Mazzuchi, T.: Resource Allocation for Air Traffic Controllers using Dynamic Airspace Configuration. In: Proceedings of the World Congress on Engineering and Computer Science, vol. II (2009)
13. Vossen, T., Ball, M.O.: Slot Trading Opportunities in Collaborative Ground Delay Programs. Transportation Science 40(1), 29–43 (2006)

An Architecture-Based Verification Technique for AADL Specifications[⋆]

Andreas Johnsen, Paul Pettersson, and Kristina Lundqvist

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
{andreas.johnsen,paul.pettersson,kristina.lundqvist}@mdh.se

Abstract. Quality assurance processes of software-intensive systems are an increasing challenge as the complexity of these systems dramatically increases. The use of Architecture Description Languages (ADLs) provide an important basis for evaluation. The Architecture Analysis and Design Language (AADL) is an ADL developed for designing software-intensive systems. In this paper, we propose an architecture-based verification technique covering the entire development process by adapting a combination of model-checking and model-based testing approaches to AADL specifications. The technique reveals inconsistencies of early design decisions and ensures a system's conformity with its AADL specification. The objective and criteria (test-selection) of the verification technique is derived from traditional integration testing.

1 Introduction

The architecture design phase is one of the most critical phases in the development process of software-intensive systems. The architecture specification is the initial development artefact representing the earliest design decisions made on the intended system's structure, functional properties and quality attributes (also known as non-functional properties or extra-functional properties). Design decisions involve the allocation of functional properties – which are closely related to a system's behavior, capabilities and services – to certain structures to achieve certain quality attributes. Furthermore, the architecture specification is used as a mutual communication blueprint among stakeholders and guides the implementation phase of the system. Consequently, the developed system will heavily depend on the architecture specification, which it ideally should conform to.

The design decisions established in the architecture design phase, or the absence of some, may impose incorrect properties of the system and thereby creating challenges in quality assurance processes. These incorrect structural, functional as well as non-functional properties may go unnoticed until later phases of the development process where a correction is known to be significantly more

[⋆] This work was partially supported by the Swedish Research Council (VR), and Mälardalen Real-Time Research Centre (MRTC)/Mälardalen University.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 105–113, 2011.

costly compared to a correction in the architecture design phase. Hence, evaluating the architecture specification is crucial in order to detect possible faults and inconsistencies before the development process progresses, reducing a significant amount of cost and time. Furthermore, in order to preserve the valuable effort made at the architecture design phase, an implementation of the system must be implemented in conformance with the architecture specification. The verification techniques used to tackle these challenges, i.e. *1) to evaluate an architecture specification* and *2) to test the conformance of an implementation with respect to its architecture specification*, rely on what kind of properties and their relations that may be described with the Architecture Description Language (ADL) used to specify the system's architecture.

Software-intensive systems are systems where software interacts with sensors, actuators, devices, other systems and people. Examples of such systems are embedded systems for aerospace, automotive and telecommunications. What these systems have in common is that they often are operating in dynamic, time- and safety-critical environments. One ADL that has been developed for this kind of systems, is the Architecture Analysis and Design Language (AADL) [1], which is widely used both within industry and the research community. In this paper we propose an architecture-based verification technique, for software-intensive systems specified by AADL, addressing challenge *1)* and *2)* mentioned above. The technique is based on formal constructs enabling automation of the verification activities where challenge *1)* and *2)* are tackled by adapting model-checking and model-based testing approaches to an architectural perspective. The objective of the technique is to evaluate the integration of components at both the specification-level and the implementation-level. Automated simulation of AADL specifications is not feasible directly from the artefact since AADL lacks formal semantics and implemented semantics. Formal semantics of a subset of AADL and an implementation thereof can be found in [2].

The rest of this paper presents an overview of AADL in Section 2. The architecture-based verification technique is introduced in Section 3, defined verification criteria are presented in Section 4, followed by concluding remarks in Section 5.

2 Preliminaries

AADL [3] was released and published as a Society of Automotive Engineers (SAE) Standard AS5506 [1] in 2004. It is a textual and graphical language used to model, specify and analyze software- and hardware-architectures of real-time embedded systems. The AADL language is based on a component-connector paradigm that describes components, component interfaces and the interactions (connections) among components. Hence, the language captures functional properties of the system, such as input and output through component interfaces, as well as structural properties through configurations of components and connectors. Furthermore, means to describe quality attributes, such as timing and reliability, are also provided. AADL defines ten types of component abstractions which can be divided into three groups:

- **Application software:** process, thread, thread group, data and subprogram
- **Hardware/Execution platform:** processor, bus, memory and device
- **Composite:** system

3 The Architecture-Based Verification Technique

This section presents an overview of the automatable verification technique for AADL specifications. The technique comprises both evaluation of specifications and the systems' conformity to them. It is depicted as a flowchart in Figure 1, where initially a system's intended architecture is specified using AADL. Such an artefact is commonly specified through a translation from something cognitive, an idea, a need or an informal/semi-formal requirement specification, but since it is informal, it is not possible to formally prove that the AADL specification correctly conforms to the informal one it is derived from [4]. Consequently, making this type of evaluation far from possible to automate and thus is out of scope in this technique. What is possible though is to formally reason about a system solely through the AADL specification, to prove its consistency and completeness, and later use it as a test model to perform model-based testing on. The different steps of the verification technique are as follows: The first step is to use the mappings/transformation rules (described in [2]) to transform an AADL specification to a timed automata model upon which automated formal verification can be performed.

The second step is to apply the architecture-based verification criteria (section 4) to the AADL specification. They define the test selection, i.e., what samples of the specification to evaluate and how they are extracted, and the coverage requirement, i.e., how many samples to evaluate. The samples generated from the criteria are sequences of component-integrations in terms of control-flows and data-flows.

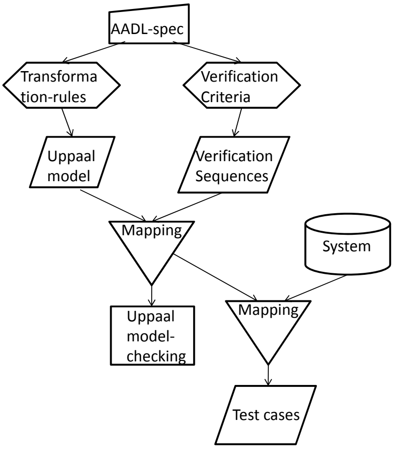


Fig. 1. Flowchart of the technique

Sequences are transformed, in the third step, to the corresponding timed automata paths through a structural mapping between them.

The outcome, a set of timed automata paths are required in the fourth step to be fully simulated by the Uppaal model-checker, by using temporal logics, in order to satisfy the criteria. The verdict from the simulations reveals the consistency and completeness of the AADL specification, where a correction of the specification should be made if it is shown inconsistent or incomplete.

The paths are later used in the fifth step to generate test cases to the implementation (model-based testing), to test the conformance of the implementation with respect to the architecture specification. Test paths are transformed to concrete test cases through a mapping between the architecture specification and its implementation (we assume identical name spaces between the AADL specification and the system).

4 AADL Verification Criteria

Due to features of an ADL, the primary focus of evaluation at this level is the integration of components as described by Eickelmann and Richardson [5] in their work about architecture-based defect prevention and detection. The idea of taking traditional data-flow and control-flow analysis criteria to the architectural-level has been proposed by Jin and Offut in [6], where explicit data-flow and control-flow properties through system architectures are defined. Based on these properties, they define general architecture-based testing criteria applicable to any ADL treating components and connectors as separate entities interconnected through their interfaces. Since AADL connectors do not have interfaces, and are dependent on the interfaces of the components they connect, the defined criteria are not applicable to AADL specifications. From the definition of the general criteria defined by Jin and Offut, we define architecture-based verification criteria specific to AADL based on the possible bindings of data-flow and control-flow properties (leading to sequences) described by AADL.

4.1 Verification Objectives

AADL specifications have explicit control-flows and data-flows through the architecture described by the informal semantics of AADL. These flows are dependent on how components transfer control and data through their interfaces (AADL component features). The possible interactions among components are represented by, and restricted to, four different types of connections: *port connections*, *data access connections*, *subprogram calls* and *parameter connections*.

Component abstractions within the software group, except data components, may have port interfaces for directional (in port, out port or inout port) interactions of typed data and events. A port can either be a data port (for transfer of data), a data event port (for transfer of data and associated control) or an event port (for transfer of control). Port interfaces can be connected through *Port connections*, which describe the transfer of control and data among/through concurrently executing thread components, or between a thread component and device component (threads and subprograms are used to represent functionality executed with or by device components). The flow of data or control through a port connection is determined by the directions of the connected ports.

Data components representing static data sharable among components are not accessible though ports, they are accessible through data access interfaces that may be declared with components of the software group. *Data access connections* describe the transfer of data where the data flow is determined by the value (read or write) of an *Access Right* property associated with the connection.

Subprogram components are not declared as subcomponents, instead they are called from thread or subprogram components through explicit *subprogram calls* declarations, expressing a flow of control from the calling component to the called subprogram. Call declarations may implicitly describe flows of data, where data can be provided to or received from a subprogram through parameter or data access connections. Parameters are interfaces of subprograms, similarly to data ports, for directional data interactions, where a parameter can be connected to a data port or another parameter through *parameter connections* describing the transfer of data. The data flow through a parameter connection is determined by the directions of the connected interfaces.

The runtime configuration of subcomponents and their interactions within a component may change if it is specified with *modes*. For each mode, it is possible to set the active components and connections, mode-specific subprogram calls and mode-specific properties. The transition from one mode to another is triggered by events derived from event ports, which is specified in a mode state machine. These modes can also be used to describe the internal logical execution (functional behavior) of thread and subprogram components. In addition to modes, a behavioral annex (BA) [7] extending the expressiveness of mode state machines has been developed to specify logical execution through automata syntactically similar to mode state machines. Thereby, it is possible to refine logical execution through state variables, states and transitions operating on a component's interfaces. Transitions can be specified with guards, such as boolean expressions and events, as well as actions, such as assignments and subprogram calls. Consequently, the control- and data-flows specified with the four different connections (described above) are refined if a behavioral model operates on the interfaces the connections connect.

The four different types of connections specify the architectural control-flows and data-flows of an AADL specification. Architectural control-flows are the different execution orders of architectural elements whereas architectural data-flows are the relations between definitions of data elements in a source component and uses of the corresponding data elements in a target component. These flows may be dependent on mode state machines, refined by the BA and constrained by associated properties where conflicts may occur between these constructs. The objective of the verification criteria is to ensure consistency and completeness of and between the flows, their refinements and their constraints through analysis of control-flow reachability, data-flow reachability and concurrency among flows:

Control-flow reachability: Every architectural element in an execution order should be able to reach the subsequent element to be executed in the order. The subsequent element should be reached without conflicting properties (constraints) of the execution order.

Data-flow reachability: Every data element should be able to reach its target component, where the data is used, from its source component, where the data is defined. The target component should be reached without conflicting properties of the data flow.

Concurrency among flows: Analysis of single interactions of data or control is not enough since there are implicit relations between them that may cause deadlocks in the system. The relations between the flows should not prevent control-flow reachability or data-flow reachability, and where the system should be free from deadlocks.

4.2 Verification Criteria

In order to extract control-flows and data-flows from an AADL specification, we define the atomic bindings of control and data that generates the flows, where we refer these atomic bindings to AADL relations. The relations are used to define integration verification sequences of control-flow and data-flow upon the verification criteria are defined.
In the definitions of AADL relations, an AADL Specification is represented as a 5-tuple:

$$AADLSPEC = \langle N, I, C, BAC, PAC \rangle$$

Where:
N is the set of Components $= \{n_1, n_2, ..., n_n\}$

I is the set of component interfaces $= \{n_x.i \mid n_x.i$ is a port, data acess, subprogram or parameter interface of n_x and $n_x \in N\}$

C is the set of Connections $= \{c(s,d) \mid c(s,d)$ is a port-, a data access-, a subprogram call- or a parameter-connection connecting the source interface $s \in I$ to the destination interface $d \in I\}$

BAC is the set of BA Connections $= \{bac(s,d) \mid bac(s,d)$ is an automaton path and the initial location in the path or a transition from the initial location is labeled with $s \in I$ and the last location in the path or a transition to the last location is labeled with $d \in I\}$

PAC is the set of Property Associated Constructs $= \{pac \mid pac \in I \cup C \cup BAC$ and is constrained by at least one associated property$\}$

Based on this representation of an AADL specification, the defined relations are:

1. **Connection Transfer Relation:** defines the data or control transfer that is generated between two interfaces connected trough a connection.
   CTR is the set of Connection Transfer Relations where $CTR \subseteq I \times I$ such that $\langle n_x.i_1, n_y.i_2 \rangle \in CTR$ iff $c(n_x.i_1, n_y.i_2) \in C$
2. **Connection Property Relation:** defines the constrained data or control transfer that is generated between two interfaces connected trough a connection.

CPR is the set of Connection Property Relations where $CPR \subseteq I \times I$ such that $\langle n_x.i_1, n_y.i_2\rangle \in CPR$ iff $c(n_x.i_1, n_y.i_2) \in C$ and $n_x.i_1$or $n_y.i_2$ or $c(n_x.i_1, n_y.i_2) \in PAC$

3. **Component Internal Relation:** defines the (possibly constrained) data or control transfer that is generated between two interfaces of a component that are connected through a connection or a BA.
   CIR is the set of Component Internal Relations where $CIR \subseteq I \times I$ such that $\langle n_1.i_1, n_1.i_2\rangle \in CIR$ iff $\langle n_1.i_1, n_1.i_2\rangle \in CTR \cup CPR$ or $\langle n_1.i_1, n_1.i_2\rangle \in BAC$
4. **Direct Component to Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are directly connected through a connection.
   $DCCR$ is the set of Direct Component to Component Relations where $DCCR \subseteq I \times I$ such that $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in CTR \cup CPR$
5. **Indirect Component to Component Relation:** defines the (possibly constrained) data or control transfer that is generated between two components that are indirectly connected through one or several component(s). The relation is recursive in order to cover any number of interconnected components. The base case is:
   $ICCR$ is the set of Indirect Component to Component Relations where $ICCR \subseteq I \times I \times I^*$ such that $\langle n_1.i_1, n_3.i_4, t\rangle \in ICCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ and $\langle n_2.i_2, n_2.i_3\rangle \in CIR$ and $\langle n_2.i_3, n_3.i_4\rangle \in DCCR$ and $t = \langle\langle n_1.i_1, n_2.i_2\rangle, \langle n_2.i_2, n_2.i_3\rangle, \langle n_2.i_3, n_3.i_4\rangle\rangle$
   The recursive definition is:
   $ICCR$ is the set of Indirect Component to Component Relations where $ICCR \subseteq I \times I \times I^*$ such that $\langle n_1.i_1, n_x.i_y, t\rangle \in ICCR$ iff $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$ and $\langle n_2.i_2, n_2.i_3\rangle \in CIR$ and $\langle n_2.i_3, n_x.i_y, t'\rangle \in ICCR$ and $t = \langle\langle n_1.i_1, n_2.i_2\rangle, \langle n_2.i_2, n_2.i_3\rangle, \langle t'\rangle\rangle$

From these AADL relations three verification sequences are derived, which are paths of the architecture specification:

1. **Component Internal Transfer Path:** If there exists a $\langle n_1.i_1, n_1.i_2\rangle \in CIR$, there exists a path from $n_1.i_1$ to $n_1.i_2$. The path is constrained if $\langle n_1.i_1, n_1.i_2\rangle$
   $\in CPR$.
2. **Direct Component to Component Path:** If there exists a $\langle n_1.i_1, n_2.i_2\rangle \in DCCR$, there exists a path from $n_1.i_1$ to $n_2.i_2$. The path is constrained if $\langle n_1.i_1, n_2.i_2\rangle \in CPR$.
3. **Indirect Component to Component Path:** If there exists a $\langle n_1.i_1, n_x.i_y, t\rangle \in ICCR$, there exist a path from $n_1.i_1$ to $n_x.i_y$ via t. The path is constrained if any pair in $t \in CPR$.

The AADL specification is consistent if each path is free from contradictory behavior, that is, each path does not contradict Control-flow reachability, Data-flow reachability and Concurrency among flows. The AADL specification is complete if each path not yielding an end-to-end flow (typically a sensor-to-actuator flow) is subsumed in another path.

Upon the integration verification sequences, we define the three architecture-based verification criteria, which specifies requirements for a set of simulations or test cases to be adequate. Within following definitions, "S" is either a set of simulations of an AADL specification or a set of test cases for an implementation implemented to conform an AADL specification.

- **Component Internal Coverage:** requires that S covers all *Component Internal Transfer Paths.*
- **Direct Component to Component Coverage:** requires that S covers all *Direct Component to Component Paths.*
- **Indirect Component to Component Coverage:** requires that S covers all *Indirect Component to Component Paths.*

5 Conclusion

The AADL language is a formalism for development of safety-critical software-intensive systems. In this paper we have presented a verification technique covering the entire development process of a system specified with this formalism. The technique evaluates the consistency and completeness of an AADL specification and tests a systems' conformity to it. The entire development process is covered by adapting a combination of model-checking and model-based testing approaches to an architectural perspective. The adaption is performed through the definition of AADL-specific verification criteria. We are currently validating the technique against a system developed by a major vehicle manufacturer. The next step is to enrich the verification criteria with further details as well as formally define consistency and completeness of AADL specifications.

References

1. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards no. AS5506 (November 2004)
2. Johnsen, A., Pettersson, P., Lundqvist, K.: An Architecture-based Verification Technique for AADL Specifications. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-253/2011-1-SE, Mälardalen University (May 2011)
3. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The Architecture Analysis and Design Language (AADL): An Introduction. Technical report, Technical report (2006)
4. Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Trans. Softw. Eng. 22(11), 777–793 (1996)
5. Eickelmann, N.S., Richardson, D.J.: What makes one software architecture more testable than another? In: ISAW 1996: Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints 1996) on SIGSOFT 1996 Workshops, pp. 65–67. ACM, New York (1996)

6. Jin, Z., Offutt, J.: Deriving Tests From Software Architectures. In: ISSRE 2001: Proceedings of the 12th International Symposium on Software Reliability Engineering, p. 308. IEEE Computer Society Press, Washington, DC, USA (2001)
7. Franca, R.B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., Thomas, D.: The AADL behaviour annex – experiments and roadmap. In: ICECCS 2007: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 377–382. IEEE Computer Society Press, Washington, DC, USA (2007)

Change Impact Analysis in Product-Line Architectures

Jessica Díaz[1], Jennifer Pérez[1], Juan Garbajosa[1], and Alexander L. Wolf[2]

[1] Technical University of Madrid (UPM) - Universidad Politécnica de Madrid
Systems & Software Technology Group (SYST), E.U. Informática, Madrid, Spain
yesica.diaz@upm.es, {jenifer.perez,jgs}@eui.upm.es
[2] Department of Computing
Imperial College of London, London, UK
a.wolf@imperial.ac.uk

Abstract. Change impact analysis is fundamental in software evolution, since it allows one to determine potential effects upon a system resulting from changing requirements. While prior work has generically considered change impact analysis at architectural level, there is a distinct lack of support for the kinds of architectures used to realize software product lines, so-called product-line architectures (PLAs). In particular, prior approaches do not account for *variability*, a specific characteristic of software product lines. This paper presents a new technique for change impact analysis that targets product-line architectures. We propose to join a *traceability-based algorithm* and a *rule-based inference engine* to effectively traverse modeling artifacts that account for variability. In contrast to prior approaches, our technique supports the mechanisms for (i) specifying variability in PLAs, (ii) documenting PLA knowledge, and (iii) tracing variability between requirements and PLAs. We demonstrate our technique by applying it to the analysis of requirements changes in the product-line architecture of a banking system.

Keywords: Product-line architectures, product-line evolution, change impact analysis.

1 Introduction

Coping with changing requirements is an essential issue in the evolution of software systems. The causes of requirements change range from those that are technical, due to the changing market of technology platforms, to those that are business, due to the inherent volatility of the business context. A deep understanding of the software architecture in terms of structure, behavior, key properties, and relationship with the execution environment helps us to address software evolution. Also helpful is an understanding of the impact of change. Change impact analysis (CIA) is fundamental in software evolution, since it allows one to determine the potential effects upon a system resulting from a proposed change [18]. CIA can be used to predict the effects of a change before

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 114–129, 2011.

it is implemented, as well as possibly giving an estimate of the effort/cost to implement the change.

Most CIA approaches focus on source-code analysis [14,15], often limited to the dependence (data and control flow) or call relationships, and therefore tied to the coding technique and programming language. This restricts the kinds and visibility of changes that can be analyzed, and makes it difficult to locate the impacted code when the changes originate in higher-level abstractions, such as requirements, rather than the low-level code abstraction.

As software architectures bridge the gap between requirements and implementation, reasoning about the architecture can provide the high-level insight necessary to make better requirements-driven evolution decisions [3]. Therefore, CIA techniques need to be adapted to assess *architectural knowledge*. This knowledge should encompass not only the design of the solution, but also the *decisions* driving the design, the *rationale* behind those design decisions [16,37], and the *dependencies* among the design decisions. In recent years, researchers have emphasized the need for documenting architectural knowledge of this sort to maintain and evolve architectural artifacts [1,7,13,33] and to avoid architectural erosion, drift, or aging [26].

Prior work has addressed CIA at architectural level by means of *traceability* mechanisms [5,34]. Traceability connects development artifacts that contain the information necessary to analyze change impact. At architectural level, the traceability aim is to understand the relationship between requirements and their architectural realization, in both forward (from requirements to architecture) and backward (from architecture to requirements) directions. Traceability is desirable in those software architectures that realize a software product line (SPL), widely known as product-line architectures (PLAs). However, common traceability approaches to aid CIA do not recognize *variability*, which is a specific characteristic of SPLs, serving to link requirements with PLAs [20]. Correspondingly, assessing change impact is still a challenge in the evolution of SPLs [6]. While SPL engineering helps to significantly reduce cost and time when products of the same family or domain are developed, it also increases the complexity when SPLs must evolve. Bosch [4] asserts that SPLs demand even more knowledge of architecture and design decisions than that of general reuse-based software engineering. This is due to the need to manage variability and dependencies between product variants. As a result, variability must be considered as a critical element in SPL change impact analysis.

This paper addresses change impact analysis in PLA evolution from the structural point of view. Specifically, we propose to join a *traceability-based algorithm* and a *rule-based inference engine* with the aim of traversing the PLA models via a set of traceability links and propagation rules. To achieve this, variability plays a key role, so that the models of PLA and traceability must be able to completely support variability. This is why we consider mechanisms for (i) specifying variability in PLAs, (ii) documenting PLA knowledge (PLAK), and (iii) tracing variability between requirements and PLAs.

Our work has previously formulated partial solutions for each of these mechanisms: the Flexible-PLA metamodel [24,25], which addresses the problem of specifying variability in PLAs; and the PLAK metamodel[1], which supports the documentation of design decisions associated with variability, the design rationale behind the variability, the dependencies between design decision, and the definition of the basic traceability linkage between requirements and PLAs. Together these metamodels form a basis for identifying the architectural artifacts that are impacted by changes in SPL requirements, as well as in product-specific requirements. What remains is to define the technique for effectively *traversing* the models to obtain a CIA method that can account for variability.

The structure of this paper is as follows: Section 2 reviews the Flexible-PLA and PLAK metamodels. Section 3 presents our technique for CIA. Section 4 illustrates the technique through an extended example. Section 5 discusses related work. Finally, conclusions and further work are presented in Section 6.

2 Background

Variability is spread across several different development artifacts generated during various phases of the SPL development life cycle. Design decisions related to variability may affect other variability decisions. Because the introduction of a change to the requirements could impact variability, it is necessary to understand the relationship between design decisions and variability, and the interdependencies between design decisions related to variability. The variability model on its own, which defines variability independently of those models where variability is realized (e.g. features or software architectures), is not sufficient when information about those software artifacts is required [19]. The specification of PLAs and the documentation of PLAK can provide more complete support for variability to effectively aid CIA in PLAs. Below, we briefly describe the concepts of the Flexible-PLA metamodel [25] for specifying PLAs and the PLAK metamodel for documenting PLAs needed for the purposes of this paper.

2.1 Flexible-PLA Metamodel

The Flexible-PLA metamodel allows us to completely describe the structure of PLAs by specifying not only variations in terms of adding or removing components and connections to/from the PLA (external variation) [9,36], but also in terms of variations inside components (internal variations). Internal variations are specified using *Plastic Partial Components* (PPCs) [25]. The variability mechanism underlying PPCs is based on the principles of Invasive Software Composition [2]. The variability of a PPC is specified using *variability points*, which hook fragments of code to the PPC known as *variants*. These variants, as well as components and PPCs, realize the requirements that have been defined by the domain engineering process at architectural level. Requirements can be related to concerns that crosscut the software architecture (crosscutting concerns)

[1] The PLAK metamodel is the main contribution of a paper currently under review.

or not (non-crosscutting concerns). At architectural level, those variants that realize crosscutting concerns are called *aspects*, and those that realize non-crosscutting concerns are called *features*. Therefore, a PPC is completely defined for a specific product by means of the selection of aspects and/or features through the variability points. Section 3.1 provides an example by defining a model, composed of multiple views, that conforms to the Flexible-PLA metamodel.

2.2 PLAK Metamodel

The PLAK metamodel defines the modeling primitives to capture variability design rationale, as well as the traceability linkage between requirements and PLAs (see Fig. 1). Overall, the aim is to completely document PLAs in a formal way. The PLAK metamodel contains a set of interrelated metaclasses and the services of metaclasses that allow us to manage instances by creating, destroying, adding, or removing elements that are compliant with the constructors of the metamodel. The primitive concepts provided by the metamodel are the following.

- *Closed design decisions* (Closed DDs) support the realization of the common structure of SPLs, so called core assets. They are completely closed (or bound) during the domain engineering process.
- *Open design decisions* (Open DDs) support the realization of the variability of SPLs. They are intentionally left open (or delayed) during the domain engineering process. Open DDs consist of a set of optional design decisions.
- *Optional design decisions* (Optional DDs) support each of the variants of an Open DD. Open DDs are defined during the domain engineering process, and bound to the appropriate Optional DD during the application engineering process.
- *Alternative design decisions* (Alternative DDs) support the alternative realization of Closed and Open DDs, respectively.
- *Constraints*, *assumptions*, *rationale*, *design*, and *patterns* are the major elements of design rationale models [32] and so adopted in our metamodel.
- *Feature concepts* are the concrete representation or expression of requirements in the metamodel. This follows Czarnecki's feature metamodel [8] in which *solitary features* represent optional or mandatory system's characteristics, and *feature groups* consist of a set of alternative system's characteristics —*grouped features.*
- *PLA concepts* are the concrete representation or expression of architectural design in SPLs. This follow the Flexible-PLA metamodel.
- *Linkage rules* comprise the semantics that establish the bridge between features and PLAs. Linkage rules define the logic to create links between specific concepts of the feature metamodel and specific concepts of the Flexible-PLA metamodel.

Product-Line Architectural Knowledge Metamodel (see A, Fig. 1). The metaclass *DesignDecision* offers the primitives to create a design decision. A design decision consists of Alternative DDs specified in the metamodel through

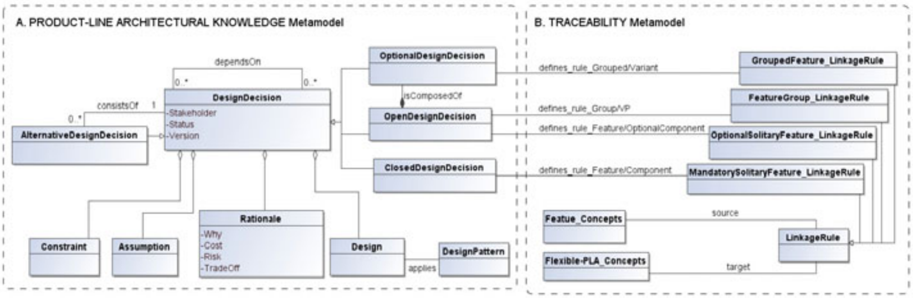


Fig. 1. PLAK Metamodel

the relationship *consistsOf* between the metaclasses *DesignDecision* and *AlternativeDesignDecision*. Open DDs and Closed DDs are specified in the metamodel by means of the metaclasses *OpenDesignDecision* and *ClosedDesignDecision*, respectively. An Open DD is composed of a set of Optional DDs, specified in the metamodel through the aggregation *isComposedOf* between the metaclasses *OpenDesignDecision* and *OptionalDesignDecision*. Since these three metaclasses are describing design decisions, all of them inherit from the *DesignDecision* metaclass. A design decision may have a relation *dependsOn* with other design decisions. Finally, the metaclass *DesignDecision* is composed of the metaclasses *Constraint Assumption*, *Design*, and *Rationale*. The metaclass *Rationale* defines four properties: *why*, *cost*, *risk*, and *tradeoffs* used to justify the design decision. The metaclass *Design* may apply a *Pattern* specified in the metamodel through the relationship *applies*.

Traceability Metamodel (see B, Fig. 1). Design decisions act as traceability links between features and PLA concepts, offering the primitives to define *linkage rules*. The key concepts of the feature metamodel that are involved in the linkage rules are *solitary feature*, *feature group*, and *grouped feature*. The key concepts of the Flexible-PLA metamodel that are involved in the linkage rules are: *component*, *plastic partial component*, *connector*, *(aspect or feature) variability point*, and *(aspect or feature) variant*. The linkage rules are specified in the metamodel through the following metaclasses: *MandatorySolitaryFeature_LinkageRule*, *OptionalSolitaryFeature_LinkageRule*, *FeatureGroup_LinkageRule*, and *GroupedFeature_LinkageRule*. The logics of the possible linkage rules between the feature and Flexible-PLA metamodels are defined in Table 1.

3 CIA in PLAs

From the metamodels that have been introduced in Section 2, we define a CIA technique to assess PLAs. Specifically, we propose to join a *traceability-based algorithm* and a *rule-based inference engine* for analyzing change impact in PLAs from the structural point of view. This involves the traversal of PLAK models

Table 1. Linkage Rules for Design Decisions Traceability Links

Linkage Rule	Description
Mandatory SolitaryFeature	A *SolitaryFeature* whose cardinality attribute is 1..n can trace with a *Component*
	A *SolitaryFeature* whose cardinality attribute is 1..n can trace with a *Connector*
	A *SolitaryFeature* whose cardinality attribute is 1..n can trace with a *PlasticPartialComponent*
Optional SolitaryFeature	A *SolitaryFeature* whose cardinality attribute is 0..n can trace with a *Component* whose cardinality attribute is 0..n
	A *SolitaryFeature* whose cardinality attribute is 0..n can trace with a *Connector* whose cardinality attribute is 0..n
FeatureGroup	A *FeatureGroup* can trace with a *Variability Point* or any of its specializations (*FeatureVP* or *AspectVP*)
GroupedFeature	A *GroupedFeature* can trace with a *Component* whose cardinality attribute is 0..n
	A *GroupedFeature* can trace with a *Connector* whose cardinality attribute is 0..n
	A *GroupedFeature* can trace with a *Variant* or any of its specializations (*Feature* or *Aspect*)

(which encompass all others: feature, PLA, and PLA knowledge models), based on a set of traceability links and propagation rules, to determine he potential impact of implementing a change.

Given a change in requirements, we have defined a *traceability-based algorithm* that allows us to determine (i) the *first-order* design decisions that are involved with the requirement to be changed, (ii) the *n-order* design decisions that depend on the first-order design decisions, and (iii) the *first-order* architectural elements that are involved in each (first and n-order) design decision. The algorithm traverses the traceability links that bridge features and PLA elements, and the dependency relationships between design decisions.

Given a change in the PLA that realizes the change in requirements, we have defined a *rule-based inference engine* that fires propagation rules to obtain the change propagation in the architecture. In other words, when a modification over the PLA is applied, propagation rules are fired to simulate the effects on the rest of the PLA. We thereby obtain the *n-order* architectural elements that are impacted by the change.

The traceability-based algorithm, the rule-based inference engine, and types of changes that they take into account are presented next.

3.1 Change Typology

Williams and Carver [39] have previously classified the change. We have extended these classifications to deal with changes that affect the entire SPL (core components and connectors), specific products (optional components and connectors), and both (plastic partial components with common and variable functionality). Basically, changes in requirements may affect the architectural structure or the architectural behavior. Our solution focuses on the first one. Next, we detail

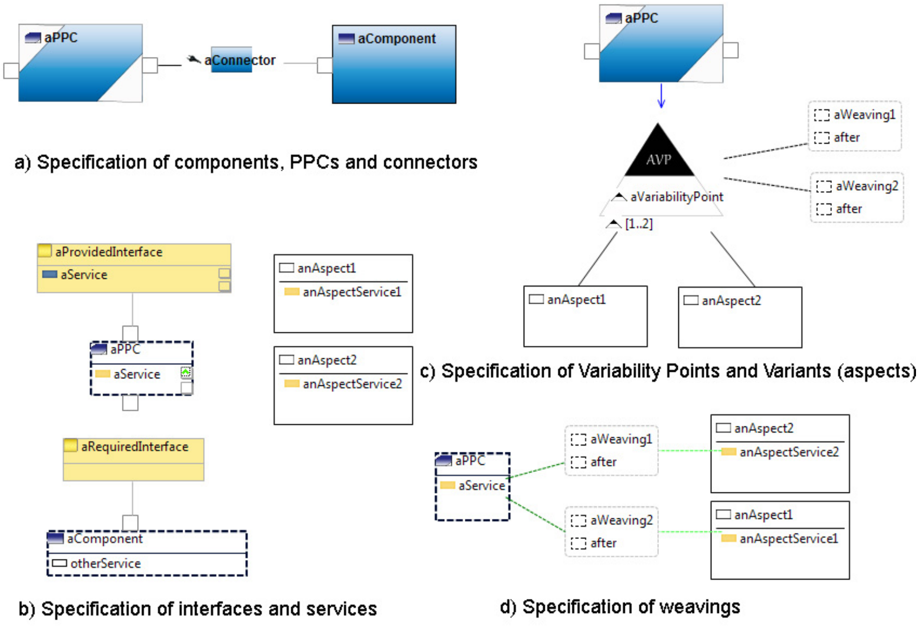


Fig. 2. An Overview of a Flexible-PLA Model using FPLA

the type of changes which we have considered and illustrate them using several snapshots of the tool FPLA[2] (Fig. 2).

- *Functional/non-functional changes*, also known as interface evolution, affect user-observable features. These changes affect the portion of the PLA that is responsible for providing the functional/non-functional feature, and they may not impact the architectural structure. The specific cases which we have considered are additions/deletions of: (i) interfaces and (ii) services to/from components, PPCs, and variants (Fig. 2b).
- *Architectural changes*, also known as structural evolution, affect only the PLA structure. They may be unnoticeable to users and are implemented by means of architecture refactoring and restructuring, enhancing quality attributes. These changes can be classified into (i) *kidnapping*: movement of an entire architectural element from one subsystem to another; (ii) *splitting*: division of the functions of an architectural element into two or more distinct elements; and (iii) *relocating*: movement of functionality from one architectural element to another. They may affect the whole PLA structure, the internal PPC structure, or just some variability points and variants (Fig. 2a, 2b, and 2c).
- *Functional/non-functional and architectural changes* mix the two previous types of changes. The specific cases that have been considered are

[2] https://syst.eui.upm.es/FPLA/home/

additions/deletions of: (i) components, PPCs, and connectors (Fig. 2a) and (ii) variants that implement services that the PPC provides/requires to/from other components or PPCs (Fig. 2c and 2d).

- *Dependency changes* consider change on dependencies between design decisions. It may imply interdependencies between Open DDs (related to variability) and between certain variant selections. There are two forms: *requires* and *excludes.*

3.2 Traceability-Based Algorithm

We have defined a traceability-based algorithm that is able to derive the effects of changes in features. It returns the set of (first and n-order) design decisions, as well as first-order architectural elements that are impacted by the change. Since the change may impact many architectural elements, due to dependencies between design decisions, we need to narrow the traversal of the traceability model (see B in Fig. 1), as well as control the infinite loops. We studied several techniques that address these issues and, finally, chose the one known as *latent semantic analysis* (LSA) [17]. LSA infers the meaning of words from natural text by statistical analysis of the context in which words are used. We apply LSA to the assumptions, constraints, and rationale (see A in Fig. 1) of a design decision and a set of key words related to the requirements to be changed. LSA measures the similarity between design decisions and the requirements to be

Code 1. Traceability-Based Algorithm

```
CONST LSAvalue; //stop condition to narrow the traversal

//Let foAE be first-order architectural elements
[vector] foAE traceabilityBasedAlgorithm (changedFeature, keywords)   {

  //Traversing of the PLAK model to get the DDs which are involved with a changed feature
  [vector] DDs = traversalFromFeaturesToDDs (changedFeature);
  For each designdecision in DDs do {
    //Traversing of the PLAK model to get the AE which are involved with a designdecision
    foAE += traversalFromDDsToAE (designdecision);
    //Traversing of the dependencies of the designdecision with other design decisions
    foAE += processDependencies (designdecision, DDs, keywords);
  }
  return foAE;
}

[vector] foAE processDependencies (designdecision, DDs, keywords) {

  currentLSAvalue = LSA (keywords, designdecision);
  if ((currentLSAvalue>LSAvalue) and (not DDs.contains(designdecision))) {
    DDs.add(designdecision);
    foAE = traversalFromDDsToAE (designdecision);
    [vector] Dependencies = getDependencies (designdecision);
    For each dependency in Dependencies do {
      foAEaux = processDependencies (dependecy, DDs, keywords);
      foAE += foAEaux;
    }
    return foAE;
 }else return [vector] new foAE;
}
```

changed. Code 3.1 describes the pseudo-code algorithm and explains each step through comments.

3.3 Rule-Based Inference Engine

We present the rules to determine the propagation of changes in PLAs. Let: *C* be a component; *PPC* be a plastic partial component; *AVP* be an aspect variability point; *FVP* be a feature variability point; *RI(C)* be the required interface of *C*; *RI(PPC)* be the required interface of *PPC*; *PI(C)* be the provided interface of *C*; *PI(PPC)* be the provided interface of *PPC*; *Hooks*, *Links*, *Weaves*, *Pointcut*, *Advice*, and *Defines* be the primitives to address variability inside PPCs; *UpdateC* be the modification to the cardinality of a variability point; and *N* be a neighbor of a component or a PPC. We define the propagation rules in Table 2. These rules are a set of statements formulated following the pattern *Relation(element, subelement)* in such a way that *element* is related to *subelement*, and the addition/deletion of *Relation* implies the addition/deletion of the subelement from the element. For instance, the deletion *PI(C,s)* consists of deleting a service *s* that is provided by *C* through its provided interface; the addition *Hooks(FVP,feature)* consists of adding the relation hooks from a feature variability point to a feature.

Table 2. Propagation Rules

Type	ID	Rule Description
Intra-Component	R1	deletion RI(C,s) may cause deletion PI(C,s)
	R2	addition PI(C,s) may cause addition RI(C,s)
Intra-PPC	R3	deletion RI(PPC,s) may cause deletion PI(PPC,s)
	R4	addition PI(PPC,s) may cause addition RI(PPC,s)
	R5	addition Hooks(FVP,feature) cause additions Weaves(FVP), Pointcut(PPC,w), Advice(feature,w), UpdateC(FVP) and may cause addition PI(PPC) and/or RI(PPC)
	R6	addition Links(AVP,aspect) cause additions Weaves(AVP), Pointcut(PPC,w), Advice(aspect,w), UpdateC(FVP) and may cause addition PI(PPC) and/or RI(PPC)
	R7	deletion Hooks(FVP,feature) cause deletions Weaves(FVP), Pointcut(PPC,w), Advice(feature,w), UpdateC(FVP) and may cause deletion PI(PPC)
	R8	deletion Links(AVP,aspect) cause deletions Weaves(AVP), Pointcut(PPC,w), Advice(aspect,w), UpdateC(FVP) and may cause deletion PI(PPC)
	R9	deletion FVP cause deletion Defines(PPC,FVP) and Hooks(FVP,features)
	R10	deletion AVP cause deletion Defines(PPC,AVP) and Links(AVP,aspects)
	R11	deletion Defines(PPC,AVP) or Defines(PPC,FVP) may cause deletion PI(PPC)
	R12	addition Defines(PPC,AVP) or Defines(PPC,FVP) may cause addition PI/RI(PPC)
Inter-Component	R13	deletion PI(C) or PI(PPC) cause deletion RI(N)
	R14	addition RI(C) or RI(PPC) cause addition PI(N)

4 Example

This section describes a scenario to illustrate the use of our approach. It exemplifies the SPL for banking systems. Banking systems typically consist of a set of core components that offer their functionality to ATM machines. Among all the functionality, we focus here on that of maintaining an account balance. This functionality must fulfill the non-functional requirement of *availability*. Some products of this SPL require *strict* 24/7 availability, while others permit a weaker, *non-strict* availability. Therefore, the strictness of availability is a variability point.

Various architectural tactics to realize availability have been proposed [31]. We have selected *active redundancy* and *passive redundancy* tactics to implement strict and non-strict availability, respectively. The active redundancy tactic requires a *load balancer* in order for a set of nodes to process identical inputs (request load), and a *synchronizer* in order for the set of nodes to maintain identical state. On the other hand, the passive redundancy tactic requires a *router* in order to provide the active node processes with all the inputs as well as change the route to the passive node when there is a fault, and a *periodic data monitoring* in order to allow the active node and the redundant nodes to maintain periodic state updates.

Fig. 3 shows a small portion of the PLAK model specified for our scenario. The core functionality of banking systems is partially implemented by a component (*ATM*), and a plastic partial component (*balance*), which means it has a part that is common to all products of the SPL and a variable part that is specific to the derived product. PPC *balance* defines two AVPs to implement availability in its two variants: strict and non-strict. Active redundancy is supported by the *Synchronization* and *LoadBalancing* aspects, while passive redundancy is supported by the *Routing* and *DataMonitoring* aspects. The model of Fig. 3 also captures PLA knowledge as follows: a *ClosedDesignDecision* to store the design rationale of realizing the feature *AccountBalance* (see A in Fig. 3); an *OpenDesignDecision* to store the design rationale of realizing the feature *availability* (see B in Fig. 3); links between *Features* and *VariabilityPoints* to store the set of architectural elements that realize the feature *availability* (see C in Fig. 3); *OptionalDesignDecisions* to capture the design options to support the variability of *availability* (see D and E in Fig. 3); links between *Features* and *Variants* to store which set of architectural elements realize *strict availability* or *non-strict availability* (see F and G in Fig. 3); and dependency between two design decisions, shown as the relationship between ClosedDesignDecision_001 and OpenDesignDecision_002 in Fig. 3.

Consider what happens when a customer asks for their account balance to be displayed on the screen of the ATM. In the current formulation of the banking system, if there are too many such requests, the system rejects the request (see the constraint in *ClosedDesignDecision_001*). In this *overload status*, the ATM machine aborts the session with the customer and returns their card. Suppose an engineer is given the task of improving the apparent reliability of the interaction between banking systems and ATM machines when a customer asks for their account balance to be displayed. Instead of simply a rejecting and aborting the customer session, the SPL must be changed in such a way that banking systems respond to an overload status by taking advantage of the *retry* functionality: the system should wait some random amount of time before retrying. If some

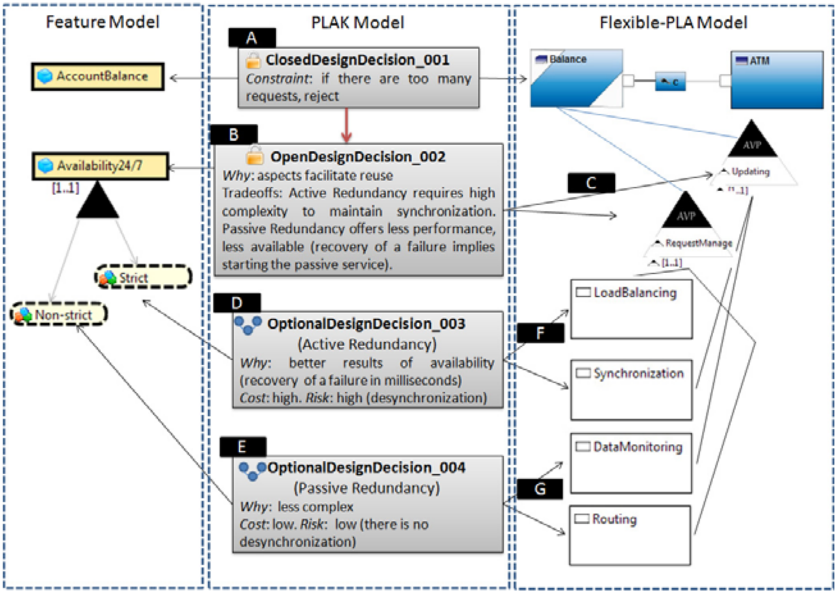

Feature Model
PLAK Model
Flexible-PLA Model
AccountBalance
Availability24/7
[1..1]
Strict
Non-strict
A
ClosedDesignDecision_001
Constraint: if there are too many requests, reject
B
OpenDesignDecision_002
Why: aspects facilitate reuse
Tradeoffs: Active Redundancy requires high complexity to maintain synchronization. Passive Redundancy offers less performance, less available (recovery of a failure implies starting the passive service).
D
OptionalDesignDecision_003
(Active Redundancy)
Why: better results of availability (recovery of a failure in milliseconds)
Cost: high. *Risk*: high (desynchronization)
E
OptionalDesignDecision_004
(Passive Redundancy)
Why: less complex
Cost: low. *Risk*: low (there is no desynchronization)
Balance
c
ATM
AVP
Updating
[1..1]
C
AVP
RequestManage
[1..1]
F
LoadBalancing
Synchronization
G
DataMonitoring
Routing

Fig. 3. Banking Product-Line Architectural Knowledge Model

number of retries are rejected, then the session is aborted. In essence, the protocol between banking systems and ATM machines needs to be modified to decrease the number of aborted sessions.

The engineer has decided to add a new functionality to the feature *AccountBalance* that allows such a retry of a previously rejected request. This retry functionality is optional, that is, it is specific to the bank system to be derived. As a result, the engineer proposes its realization by adding a variability point that stems from the PPC *Balance*. Following the typology described in Section 3.1, this change is a *functional/non-functional and architectural change* because it affects a user-observable feature and the PLA structure.

Given this change, we could start by selecting the features that are affected by the change and trace forward to the design decisions and components that are further impacted. In addition, we could start by selecting the components that must be modified and trace backward through various design decisions and features to validate that the changes will not violate some other design decisions. Here we focus on the first of these actions.

The application of our technique in this scenario consists of first executing the traceability-based algorithm on the PLAK model of Fig. 3, and then the rule-based inference engine. The execution of Code 3.1 returns the set of first-order architectural elements that are impacted by adding the new functionality *retry*, which stems from the feature *AcccountBalance*. These architectural elements are the PPC *Balance* and the AVPs *RequestManager* and *Updating*. The first one directly results through *ClosedDesignDecision_001*, but the second one would not have been detected if we had analyzed the change impact without taking into account PLA knowledge—specifically, the dependencies between design decisions. The dependency between *ClosedDesignDecision_001* and *OpenDesignDecision_002* causes the impact on the AVPs *RequestManager* and *Updating*. This is important because the impacted architectural elements are variability points, and this impact may be higher if their aspects were reused by other PPCs. In fact, in our scenario, these aspects realize in different ways a non-functional requirement, availability, that crosscuts the architecture and involves other design decisions, rationale, and tradeoffs.

Once the algorithm returns the set of impacted elements, the engineer can manually examine them. As *OpenDesignDecision_002* is affected, their options *OptionalDesignDecision_003* (active redundancy) and *OptionalDesignDecision_004* (passive redundancy) are also affected. The engineer is able to infer at this point that the functionality *retry* must be implemented in a different way in the cases of active and passive redundancy. In the first case the retry is automatically directed to another active node, while in the second case the retry requires explicit data synchronization, route change, and retry execution.

The execution of the propagation rules of Table 2 in the PLA returns the following impacts: First, adding the new variability point (FPV) *retry*, stemming from the PPC *Balance*, causes (by rule R12) the addition of a service that provides the functionality *retry* through an interface of the PPC *Balance* to the PPC *ATM*. Second, adding a new variant (feature) *retry* requires adding a relation *Hooks* from the FVP *retry* to the feature *retry*. Third, adding a relation *Hooks* causes (by rule R5) the additions of relations *Weaves*, *Pointcut*, and *Advice*, and the update of *Cardinality*.

In essence, the purpose of automating the traversal of PLAK models is to gather the set of nodes of *potential* interest to the engineer, and the possible additions and deletions that could be automated. That includes detecting features or design decisions that turn out to be in conflict with the proposed change. The engineer would presumably then need to examine the nodes in that set manually to validate the proposed change.

5 Related Work

Many approaches have been proposed to support CIA at the source-code level [14,15], but few have addressed CIA at the architectural level [11,12,40]. It is commonly agreed that traceability is key for identifying artifacts affected by a change [27], from requirements to architecture and vice versa. In this sense, there is a growing body of approaches that address traceability between requirements and architecture [5,23,28], and more specifically variability traceability between requirements and architecture in SPL engineering [21,22,30]. Moon et al. [22] propose a *variability trace metamodel* that connects two metamodels, requirements and architecture, considering variability. Satyananda et al. [30] propose a framework to formally identify traceability between the feature and architecture models using formal concept analysis, functional decomposition, and a set of mapping analysis rules. These approaches show potential and are promising but do not address change impact analysis. Moreover, these approaches only support architectural variability by adding/removing components or connections, or relate to functionality that is provided by component interfaces. Correspondingly, they define traceability links using high-level abstractions of variability, that is, variability of coarse-grained elements. As a result of this, their traceability schema does not describe variability traces at the same detail level than ours.

There are a few approaches that consider architectural design decisions and design rationale to aid change impact analysis [20,29,34]. Moahn and Ramesch [20,21] assert that both documentation of design decisions associated with variations and the capability to trace the life of these variations are key to effectively aiding CIA in SPLs. Their work is very close to ours, but they admit that their approach should be complemented by specialized design-modeling representations to model variation points.

Summing up, the novelty of our proposal relies on coping with all the key elements that can effectively aid CIA in SPLs: what is the knowledge that aids CIA, how this knowledge is modeled, and how this knowledge is utilized. They are addressed in our proposal by: (i) Tracing variability between features and PLAs. Our work differs from previous approaches [21,22,30] in the concept of variability. The variability concept is fine grained, considering architectural variability by adding components and connectors, but also variability inside components by adding variability points, aspects and features. This is why we rely on the Flexible-PLA metamodel to thoroughly specify variability in PLAs, and the PLAK metamodel to define the linkage rules between features and PLAs. (ii) Documenting architectural knowledge. Our work differs from previous approaches [20,29,34] in the architectural knowledge that we have proposed. As a result, conventional models for supporting architectural knowledge have been adapted and extended to capture PLA knowledge. This is why we have defined the PLAK metamodel. (iii) Joining a traceability-based algorithm and a rule-based inference engine to take advantage of the two techniques. Most authors only address one of them: a graph-based analysis (based on links) [34,40] or rule-based systems [11,12,38]. Some others have incorporated additional techniques to improve CIA by using Bayesian networks [35]. Instead of that, we use latent semantic analysis over the documentation that is generated in the software development cycle [10], which is showing promising results in architectural knowledge discovery.

6 Conclusions and Further Work

Ineffective CIA complicates the decision-making process and seriously jeopardizes the success of software evolution. The identification of the architectural elements that are affected by changes in variability is critical to appropriately evolve SPLs. Tracing and documenting the design rationale behind the variability enables more effective CIA. Our approach joins *traceability-based* and *rule-based* techniques for analyzing change impact in PLAs. It traverses and mines the knowledge of two metamodels—Flexible-PLA and PLAK—that together provide the basis for (i) specifying variability in PLAs, (ii) documenting PLA knowledge, and (iii) tracing variability between features and PLAs. Currently, a tool for specifying PLAs is available, as it is shown in Section 3.1, and soon, the view for documenting PLAK will be available as well. At the moment, the inference engine is at the theoretically designed stage but we plan to implement it to validate our approach in a real case study and prove its scalability. In addition, our approach provides the guidelines to be deployed in a model-driven development framework compliant with the MOF four-level architecture. The MOF architecture provides the facilities to construct models using the modeling primitives and guaranteeing model correctness. These models are ready to be involved by transforming their outputs into platform dependent models and/or into automatically generated code. Hence, in future work, software evolution may be (semi-automatically) assisted through model transformations among these models: Flexible-PLA and PLAK models.

Acknowledgment. The work reported here has been partially sponsored by the Spanish MEC (DSDM TIN2008-00889-E), MICINN (INNOSEP TIN2009-13849), and by UPM (Researcher Training program).

References

1. Ali Babar, M., Dingsyr, T., Lago, P., van Vliet, H.: Software Architecture Knowledge Management. Springer, Heidelberg (2009)
2. Assmann, U.: Invasive Software Composition. Springer-Verlag New York, Inc., Secaucus (2003)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edition, 2nd edn. Addison-Wesley Pearson Education, Reading (2003)
4. Bosch, J.: Product-line architectures in industry: a case study. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 544–554. ACM, New York (1999)
5. Chen, C.-Y., Chen, P.-C.: A holistic approach to managing software change impact. J. Syst. Softw. 82(12), 2051–2067 (2009)
6. Cho, H., Gray, J., Cai, Y., Wong, S., Xie, T.: Model-Driven Impact Analysis of Software Product Lines. In: Model-Driven Domain Analysis and Software Development: Architectures and Functions, pp. 275–303. Information Science Reference (2010)
7. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.A.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley Professional, Reading (2010)
8. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. ACM Trans. Softw. Eng. Methodol. 14(2), 199–245 (2005)
10. de Boer, R.C., van Vliet, H.: Architectural knowledge discovery with latent semantic analysis: Constructing a reading guide for software product audits. J. Syst. Softw. 81, 1456–1469 (2008)

11. Feng, T., Maletic, J.I.: Applying dynamic change impact analysis in component-based architecture design. In: SNPD-SAWN 2006: Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp. 43–48. IEEE Computer Society, Washington, DC, USA (2006)
12. Hassan, M.O., Deruelle, L., Basson, H.: A knowledge-based system for change impact analysis on software architecture. In: Proceedings of Fourth International Conference on Research Challenges in Information Science, pp. 545–556 (2010)
13. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 109–120 (2005)
14. Kagdi, H., Hammad, M., Maletic, J.: Who can help me with this source code change? In: IEEE International Conference on Software Maintenance (ICSM 2008), p. 157 (2008)
15. Kim, S., Whitehead, E., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2), 181–196 (2008)
16. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
17. Landauer, T., Foltz, P., Lahan, D.: An introduction to latent semantic analysis (1998), `http://lsa.colorado.edu/papers/dp1.LSAintro.pdf`
18. Lee, M., Offutt, A.J., Alexander, R.T.: Algorithmic analysis of the impacts of changes to object-oriented software. In: TOOLS 2000: Proceedings of the Technology of Object-Oriented Languages and Systems, pp. 61–70. IEEE Computer Society, Washington, DC, USA (2000)
19. Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action. Springer, Heidelberg (2007)
20. Mohan, K., Ramesh, B.: Managing variability with traceability in product and service families. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS, pp. 1309–1317 (2002)
21. Mohan, K., Ramesh, B.: Tracing variations in software product families. Commun. ACM 50, 68–73 (2007)
22. Moon, M., Chae, H.S., Nam, T., Yeom, K.: A metamodeling approach to tracing variability between requirements and architecture in software product lines. In: CIT 2007: Proceedings of the 7th IEEE International Conference on Computer and Information Technology, pp. 927–933. IEEE Computer Society, Washington, DC, USA (2007)
23. Olsen, G., Oldevik, J.: Scenarios of traceability in model to text transformations. In: Akehurst, D., Vogel, R., Paige, R. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 144–156. Springer, Heidelberg (2007)
24. Pérez, J., Díaz, J., Garbajosa, J., Alarcón, P.P.: Flexible *working architectures*: Agile architecting using ppcs. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 102–117. Springer, Heidelberg (2010)
25. Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture, WICSA/ECSA, pp. 221–230. IEEE Computer Society Press, Los Alamitos (2009)
26. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)

27. Pohl, K., Brandenburg, M., Glich, A.: Integrating requirement and architecture information: A scenario and meta-model approach. In: REFSQ 2001: Proceedings of The Seventh International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 68–84 (2001)
28. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Trans. Softw. Eng. 27(1), 58–93 (2001)
29. Riebisch, M., Wohlfarth, S.: Introducing impact analysis for architectural decisions. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 381–392. IEEE Computer Society, Washington, DC, USA (2007)
30. Satyananda, T.K., Lee, D., Kang, S., Hashmi, S.I.: Identifying traceability between feature model and software architecture in software product line using formal concept analysis. In: ICCSA 2007: Proceedings of the The 2007 International Conference Computational Science and its Applications, pp. 380–388. IEEE Computer Society, Washington, DC, USA (2007)
31. Scott, J., Kazman, R.: Realizing and refining architectural tactics: Availability. Technical report, CMU/SEI-2009-TR-006 ESC-TR-2009-006 (2009)
32. M., Shahin, Liang, P., Khayyambashi, M.: Architectural design decision: Existing models and tools. In: Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture, WICSA/ECSA 2009, pp. 293–296 (2009)
33. Tang, A., Babar, M.A., Gorton, I., Han, J.: A survey of architecture design rationale. J. Syst. Softw. 79, 1792–1804 (2006)
34. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. J. Syst. Softw. 80, 918–934 (2007)
35. Tang, A., Nicholson, A., Jin, Y., Han, J.: Using bayesian belief networks for change impact analysis in architecture design. J. Syst. Softw. 80, 127–148 (2007)
36. van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N.: Taming architectural evolution. In: Proceedings of the ESEC/FSE-9, pp. 1–10. ACM, New York (2001)
37. van Vliet, H.: Software architecture knowledge management. In: 19th Australian Conference on Software Engineering, 2008, ASWEC 2008, pp. 24–31 (2008)
38. Vora, U.: Change impact analysis and software evolution specification for continually evolving systems. In: Proceedings of Fifth International Conference on Software Engineering Advances (ICSEA), pp. 238–243 (2010)
39. Williams, B.J., Carver, J.C.: Characterizing software architecture changes: A systematic review. Inf. Softw. Technol. 52(1), 31–51 (2010)
40. Zhao, J., Yang, H., Xiang, L., Xu, B.: Change impact analysis to support architectural evolution. Journal of Software Maintenance 14, 317–333 (2002)

Extending UML Components to Develop Software Product-Line Architectures: Lessons Learned

Antonio C. Contieri Junior[1], Guilherme G. Correia[1], Thelma E. Colanzi[1],
Itana M.S. Gimenes[1], Edson A. Oliveira Junior[1], Sandra Ferrari[1],
Paulo C. Masiero[2], and Alessandro F. Garcia[3]

[1] State University of Maringá, Maringá-PR, Brazil
{contierijr,guilherme.tusso}@gmail.com,
{thelma,itana,sandra}@din.uem.br, edson@edsonjr.pro.br
[2] University of São Paulo, São Carlos-SP, Brazil
masiero@icmc.usp.br
[3] Pontificial Catholic University of Rio de Janeiro, Rio de Janeiro-RJ, Brazil
afgarcia@inf.puc-rio.br

Abstract. This paper presents an experience in extending and evaluating UML Components for guiding the derivation of component-based product-line architectures (PLAs). We present a quantitative and qualitative evaluation of such an extension when applied to the proactive design of two PLAs. We have found that our approach supported an agile conception of architectural designs that are modular and likely to be resilient to changes over time and good enough to serve as the basis for more specific design decisions made by the architects.

Keywords: Software Product-Line Architectures, Component-based Development Methods, Variability Management.

1 Introduction

Several adopters of component-based software engineering are increasingly embracing software product lines (PL) [3]. Their goal is shifting from the reuse of individual components to the large-scale reuse of a product-line architecture (PLA) for a specific domain [1]. The proactive design of a PLA should encompass the components realizing all the mandatory and varying features in a domain [6]. A key condition for the successful early design of a PLA is the identification of components and interfaces that modularize each individual PL feature. Otherwise, the PLA is likely to be the target of early reviews and major refactorings since the design outset, thereby delaying the development process. A non-modular PLA will also suffer more changes when new features need to be accommodated, making it difficult to sustain its design stability over time.

Unlike product-line development methods (e.g. [3,13]), UML Components [4] is a consolidated and genuine component-based development method, which defines practical, simple and concise directives for architectural design. It allows

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 130–138, 2011.

components, interfaces and their relationships to be identified and specified directly from the system requirements. Therefore, UML Components could be extended to also support organizations in designing component-based PLAs. When a proactive approach is adopted, a PLA needs to realize up front all the product variations that are likely to be designed and implemented on the foreseeable horizon [6]. The major decisions of the PLA design are made in initial stages of the development to avoid design instabilities in later stages. UML Components was not originally conceived to support the development of product lines. In fact, there is a lack of more general approaches which take advantages of standard notations, such as UML, and its profiling extension mechanism for a wide range of PLA decomposition styles.

This paper presents an experience in extending and evaluating UML Components for guiding the development of component-based PLAs. In particular, we extend UML Components with an existing technique [7] to support variability identification and representation in architectural design. This extension, called SMarty Components, enables PLA developers to use well-known and consolidated architectural design methods without significantly changing its original component-based design practices. Our goal is to assess to what extent SMarty Components would be effective on supporting agile architecture creation of a PL. We consider that an architectural design is effective when it is considered modular and stable according to a number of well-known software metrics and when it entails an initial design that, according to architects, is good enough to serve as the basis for more specific architectural decisions.

There are several comprehensive methods to develop PL using a component-based approach such as Kobra [3] but in this research we are interested in methods that are easy to use, rely on the UML notation and allows an agile, early definition of a PLA that can be used to assess properties of the architectural design and be later refined. In this paper we present main concepts of an approach of variability management (Section 2); the application of SMarty Components to develop PLs (Section 3); an analysis of the conceived PLAs (Section 4), and conclusions (Section 5).

2 Stereotype-Based Management of Variability (SMarty)

The PL approach is focused on the design and implementation of a PLA, a common architectural design for its products. The degree of variability of a PL is directly related to the abstraction capacity of the PLA and the number of products that can be conceived from it [1]. Variability management [7] encompasses activities for identifying, representing, resolving, analyzing and controlling PLA variabilities. There are several approaches for realizing such activities and support an accurate PL domain analysis [1] [13] [9]. They are limited to represent variability in a specific language or notation, such as Architecture Description Languages (ADLs) that are non-standard languages.

Stereotype-based Management of Variability (SMarty) [7] is composed of a UML profile, named SMartyProfile, and a process for managing variability, named SMartyProcess, to provide an alternative to variability identification and representation by means of a well-defined process and a UML profile. SMarty was proposed to enable generic software development approaches to be applied in the context of UML-based PL engineering. It does not impose an entirely-new method to component-based PL development, as it can be easily coupled to component-based design practices. SMarty takes as inputs the PL development outputs to realize the SMartyProcess activities. Based on such inputs, SMartyProcess produces the PL variability models.

SMartyProfile contains a set of stereotypes and meta-attributes for representing variability in UML models. Such stereotypes are based on the concepts of: variability, variation point, variant, and variant constraints. The SMartyProcess supports the identification, delimitation, representation, and tracking of the variabilities. Its realization is iterative and incremental, and occurs in parallel with the product-line development process, by following specific guidelines. Product-line activities progressively feed the SMartyProcess with inputs to realize its respective activities. Thus, the variability degree tends to increase as the SMartyProcess activities are carried out [7]. The activities of SMartyProcess are not related to a specific development approach for PLs. Therefore, it can be incorporated in any approach for UML-based PL design.

3 Applying SMarty Components

The activities of UML Components produces a Use Case and a Business Type Model. According to the SMartyProcess, both models might be used to produce variability models for a PL by following guidelines which suggest how to apply the SMartyProfile to manage variabilities. Therefore, the extension of UML Components by means of SMarty is represented as follows [14]: (i) the Business Type Model from UML Components is the input to the SMartyProcess to define the Business Type Variability Model; and (ii) the Component Architecture from UML Components is the input to define the Variability Implementation Model.

SMarty Components was applied to develop the proactive design of two PLs: AGM (Arcade Game Maker) [10] and Mobile Media [12]. Due to space limitation, results of the application of SMarty Components to the Mobile Media were not included in this paper, but they can be found in [14]. AGM encompasses three arcade games: Brickles, Bowling, and Pong. Its main variations are: (i) rules of the games; (ii) kind, number and behavior of elements; and (iii) physical environment, where the games take place. There are some common rules, such as: (i) each game has a set of Sprites; (ii) each game has a set of rules; and (iii) every game involves movement [7]. There are four variabilities in Figure 1: `game`, `sprite`, `movable sprite`, and `stationary sprite`, represented by UML notes. They are respectively related to the following business types: Game, Sprite, MovableSprite and StationarySprite. Based on the Business Type Variability Model for AGM, it was conceived a component-based PLA (Figure 2). It has two system components: `GameCtrl` and `GameBoardCtrl`. They use the provided interfaces for business components that run the AGM games.

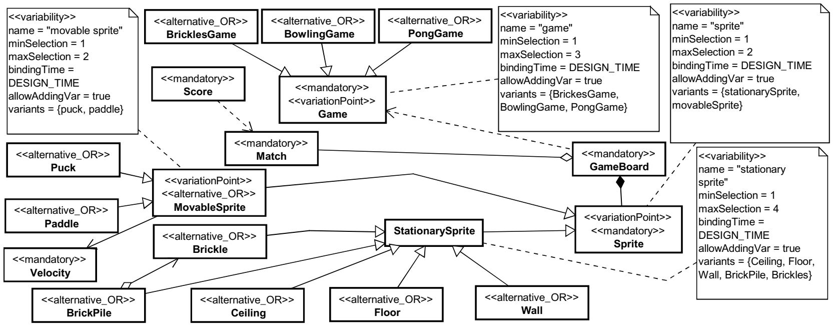

<<variability>>
name = "movable sprite"
minSelection = 1
maxSelection = 2
bindingTime =
DESIGN_TIME
allowAddingVar = true
variants = {puck, paddle}
<<alternative_OR>> BricklesGame
<<alternative_OR>> BowlingGame
<<alternative_OR>> PongGame
<<variability>>
name = "game"
minSelection = 1
maxSelection = 3
bindingTime = DESIGN_TIME
allowAddingVar = true
variants = {BrickesGame, BowlingGame, PongGame}
<<variability>>
name = "sprite"
minSelection = 1
maxSelection = 2
bindingTime = DESIGN_TIME
allowAddingVar = true
variants = {stationarySprite, movableSprite}
<<mandatory>> Score
<<mandatory>> <<variationPoint>> Game
<<mandatory>> Match
<<mandatory>> GameBoard
<<alternative_OR>> Puck
<<variationPoint>> <<alternative_OR>> MovableSprite
<<alternative_OR>> Paddle
<<variationPoint>> <<mandatory>> Sprite
<<alternative_OR>> Brickle
StationarySprite
<<mandatory>> Velocity
<<variability>>
name = "stationary sprite"
minSelection = 1
maxSelection = 4
bindingTime =
DESIGN_TIME
allowAddingVar = true
variants = {Ceiling, Floor, Wall, BrickPile, Brickles}
<<alternative_OR>> BrickPile
<<alternative_OR>> Ceiling
<<alternative_OR>> Floor
<<alternative_OR>> Wall

Fig. 1. AGM Business Type Variability Model according to SMarty Components

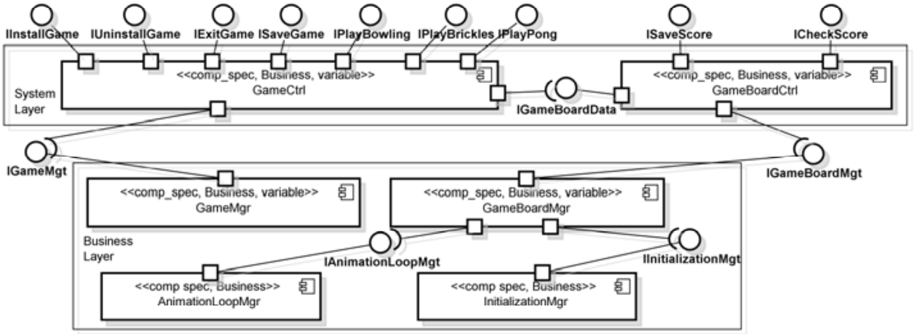

IInstallGame
IUninstallGame
IExitGame
ISaveGame
IPlayBowling
IPlayBrickles
IPlayPong
ISaveScore
ICheckScore
System Layer
<<comp_spec, Business, variable>> GameCtrl
IGameBoardData
<<comp_spec, Business, variable>> GameBoardCtrl
IGameMgt
IGameBoardMgt
<<comp_spec, Business, variable>> GameMgr
<<comp_spec, Business, variable>> GameBoardMgr
Business Layer
IAnimationLoopMgt
IInitializationMgt
<<comp spec, Business>> AnimationLoopMgr
<<comp spec, Business>> InitializationMgr

Fig. 2. The AGM Component-based Architecture

The business types `Game`, `BricklesGame`, `PongGame` and `BowlingGame` belong to the component `GameMgr`. All of the other business types belong to `GameBoardMgr`.

In Figure 2 we use the stereotypes ≪comp spec≫ from UML Components to indicate an architectural component and ≪variable≫ from SMarty to indicate a variability. Additionally, some components are also stereotyped with the name of the concerns they encapsulate (e.g. Exception Handling and Persistence). These stereotypes support the quantitative analysis process. Concerns related to operations were supressed. Components with suffix Ctrl indicate managers that use the services available in the business layer. These components provide operations to the interface and dialog layers. Business components have the Mgr suffix. These components represent application independent elements to be developed. They can be further reused in several systems.

4 Analysis of the AGM Archictectures

To evaluate the effectiveness of the SMarty Components, we decided to analyze its application in the context of PLAs following different architectural

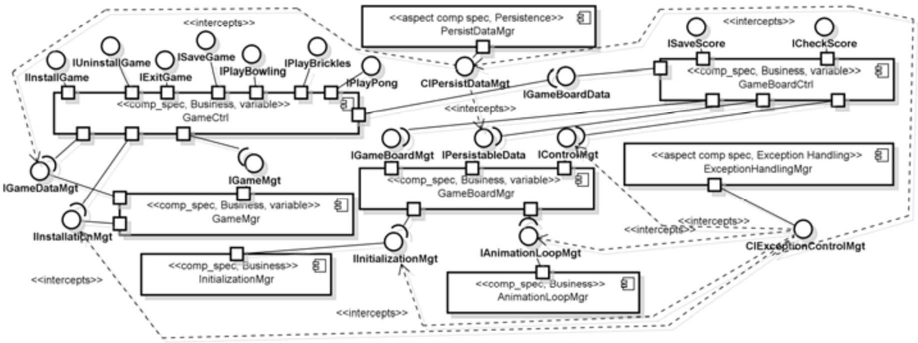


Fig. 3. The Component-based AO Architecture of AGM

styles [2]. To accomplish this, an alternative aspect-oriented (AO) architecture was also designed for the AGM. To support the design of AO elements, a method called DSBC/A, was used [8]. DSBC/A is an extension of UML Components that allows designing component-based AO architectures. This method follows the UML Components process with additional activities to modularize crosscutting features. The use of DSBC/A guides the elaboration of functional and non-functional models and the interaction between them; the goal is to identify transversal and aspectual elements in the architectural design. Therefore, it enables to identify crosscutting interfaces for the aspectual components in charge of modularizing crosscutting features. SMarty Components must be applied to such interfaces in order to produce corresponding variability implementation models.

Figure 3 presents the component-based AO architecture for AGM. The major differences between the two alternative architectures for AGM are in the inclusion of aspectual components represented by the stereotype ≪aspect comp spec≫. These components encapsulate operations that crosscut the whole system to improve modularization. For instance, the component `PersistDataMgr` encompasses the persistence operations which are required in several parts of the architecture. The component `ExceptionHandlingMgr` is responsible for operations related to error handling and recovery and came from the crosscutting use case of AGM. These components were scattered in several parts of the product-line design and, according to DSBC/A, they are candidates to be implemented as aspects.

An additional issue to observe in this architecture is the separation of operations into more than one interface of business components. This was done to make them clear. The separation allows the isolation of operations to be crosscut by ordinary ones. An example is the interface `IInstallationMgt` in the component `GameMgr` that is crosscut by the component `ExceptionHandlingMgr`.

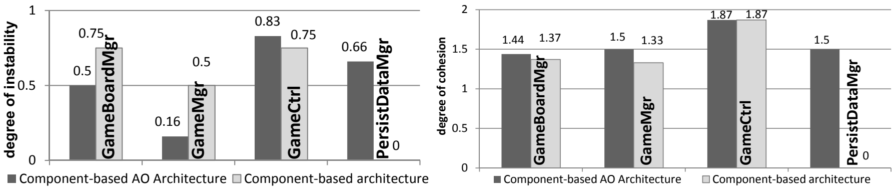
4.1 Quantitative Analysis

The objective of the metrics collected, as shown in Table 1, was to analyze if the conceived architectures are likely to entail proper designs from the point of view of modularity and stability. The instability measure (I) [11] takes into account the coupling represented in the relationships between components with the objective of predicting the difficulty of modifying a component. Tightly coupled components are unstable and difficult to be reused. The values of I for the AGM PLAs can be seen in Figure 4a. Some components were not shown in the graphics due to lack of space although they have similar values. Overall, both PLAs present medium values for I. This indicates that the components are not very strict and are little stable. Thus they admit modifications but control must be in place to reduce impacts and dissemination of changes for the whole architecture.

Table 1. Metrics Used

Metric		Definition
Stability metrics	Afferent Coupling (Ca)	The number of classes outside a component that depend upon classes within this component.
	Efferent Coupling (Ce)	The number of classes inside a component that depend upon classes outside this component.
	Instability (I)	Defined as I = Ce / (Ca + Ce). This metric has the range [0,1]. I=0 indicates a maximally stable component. I=1 indicates a maximally unstable component.
	Cohesion (H)	It counts the average number of relationships between classes and interfaces inside a component.
Modularity Metrics	Concern Diffusion over Architectural Components (CDAC)	It counts the number of architectural components which contributes to the realization of a certain concern.
	Lack of Concern-based Cohesion (LCC)	It counts the number of concerns addressed by the assessed component.

The cohesion measure (H) [5] supports designers to identify components which have not strongly-related elements. This means that component cohesion is weak, thus requiring redesign to improve the architecture. The H values for AGM's PLAs have indicated that the internal elements of components present a reasonable number of relationships (Figure 4b), so these are cohesive.



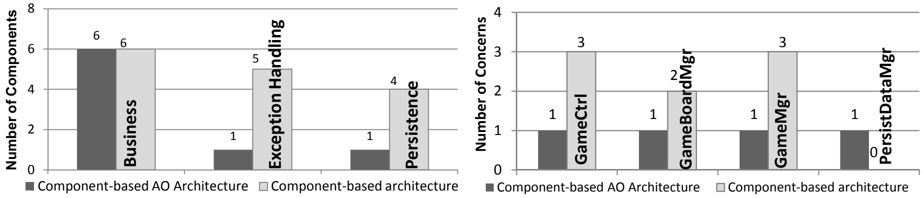
(a) Instability **(b)** Cohesion

Fig. 4. Stability values for AGM architectures

In addition to these measures, Lack of Concern-Based Cohesion (LCC) and Concern-Diffusion over Architectural Components (CDAC) metrics [12] have also been collected. They evaluate the degree of modularization of an architecture in terms of the architectural concerns being realized. They are relevant in the PL design because if features are not neatly encapsulated in specific components, the PLA may negatively impact reusability and maintainability of the PL.

LCC [12] indicates that a component that addresses many features is not stable as a modification in any of the associated features may impact the others.

The LCC values for the AGM's PLAs show that the component-based PLAs present a higher number of features implemented by each component than the component-based AO PLAs (Figure 5a). As expected, the AO PLA contains only one feature per component. This is due to the possibility of encapsulating crosscutting feature into aspectual components. CDAC [12] considers that a feature scattered through a high number of elements has negative impact on modularity. The component-based AO architecture presents low values to exception handling and persistence whereas component-based PLA presents higher values (Figure 5b). This reflects the scattering of those crosscutting features across several components. These values are higher due to the same reasons that justify the difference between the LCC values.



(a) LCC **(b)** CDAC

Fig. 5. Modularity values for AGM architectures

From the point of view of stability and modularity, SMarty Components led to effective architectural solutions for AGM. Although the PLAs present differences in respect to feature modularization, the values of CDAC and LCC are acceptable. These are good indicators that SMarty Components can lead to the design of modular architectures. In addition, by using a PL variability management approach, it is possible to apply measures that reveal the differences between PLAs versions. Thus, the best architecture can be chosen according to the business drivers.

4.2 Qualitative Analysis

The goal of the qualitative evaluation was to gather additional insights that complement the quantitative evaluation and enable us to better understand the efficacy of using SMarty Components. Three specialists and architects who are aware of the AGM requirements analyzed both AGM´s PLA. They also have long-term experience on component-based and aspect-based architectural designs. The architectural designs created have been considered by the specialists as good first-cut architectures. They all agreed that the conceived component-based and aspect-oriented decompositions were useful to enable them to concentrate in minor design enhancements. Therefore, the suggested improvements concentrated on minor refactorings of the produced components and interfaces. First, certain domain-specific variabilities could be factored out as aspects. Second, the component in charge of error handling in the component-based AO architecture could be further decomposed into more specific components. The goal is to avoid that information specific to each AGM layer is not propagated to upper layers. This suggested that refactoring does not apply to the component-based decomposition as it cannot segregate the crosscutting behavior of error handling.

A final observation was that the `GameCtrl` component in both component-based and component-based AO architectures was considered to accumulate too many responsibilities. Therefore, this component was a candidate to manifest an architectural design anomaly. In fact, this is confirmed by metrics for this component (e.g. Figure 4). Coarse-grained components can make it difficult to sustain the stability of PLAs if new variabilities need to be included.

5 Concluding Remarks

The experiment has shown that SMarty Components enables earlier identification of components, interfaces and their relationship in the same way UML Components does. Thus, major decisions of the design can be made in the initial stages of the development. This supports both the PL development complexity management and the identification of product alternatives that can be produced from the designed PLA. SMarty Components was effective to design modular and stable PLAs regardless of the architectural style employed (either component-based or component-based AO architectures). Improvements were recommended by the architects of AGM and Mobile Media; they consist of either minor refactorings or minor design decisions that would be revealed in later design stages.

The PLAs designed by using SMarty Components represent effective solutions to the target PL domains. In particular, we observed, through the application of measures, that the PL features showed good modularization. The measurement results also point out that such modular PLAs were likely to remain stable over time.

References

1. van der Linden, F.: Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)
2. Clements, P., et al.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2010)
3. Atkinson, C., et al.: Component-based Product Line Engineering with UML. Component Series. Addison-Wesley, Reading (2002)
4. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Boston (2001)
5. Martin, R.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2003)
6. Krueger, C.: Eliminating the adoption barrier. IEEE Software - Special issue on Initiating Software Product Lines 19(4), 28–31 (2002)
7. Junior, O., et al.: Systematic Management of Variability in UML-based Software Product Lines. Journal of Universal Computer Science 16(17), 2374–2393 (2010)

8. Eler, M.M., Masiero, P.C.: Aspects as components. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 411–414. Springer, Heidelberg (2006)
9. Korherr, B., List, B.: A UML 2 Profile for Variability Models and their Dependency to Business Processes. In: International Conference on Database and Expert Systems Applications, pp. 829–834. IEEE Computer Society, Washington, DC, USA (2007)
10. SEI - Arcade Game Maker Product Line, http://www.sei.cmu.edu/productlines/ppl
11. Martin, R.: Stability. C++ Report (1997)
12. Sant'Anna, C.N.: On the Modularity of Aspect-Oriented Design: A Concern-Driven Measurement Approach. Doctoral Thesis. PUC-Rio, Brazil (2008)
13. Gomaa, H.: Designing Software Product Lines with UML: from Use Cases to Pattern-based Software Architectures. Addison-Wesley, Reading (2005)
14. Mobile Media Web Site, http://www.din.uem.br/~teclopes/MMedia

PL-AspectualACME: An Aspect-Oriented Architectural Description Language for Software Product Lines

Eiji Adachi Barbosa[1], Thais Batista[2], Alessandro Garcia[1], and Eduardo Silva[2]

[1] OPUS Research Group, Informatics Department, PUC-Rio, Rio de Janeiro, Brazil
[2] Informatics Department, UFRN, Natal, Brazil
{ebarbosa,afgarcia}@inf.puc-rio.br, thais@ufrnet.br,
duh.ciencomp@gmail.com

Abstract. Software Product Line (SPL) development typically relies on feature models to represent the commonalities and variabilities of a family of software products. Although feature models play an important role in describing SPL elements, they are limited to provide high-level feature decompositions that do not explicitly represent the SPL architecture. To tackle this problem, we present PL-AspectualACME, an extension of the ACME architecture description language that enriches existing abstractions to express architectural variabilities. They support the specification of product variations without forcing architects to learn many new abstractions. We evaluate the applicability of our proposal in the context of a real large-scale system, the Ginga SPL architecture.

Keywords: Software Product Line, ADL, PL-AspectualACME.

1 Introduction

Software product line (SPL) development [1] has become a mainstream technique to the development of systems (or *products*) with shared features (*commonalities*) and *variabilities* that distinguish them. The SPL development process typically uses feature models [2] to represent commonalities, variabilities, and variation-related constraints governing the set of features and their relationships. They are used in the domain analysis and in product derivation of a SPL. While feature models are also intended to structure the SPL features in a hierarchical fashion, they are disconnected from component-based decompositions that are commonly used to structure and implement SPL architectures.

In fact, SPL architects usually employ a component-based design approach in order to promote a smooth mapping of their decisions to elements of the implementation platform. These decisions encompass, for instance, the choice of mandatory or variable architectural elements as well as the architectural styles being adopted. In this context, this paper exploits a well-known architectural way of organizing components in a product family, the *architectural styles* [3]; they define architectural elements for domain-specific concepts that are further used to describe SPL architectures. The potential of architectural styles for broader applicability are emphasized in [4].

Architectural description languages (ADLs) typically use architectural styles to define vocabularies of types of components, connectors, properties, and sets of rules that specify

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 139–146, 2011.

how elements of those types may be legally composed in a reusable architectural domain. In this paper, we ground this idea in PL-AspectualACME, a seamless extension of a general-purpose aspect-oriented ADL, AspectualACME [5]. AspectualACME is an aspect-oriented (AO) extension of ACME [6], a general-purpose ADL proposed as an architectural interchange language. PL-AspectualACME promotes a natural blending of SPL and aspect-oriented architectural abstractions. The philosophy of PL-AspectualACME is to take advantage of ACME elements to model SPL architectures. Thus, it relies on existing ACME abstractions to represent the variabilities of a SPL.

This paper is structured as follows. Section 2 presents the background of this work: the Ginga middleware, the running example that we use along this paper (section 2.1), and ACME and AspectualACME basic elements (section 2.2). In Section 3 we present PL-AspectualACME. Finally, Section 4 presents the final remarks.

2 Background

2.1 Running Example: Ginga

Ginga, the Brazilian Terrestrial Digital TV System (SBTVD) middleware [7], was refactored in a previous work [8] in order to provide a version built based on the SPL approach. The goal was to increase the *Ginga* middleware configurability through the automatic management of variabilities. Figure 1 illustrates a partial view of the Gingafeature model. The *Tuner* feature is responsible for selecting the physical channel of signal transmission. The *Media Processing* feature manages the processing of multimedia data and makes them available to other components of the middleware. *Data Processing* is the feature responsible for accessing, processing and providing elementary data streams to other middleware components.

Although a SPL provides mechanisms to manage the variability of an application domain, it is more difficult to support variabilities in the presence of crosscutting concerns [9, 10]. Thus, AO development was applied at the SPL version of the Ginga middleware to separate and compose crosscutting concerns in terms of features, allowing to (un)plug these features of the SPL architecture, and providing the architecture the ability for better modularization and adaptability.

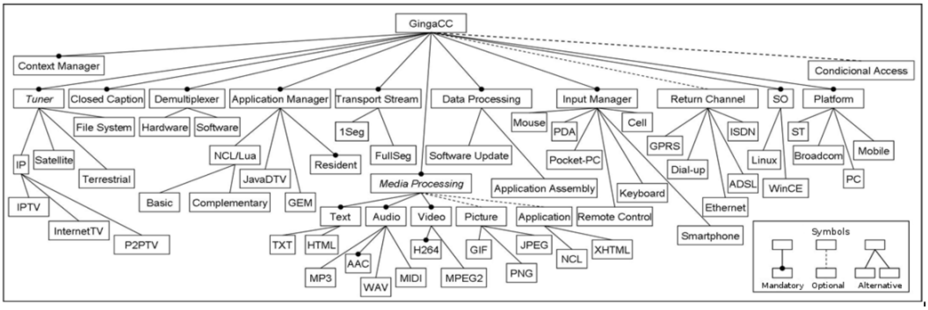


Fig. 1. Ginga Partial Feature Model

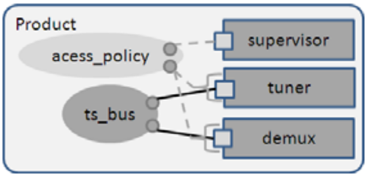
2.2 ACME and AspectualACME

The basic elements of ACME are not enough to properly modularize and compose crosscutting concerns (or features) with other system concerns. Crosscutting concerns [11] are concerns that get scattered and tangled with other concerns realized by the system components. AspectualACME [5] extends ACME in order to modularly represent crosscutting concerns at the architectural level. It proposes modeling crosscutting concerns as regular components, and enriching composition mechanisms of ACME to support the definition of crosscutting relations.

AspectualACME introduces 2 main extensions: (i) the *Aspectual Connector (AC)*, a special connector that encapsulates the crosscutting component interactions; and (ii) a *quantification mechanism* (the wildcard *), used within the Attachments section to syntactically simplify the reference to a well-defined *join points* in the architecture. Along with AC, two new interfaces – the *Base Role* and the *Crosscutting Role* – are also defined in order to distinguish the roles of the elements that participate in a crosscutting interaction. The *Base Role* is associated with the component affected by the crosscutting interaction; the *Crosscutting Role* is associated with the component that affects other components and, therefore, acts as an "aspectual component". AC has a *Glue Clause* that defines when (after, before, or around) those elements interact.

```
1.System Product:Ginga=new Ginga extended with {
2. ...
3. Component tuner: TunerT = new TunerT;
4. Component demuxer: DemuxT = new DemuxT;
5.
6. Connector ts_bus: InnerBusT = new InnerBusT;
7.
8. Component supervisor: AccessControlT = new
AccessControlT;
9.
10. AspectualConnector access_policy =
11. {BaseRole controllee1, controllee2;
12.  CrosscuttingRole controller;
13.  Glue controller around controllee;}
14.
15. Attachments {
16.   ts_bus.consumer to demuxer.ts_filter;
17.   ts_bus.provider to tuner.ts_deliver;
18.
19. access_policy.controllee1 to
        demuxer.ts_filter;
20. access_policy.controllee2 to
        tuner.ts_deliver;
21. access_policy.controller to
        supervisor.check_permission; }
22. ... }
```

(a)



(b)

Fig. 2. (a) AspectualACME textual description. (b) AspectualACME graphical representation

In Figure 2(a), the *tuner* component (line 3) abstracts the mechanism of selection and recovery of the data flow transmitted by different data providers. The *demuxer* component (line 4) abstracts the demultiplexer mechanism that transforms the data stream received from the *tunner* component into more elementary data flows. The *supervisor* component (line 8) abstracts the mechanism that controls the access to restricted content transmitted to receivers. The *access_policy* aspectual connector (lines 10-13) abstracts the composition mechanism by which the *supervisor* component controls the content transmitted between the *tuner* and *demuxer* components. In the attachment section, the *controllee1* and *controllee2* base roles are respectively attached to the *ts_filter* port of the *demuxer* component and the *ts_deliver* port of the *tuner* component (lines 16-17). Next, the *controller* crosscutting role is attached to the *check_permission* port of the *supervisor*. The aspectual connector intercepts any communication between *tuner* and *demuxer* components and deviates the data transmitted to *supervisor*; the latter checks access credentials and retransmits the receiver the data intercepted. Figure 2(b) graphically represents the example.

3 PL-AspectualACME

In software development is common to some requirements, when implemented, lead to crosscutting concerns. In SPL development, the crosscutting concerns are critical, since they usually have variation points and, thus, affect many different products. Aspect-oriented programming has been frequently used to modularly implement some kinds of variation points in SPL development, allowing developers to maintain more easily a product configuration. Due to the inherent complexity when dealing with a SPL development, especially when crosscutting concerns have variation points, the sooner those crosscutting concerns can be identified the better, because they can be treated properly in early stages of the development process, instead of demanding changes in latter stages, such as implementation. For that reason, PL-AspectualACME extends the aspect-oriented ADL AspectualACME in order to represent SPL variability in aspect-oriented architectures.

PL-AspectualACME does not add new abstractions to the language; it semantically enriches those that already exist. The core architecture of a SPL is described in PL-AspectualACME in terms of a vocabulary of types of components, connectors and ports within an ACME *Family*. Variabilities are modeled using the *Representation* construct, which was originally proposed in ACME as a mechanism for more detailed description of architectural elements. The Representation construct is typically used for a more detailed hierarchical decomposition of components, ports or connectors into subsystems. However, it is used in PL-AspectualACME also to modularize architectural variabilities that are related to specific product variations, allowing software architects to model SPL variabilities without being burdened with many new abstractions. Thus, in PL-AspectualACME the commonalities of a product line are modeled as regular elements and each *Representation* of an element identifies the possibility of realizing a specific variability. The complete definition of an ACME *Family* that represents the architecture of a SPL is basically divided into two parts: (i) the definition of a vocabulary of architectural element types, which the architect later uses to describe the SPL architecture; and (ii) the definition of the core SPL architecture, i.e., the default architecture shared by all products of a SPL.

3.1 Specifying the SPL Architectural Vocabulary

ACME provides the *Family* construct to support the definition of architectural styles. Architects can describe their architectural styles in ACME in terms of a vocabulary of architectural element types, which are defined using the *Type* construct.

PL-AspectualACME exploits these mechanisms in order to define a vocabulary of architectural element types that are closer to the SPL domain.

```
1.Family Ginga = {
2.   Component Type TunerT = {
3.     Port listen; }
4.   Component Type DataProviderT = {
5.     Port provide_data; }
6.   Component Type IP extends DataProviderT
       with = {
7.       Property variants = {IPTV, P2PTV,
                                 InternetTV}
8.       Representation IPTV = {
9.         System IPTV = {...} }
10.      Representation P2PTV = {
11.        System P2PTV = {...} }
12.      Representation InternetTV = {
13.        System InternetTV = {...} }
14.   }
15. Connector Type INetworkT = {
16.     Role provider;
17.     Role consumer;
18.     Representation Wired = {
19.        System Wired = {...} }
20.     Representation Wireless = {
21.        System Wireless = {...}
22. } } }
```

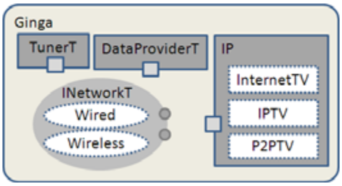


Fig. 3. (a) The SPL Architectural Vocabulary. (b) Ginga vocabulary graphical representation

In Figure 3(a) specifies the Ginga Family. The *TunerT* component type (lines 2-3) models the mechanism of selection and recovery of the data flow transmitted. The *DataProviderT* component type (lines 4-5) models the abstract behavior of data providers. The *IP* component type (lines 6-14) extends the *DataProviderT* type to represent data providers based on IP networks. It has three different *Representation* elements, each one modeling an alternative possibility of realization of the *IP* component: *IPTV* (lines 8-9), *P2PTV* (lines 10-11) and *InternetTV* (lines 12-13). The *INetworkT* connector type (lines 15-22) models the network interface via which data providers connect to the mechanism of selection and recovery of data flow. It provides alternatives: wired networks (lines 18-19) and wireless networks (lines 20-21). Figure 3(b) graphically shows the Ginga architectural vocabulary.

3.2 Specifying the SPL Core Architecture

During the definition of an ACME *Family*, it is possible to define architectural element types and also to instantiate elements within the *Family*. When elements are instantiated within a *Family,* these elements are inherited by all systems that adhere to that *Family*. PL-AspectualACME exploits this mechanism to describe the SPL core architecture, i.e., the default architecture that is shared by all products of a SPL. Hence, elements that represent SPL mandatory features are instantiated within the *Family*, forcing that all systems that adhere to that *Family* inherit these elements.

The Ginga core architecture is based on five main abstract components.The *Data Provider* component abstracts the sources of data provided to the Ginga middleware. The *Tuner* component abstracts the mechanism of selection and recovery of the data stream transmitted by data providers. The *Demuxer* component abstracts the demultiplexer mechanism that transforms the data stream received from *Tunner* components into more elementary data flows. The *Media Player* component processes the data flows transmitted by *Demuxer* components.

```
1. Family Ginga = {
2. ...
3. Component terrestrial:new
                TerrestrialT;
4. Connector i_net:INetworkT = new
              INetworkT extended with {
5.    Property selected_variants=
            {Wired}}
6. Component tuner:TunerT = new TunerT;
7. Connector ts_bus:InnerBusT = new
            InnerBusT;
8. Component demuxer:DemuxT = new
            DemuxT;
9. Connector ds_bus1, ds_bus2:
            InnerBusT = new InnerBusT;
10. Component player : PlayerT = new
      PlayerT extended with = {
11.   Port playAudio;
12.   Port playVideo;
13.   Representation = {
14.     System PlayerSystem = {
15.       Component audio : AudioT =
            new AudioT extended with= {
16.         Property
              selected_variants={AAC}}
17.Component video:VideoT = new VideoT
     extended with = {
18.   Property selected_variants=
        {H264}}}
19. Bindings {
20.    player.play to audio.play;
21.    player.play to video.play;} }
22. Attachments {
23.    i_net.consumer to tuner.listen;
24.    i_net.provider to
             terrestrial.provide_data;
25.    ts_bus.consumer to
             demuxer.ts_filter;
26.    ts_bus.provider to
             tuner.ts_deliver;
27.    ds_bus1.consumer to
             player.playAudio;
28.    ds_bus1.provider to
             demuxer.ts_transmitter;
29.    ds_bus2.consumer to
             player.playVideo;
30.    ds_bus2.provider to
             demuxer.ts_transmitter;} }
```

Fig. 4. Ginga abstract SPL Core Architecture

Figure 4 depicts the description of the Ginga SPL core architecture. *Terrestrial* (line 3) models the mandatory data provider, which transmits a terrestrial signal. The *i_net* connector (lines 4-5) models the wired network interface by which the *terrestrial* and *tuner* components (line 6) communicate. *Demuxer* (line 8) receives transport streams from *tunner* through the *ts_bus* connector (line 7). *Player* (lines 10-16) encapsulates the *audio* component (lines 15-16), which models the audio player that supports processing audio files encoded by AAC coding schema, and also the *video* component (lines 17-18), which models the video player that supports video files encoded by H264 coding schema. In the bindings section (lines 19-21), the outer *playAudio* and *playVideo* ports (lines 20-21) of the *player* component are respectively associated to the inner *play* port of both *audio* and *video* components. *Demuxer* transmits data streams to the *player* component via the *ds_bus1* and *de_bus2* connectors (line 27-30). Figure 5 graphically depicts the Ginga SPL Core Architecture.

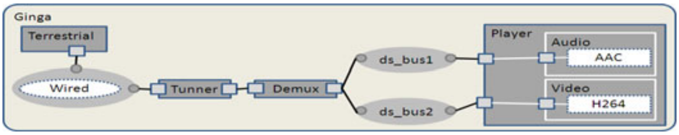


Fig. 5. Ginga Core SPL Architecture

3.3 Specifying SPL Products

The derivation process is basically the selection of which variable features will be available in a product. In PL-AspectualACME the variable features are modularized within *Representation* elements. Annotations in the *Property* elements are used to specify which *Representation* elements are selected or not; a derivation tool interpret the annotations, adding or removing the correct variant elements in the specification. The instantiation process adopted by PL-AspectualACME consists in instantiating members of the ACME family associated to the SPL. During the instantiation process, it is necessary to: (i) instantiate architectural elements based on the types defined by the family; and (ii) define which variants of each type must be selected.

Table 1. Features of *Ginga Zapper* product

Variation Point	***Mapped to***	***Selected Variant***	***Variation Point***	***Mapped to***	***Selected Variant***
Tuner	*tuner* component *provider* component *i_net* connector	Terrestrial Wired	*Data Processing*	*processor* component	Software Update
Demultiplexer	*demux* component	Hardware	*Input Manager*	*input_mng* component	Remote Control
Media Processing	*text* component *video* component *audio* component	TXT H.264 AAC	*Platform*	*platform*	ST

Table 1 shows the configuration of the *Ginga Zapper* product. It is the simplest product in the Ginga SPL. It provides only basic features to receive the TV signal.

Figure 6 shows the PL-AspectualACME textual description of the *Ginga Zapper* instantiation. The *Zapper* product is an instance of the *Ginga* family (line 1). The Tuner variation point is mapped to the *tuner* and *provider* components and the *i_net* connector. Since the *tuner* component does not have any variation point (and it was declared within the *Ginga* Family definition) there is no need to re-declare it within the *Zapper* product definition. The same applies to the *audio* and *video* components. For each variation point, the respective elements are instantiated, and the definition of the selected variant is defined via the *selected_variants* property. *Return Channel* and *Conditional Access* features are not mapped to any component in the *Zapper* system.

```
1. System Zapper:Ginga = new Ginga
      extended with = {
2.   Component provider:TerrestrialT =
new TerrestrialT;
3.   Connector i_net:NetworkInterfaceT=
      new NetworkInterfaceT extended
      with = {
4.   Property selected_variants =
        {Wired}}
5.   Component demux : DemuxT = new
       DemuxT extended with {
6.     Property selected_variants =
          {Hardware} }
7.   Component text : TextT = new TextT
     extended with = {
8.     Property selected_variants =
          {TXT}}
9.   Component processor:DataProcessingT
       =new DataProcessingT extend with={
10.    Property selected_variants =
         {Update}}
11. Component input_mng : InputManagerT
      = new InputManagerT extend with= {
12.    Property selected_variants =
         {Remote}}
13. Component platform : PlatformT =
       new PlatformT extend with = {
14.  Property selected_variants = {ST}}
15. Attachments {...}
16. ...}
```

Fig. 6. Specifying the SPL Core Architecture

4 Final Remarks

In this paper, we presented PL-AspectualACME, a flexible and extensible ADL for modeling SPL architectures with AO abstractions. PL-AspectualACME seamlessly adapts concepts of ACME to represent SPL variabilities at the architectural level. We use the *Representation* element to represent products variants. In this manner, we maintain the simplicity of the language, avoiding the addition of new abstractions, even at the cost of burdening semantically the *Representation* element. The language relies on typical architectural elements and provides mechanisms to specialize these elements in new types defined by software architects. These types constitute the architectural vocabulary by which software architects describe SPL architectures. We also presented the SPL architectural description of Ginga.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
2. Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA, SEI, Carnegie Mellon University (1990)
3. Garlan, D., Shaw, M.: An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering. World Scientific Publishing, Singapore (1993)
4. Medvidovic, N., Taylor, R.N.: Exploiting Architectural Style to Develop a Family of Applications. In: IEEE Proc. on Software Engineering, vol. 144(5-6), pp. 237–248 (1997)
5. Batista, T., et al.: Reflections on Architectural Connection: Seven Issues on Aspects and ADLs. In: Proceedings of the 5th International Workshop on Early Aspects at ICSE (2006)
6. Garlan, D., et al.: ACME: An Architecture Description Interchange Language. In: Proc. of the 1997 Conf. of the Centre for Ad. Studies on Collaborative Research. IBM Press (1997)
7. Ginga: The Brazilian Terrestrial Digital TV System Middleware, `http://www.ginga.org.br/`
8. Saraiva, D., et al.: Architecting a Model-Driven Aspect-Oriented Product Line for a Digital TV Middleware: A Refactoring Experience. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 166–181. Springer, Heidelberg (2010)
9. Figueiredo, E., et al.: Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability. In: Proc. of ICSE, 261–270 (2008)
10. Loughran, N., et al.: Language Support for Managing Variability in Architectural Models. In: Proc. of Software Composition, pp. 36–51 (2008)
11. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of the 14th European Conference on Object-Oriented Programming (1997)

Design and Evaluation of a Process for Identifying Architecture Patterns in Open Source Software

Klaas-Jan Stol[1], Paris Avgeriou[2], and Muhammad Ali Babar[3]

[1] Lero—The Irish Software Engineering Research Centre, University of Limerick, Ireland
[2] University of Groningen, The Netherlands
[3] IT University of Copenhagen, Denmark
klaas-jan.stol@lero.ie, paris@cs.rug.nl, malibaba@itu.dk

Abstract. Architecture patterns have a direct effect (positive or negative) on a system's quality attributes (e.g., performance). Therefore, information about patterns used in a product can provide valuable insights to, e.g., component integrators who wish to evaluate a software product. Unfortunately, this information is often not readily available, in particular for Open Source Software (OSS) products, which are increasingly used in component-based development. This paper presents the design and evaluation of a process for Identifying Architecture Patterns in OSS ("IDAPO"). The results of the evaluation suggest that IDAPO is helpful to identify potentially present patterns, and that a process framework may provide better opportunities for tailoring to the users' needs.

Keywords: architecture patterns, quality attributes, open source software, empirical evaluation, quasi-experiment.

1 Introduction

Architecture patterns (e.g., layers, model-view-controller) are generalized solutions to recurring system-wide design problems, which describe the main roles of system partitions and their interactions. It is widely recognized that architecture patterns have a direct effect on a system's quality attributes (QAs), such as performance and reliability [1]. Integrating components into a system whose overall QAs are incompatible with the component's QAs will hinder the achievement of a system's quality requirements. Since architecture patterns are documented solutions with known properties [2], knowledge of architecture patterns used in a software component can provide valuable insights to component integrators about which QAs are supported, and which are hindered. Previously, the use of architecture patterns has been shown to be an effective and lightweight complementary approach to traditional architecture review methods (e.g., ATAM [3]) to perform architecture reviews [4].

Unfortunately, this information about architecture patterns used in a component is often not readily available, in particular for Open Source Software (OSS) products. OSS products are increasingly used in industry [5], but the quality of products varies widely. Therefore, it is important that OSS *integrators* [6] thoroughly evaluate an OSS product before it is integrated into a system [7]. Documentation of OSS products often

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 147–163, 2011.

lacks information about their design (including patterns that are used) [8]. The literature provides little guidance to practitioners who would wish to identify architecture patterns. Existing tools for pattern identification are limited to (object-oriented) design patterns, such as documented in [9]. The varying granularity of components (e.g., a class/object as a component versus an executable as a component) makes automated identification of architecture patterns inherently difficult. Reverse engineering methods and tools may help to reverse-engineer the architecture, but do not typically focus on identifying *architecture* patterns [10].

This lack of guidance on how and where to find architecture patterns in OSS products motivated us to investigate how this task can be supported. In our previous work [11], we proposed a conceptual process to streamline the task of identifying architecture patterns in OSS. This paper reports on two additional empirical studies that contribute (a) a validation of the process steps and enhancement of our initial process, and (b) an evaluation of the resulting process. The enhanced process was named IDAPO (IDentifying Architecture Patterns in OSS, pronounced as "Idaho").

This paper proceeds as follows. We present background and motivation in Section 2. Section 3 presents the design history as well as validation and enhancement of IDAPO. Section 4 presents the design and results of a quasi-experiment to evaluate the usefulness of IDAPO. We discuss the findings of the experiment in Section 5. Section 6 concludes and provides an outlook to future work.

2 Background and Motivation

Software Architecture, Patterns and Quality Attributes. Software architecture has been shown to be an important artifact in the software development process [1]. It constitutes a set of architectural design decisions, such as the use of an architecture style or pattern. Most software architectures apply one or more architecture patterns [12]. For instance, architects speak of a 'layered system', or a 'model-view-controller' architecture. Architecture patterns have a documented effect (positive or negative) on a system's quality attributes (QAs) (e.g., performance, reliability) [1, 2, 13]. For instance, a system with layers is likely to be modifiable, as it facilitates a clear separation of concerns. However, passing large numbers of messages up and down the layer 'stack' may cause performance issues [13]. Therefore, one effective way to select OSS products that support the achievement of the system's QAs is to acquire sufficient information of architecture patterns used in those products. Once this information is available, OSS integrators can use the rich information in the pattern documentation (in pattern languages such as [2]) about the potential impact of the pattern's solution on the system QAs [4].

Use and Evaluation of OSS. Over the last decade, an increasing number of software developing organizations is integrating OSS products in component-based development [5, 7]. However, selecting suitable OSS products is a key challenge [8]. To address this, researchers and industry have proposed a variety of OSS evaluation and selection approaches [14]. Typically, these approaches prescribe a list of criteria, such as the level of activity of the OSS community and the number of open bugs, categorized in some categories (e.g., *product*, *community*), on which an OSS product is evaluated. The output is a weighted average of the scores for the criteria. The goal of these approaches is to provide practitioners with some guidance on the process of evaluating an OSS product. However, these approaches typically do not consider the architectural aspects of a product to assess its impact on a system's QAs. While the abovementioned existing approaches may provide valuable information such as the potential support provided by an OSS product's community, we argue that those methods could be used in tandem with appropriate approaches to identify and understand the architecture patterns used, such as proposed by us. The former assess the maturity of an OSS product, while the latter helps understand the impact on system QAs. While knowledge of architecture patterns is equally important for closed (proprietary) software, we focus our efforts on OSS, since it has become a viable alternative to commercial off-the-shelf (COTS) components [15]. Furthermore, due to the closed nature of COTS components, identifying architecture patterns is virtually impossible for such products.

Pattern Identification. A number of pattern identification techniques and tools have been proposed [9]. However, these techniques and tools focus on the identification of design patterns, which may not have a direct impact on the fundamental structure of a software system [2]. Furthermore, design patterns such as those presented by the Gang of Four [16], are object-oriented, which assumes that the software is written in an object-oriented programming language. These techniques and tools, however, do not support identifying architecture patterns. There are no techniques to automatically identify architecture patterns from source code, and there are a number of obstacles that prevent this. Firstly, there are no commonly accepted formalisms for describing components and connectors between them. Proposed formalisms such as Architecture Description Languages (ADLs) and the UML have several issues [17]: different ADLs focus on modeling different types of systems, and they vary greatly in their expressiveness of software architecture concepts. These obstacles hinder the reverse engineering of source code into a formalism that expresses patterns. The use of reverse engineering tools that could support pattern identification is associated with various challenges [11]. A second obstacle that hinders automated pattern identification is that patterns may be implemented in 'a thousand different ways' [2], and have to be implemented (and customized) according to the specific needs at hand. Therefore, we argue that identifying architecture patterns depends to a large extent on manual techniques. Our proposed process is designed to support this task. In the remainder of this paper, we use the word 'pattern' to refer to *architecture* pattern rather than *design* pattern.

3 Design of the Process

3.1 Design History of IDAPO

We are investigating how practitioners can be guided in the task of identifying architecture patterns. Fig. 1 shows the three empirical studies we have conducted so far.

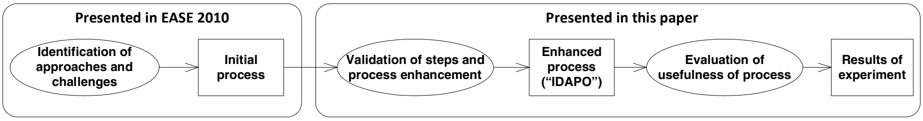

Presented in EASE 2010
Identification of approaches and challenges
Initial process
Presented in this paper
Validation of steps and process enhancement
Enhanced process ("IDAPO")
Evaluation of usefulness of process
Results of experiment

Fig. 1. Overview of research activities and research output. Ovals represent research activities; rectangles represent research outputs.

Previously, we have reported an initial study in which we identified approaches and challenges of 23 master's students that had performed a pattern-identification task in the context of master's courses on Software Architecture and Software Patterns [11]. Based on these findings, we suggested a systematic approach to identify patterns, and presented an initial process definition to support practitioners in this task. As shown in Fig. 1, the current paper reports two additional empirical studies that we conducted. The first (presented in Section 3.2) aimed at validating the process steps and enhancing the process; the second (presented in Section 4) aimed at empirically evaluating the usefulness of our enhanced process (IDAPO) through a quasi-experiment.

3.2 Process Steps Validation and Process Enhancement

In [11] we presented an initial version of a process to support the task of identifying architecture patterns in OSS. In order to validate the different steps of the initial process and enhance the process, we invited all 12 enrolled students who had performed a pattern identification assignment in the context of a master's course on Software Patterns (at the University of Groningen) for a semi-structured interview. Ten students chose to participate. We did not show the participants our initial process in order to prevent getting only confirmatory answers. Instead, we asked the students about the steps taken, their usefulness, what information they had been looking for, obstacles they had encountered, and their "lessons learned", i.e., what steps they would and would not take again.

We digitally recorded the interviews with the participants' consent. Dutch students were interviewed in their native language. The other students were interviewed in English. The interviews lasted 60 minutes on average. All interviews were transcribed verbatim by the interviewer. The Dutch transcriptions were translated into English to allow the other researchers to participate in the data analysis. The data were analyzed using qualitative data analysis techniques [18]. We systematically extracted information about the steps that the students had taken and recorded them in a spreadsheet. We compared the steps to the activities of our initial process presented in [11]. We focused primarily on the steps that students had considered to be useful; for instance, many students considered the use of reverse engineering tools to be a waste of time.

We found that the steps taken by the students mostly corroborated the activities of our initial process. We also found reason to make a number of changes based on new insights gathered from the 10 interviews. While Fig. 2 presents the enhanced process in more detail, in this section we briefly summarize the changes made. We realized that an incremental accumulation of information, such as type and domain of the product as well as implementation technologies used, could be a useful way to identify potentially used patterns, which we refer to as *candidate patterns*. Therefore, we swapped the order of steps (4, 5) with steps (2, 3) in the initial version presented in [11]. A second change we made was to enrich the process with a data flow between the steps, which describes the different pieces of information that can be gathered as a user follows the steps as well as which steps use this information. Thirdly, we used the Business Process Modeling Language (BPMN) to define the process to replace the UML activity diagram notation we used before. This allowed us to more clearly express the different steps of the process. Section 3.3 presents the enhanced process that we named IDAPO.

3.3 IDAPO: A Process for Identifying Architecture Patterns in OSS

A key feature of IDAPO is the idea of incremental accumulation of information. In each step, information regarding the use of patterns in a product is acquired. By systematically recording information about the product's characteristics, a practitioner is encouraged to make details of the product under investigation explicit, which helps the practitioner's analytical thought process. To formalize this idea in IDAPO, we added a 'data flow' to the process, to suggest what information is generated and needed for each step. The resulting definition of IDAPO is shown in Fig. 2 in BPMN. In BPMN, rounded rectangles represent activities; normal lines represent *control flow* (sequence of steps), whereas dotted lines represented *data flow* (which indicate input and output to the various activities). The OSS community is represented by a separate *pool*. In BPMN, a pool represents an organization and is used to border process participants (the default pool is implicit). For more details of BPMN, see [19]. The remainder of this section describes the process steps in more detail; step numbers are enclosed in parentheses.

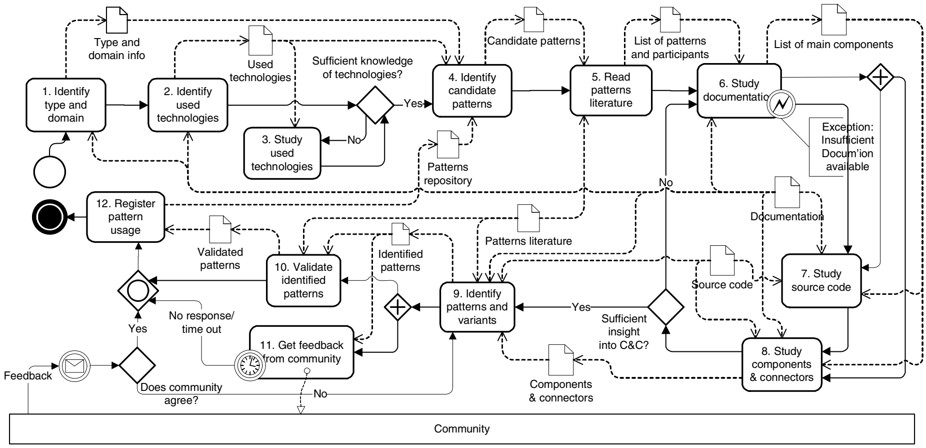

Type and domain info
Used technologies
Sufficient knowledge of technologies?
Candidate patterns
List of patterns and participants
List of main components
1. Identify type and domain
2. Identify used technologies
Yes
4. Identify candidate patterns
5. Read patterns literature
6. Study documentation
3. Study used technologies
No
Patterns repository
Exception: Insufficient Docum'ion available
No
12. Register pattern usage
Documentation
Validated patterns
Patterns literature
Identified patterns
10. Validate identified patterns
Source code
7. Study source code
9. Identify patterns and variants
Yes
No response/ time out
Sufficient insight into C&C?
Yes
11. Get feedback from community
8. Study components & connectors
Feedback
Does community agree?
No
Components & connectors
Community

Fig. 2. IDAPO: a process for identifying architecture patterns in OSS

The first step is to (1) **identify the type of software** and its domain. Knowledge of the type and domain of the software may provide hints about the use of certain patterns. For instance, an instant messenger product is likely to use the client-server pattern. Step two is to (2) **identify technologies used** for implementation. If, for instance, CORBA (Common Object Request Broker Architecture) was used, it may be useful to look for the broker pattern. If the process user has insufficient knowledge of technologies, it is advisable to (3) **study those technologies**, which may help in understanding how the system under scrutiny was implemented. Based on the information gathered in previous steps, (4) **candidate patterns may be identified** (i.e., potentially present patterns) and listed. After identifying candidate patterns, (5) the **patterns literature** (e.g. [2]) can be studied to learn more details about those patterns, which will help in recognizing and asserting that the patterns are, in fact, present. The next step is to (6) **study project documentation**, from which insights into the system's architecture, components and connectors may be gathered (note that the documentation could also be consulted in previous steps). After identifying candidate patterns and studying project documentation, the next step is to (7) **study the source code** and crosscheck with the findings of the documentation. It is important to gain insight into the various (8) **components and connectors** in the system under investigation, since this will help to identify which patterns have been used in the system. Once sufficient information is gathered through studying documentation, source code and components and connectors, the actual (9) **pattern matching** and identification activity starts. This involves comparing the structure and behavior of the pattern to the product's structure. After identification, it is important to (10) **validate the identified patterns** to make sure they have been correctly identified. One way to do this is through peer-review by others (e.g., colleagues). Findings may also be presented to the community for feedback. While the (11) **community may be contacted** earlier to ask for information, our experience has shown that providing some input is more likely to result in a reply. Once identified patterns have been confirmed, the (12) **patterns should be registered** in a patterns repository for later use by others. A few researchers have proposed such repositories for patterns [20] or architectures in general [21]. Over time, the patterns repository will be populated with information of many systems, which we envisage to be a valuable tool for others in understanding the architecture of OSS products.

4 Evaluation of the Process: A Quasi-Experiment

Following the call by Falessi et al. [22] to perform empirical evaluation of new techniques to improve the state of practice in software architecture, we decided to empirically evaluate the usefulness of IDAPO by means of an experiment [23]. We measure usefulness in terms of the number of patterns that are identified. This section is structured following the reporting guidelines for experiments in [24].

4.1 Experiment Goals and Hypotheses

We defined three goals for this experiment. Firstly, we are interested in whether using IDAPO helps to identify more patterns. We argue that the task of *correctly* identifying architecture patterns depends on practitioners' expertise and experience; if IDAPO results in more identified patterns, this expertise is important to assess their correctness. However, not all practitioners have extensive expertise to draw from. In order to be able to more precisely evaluate the usefulness of IDAPO, our second goal was to measure the output in terms of two standard measures: *precision* and *recall* [25]. Thirdly, we wanted to investigate to what extent IDAPO supports the identification of *candidate patterns* based on information gathered in the first three steps. To investigate these goals, we defined six hypotheses, which we discuss next.

To address the first goal, we decided to simply count the number of identified patterns, disregarding whether the patterns are correct or not. IDAPO describes the steps to take, and the information required to identify patterns. Hence the first hypothesis:

H_{01}: Using IDAPO does not change the number of identified patterns.

For all hypotheses, we imply a comparison to the number of patterns identified when not using IDAPO.

Besides looking at the number of identified patterns tested in H_{01}, it is also useful to use standard measures based on a confusion matrix, namely *precision* and *recall* [25]. Precision is defined as the fraction of patterns correctly identified of the total number of identified patterns, i.e., true positives ÷ (true positives + false positives). Recall is defined as the fraction of correctly identified patterns of the total number of correct patterns present, i.e., true positives ÷ (true positives + false negatives). Hence, we defined hypotheses H_{02} and H_{03}.

H_{02}: Using IDAPO does not change the *precision* of *identified* patterns

H_{03}: Using IDAPO does not change the *recall* of *identified* patterns.

As mentioned in Section 3.2, the process emphasizes a step-wise, incremental approach to gather information in a systematic way. In particular, we are interested in the *candidate* patterns based on the first few steps of the process. In order to test this idea, we defined hypothesis H_{04}:

H_{04}: Using IDAPO does not change the number of *candidate* patterns.

Likewise, we decided to also test precision and recall rates for the *candidate* patterns; Hence, we defined to H_{05} and H_{06}:

H_{05}: Using IDAPO does not change the *precision* of *candidate* patterns.

H_{06}: Using IDAPO does not change the *recall* of *candidate* patterns.

For each hypothesis, we imply an alternative hypothesis H_{an} (n=1 to 6) that states that the use of IDAPO *does* result in a higher number of (candidate) patterns.

4.2 Participants and Training

We invited 24 master's students who were enrolled in a course on Software Patterns at the University of Groningen, to participate in our experiment. Participation was not compulsory, but students were advised to participate, as one of the upcoming course assignments would also be to identify patterns in an OSS product in order to perform a pattern-based architecture review [4]; our embedded study with the students was therefore integrated with the course [26]. Fourteen students chose to participate. Table 1 presents demographic information of the participants.

Table 1. Participants of the experiment

Group	ID	Age	Work experience	Degrees	Nationality
Control	P1	24	3½ years, developer	B. (CS)	Netherlands
	P2	27	—	B. (AIM)	Greece
	P3	28	¼ year, developer	B. (CS)	Argentina
	P4	28	—	B. (BI, CS)	Netherlands
	P5	25	1 year, web developer	B. (CS)	Greece
	P6	29	5 years, developer	B. (CS); M. (Psy)	Belgium
	P7	25	—	B. (BI)	South Africa
Treatment	P8	27	2 years, developer	B. (CS)	Netherlands
	P9	25	1 year, web developer	B. (CS)	South Africa
	P10	23	2 years, web developer	B. (CS)	Netherlands
	P11	24	5 years, OSS developer	B. (CS)	Netherlands
	P12	25	2 years, web developer	B. (CS)	Spain
	P13	22	—	—	Netherlands
	P14	21	—	B. (CS)	Netherlands

Section 4.4 discusses the assignment procedure to the groups. The average age of the control group was almost 24, whereas the average age of the treatment group was approximately 26½. Note that work experience should be interpreted as part-time jobs. One participant (P11) actively contributed to a small OSS project. All but one participant (P13) had finished a bachelor's (B) degree in computer science (CS), bioinformatics (BI) or applied informatics and multimedia (AIM). One participant (P6) also had a master's (M) degree in psychology (Psy). Most of them had varying levels of expertise in different topics, as listed in Table 2, e.g., three participants assessed themselves as having *advanced* knowledge of software engineering.

When we conducted the experiment, the students had attended six 2-hour lectures of the 8-week course on Software Patterns. All students also had followed a course on Software Architecture. The data presented in Tables 1 and 2 were gathered through a pre-study questionnaire the day before the experiment.

Table 2. Participants' self-assessed levels of expertise

Topic	None	Beginner	Intermediate	Advanced	Expert
Software engineering	0	5	6	3	0
Software architecture	0	6	7	1	0
"Gang of Four" design patterns	3	5	5	1	0
Architectural patterns	0	10	4	0	0
Development process of OSS	5	5	4	0	0
Experience w. integrating COTS	4	3	4	2	1

4.3 Task and Materials

The task given to the participants was to identify as many architectural patterns in a specified OSS project as possible: the JBoss application server. We selected JBoss for three reasons. Firstly, it is an industry-strength system (no 'toy' project), which is widely used in industry. Secondly, we expected that the participants would be able to find sufficient information about this product in the limited available time, since it is well known and extensive documentation is available. Thirdly, we already had insight into the architectural patterns used in this product, which we would need as a marking scheme for assessing the number of *correctly* identified patterns as well as the precision and recall. Participants in both groups were handed out the assignment form. The treatment group was given two additional instruments: (1) our process as shown in Fig. 1 accompanied by a description of each step; and (2) a simple spreadsheet template to record information found in each step. Additionally, the participants had access to the five volumes of the Pattern-Oriented Software Architecture (POSA) series of books (e.g., [2]), which list various software patterns.

4.4 Experiment Design

The experiment design was a between-subjects design, to compare results from a control group and a treatment group. Based on our previous experience in conducting research with students, we expected that the participants would have varying levels of experience and expertise. Since this would have constituted a threat to the outcome of the experiment, we decided to non-randomly assign participants to the control and treatment groups. Hence, this experiment was a *quasi*-experiment [27, 28]. Eight participants had indicated to have other course obligations in either the morning or afternoon of the day of the experiment; based on this information, three participants were assigned to the control group, and five were assigned to the treatment group. Based on the information about work experience gathered in the pre-study questionnaire, we assigned the remaining six students, resulting in two equally sized and approximately equivalent groups (see Table 1).

The *treatment*, or *independent variable* manipulated by this study is the reference process, with one treatment: IDAPO is provided, and one control: IDAPO is not provided. The *dependent variable* is the number of architecture patterns identified by the participants using and not using the process.

4.5 Experiment Procedure

We conducted the experiment in two sessions. The control group performed the task in the morning session, and the treatment group (provided with IDAPO) was invited for the afternoon session. This order ensured that the control group did not see IDAPO (to prevent the diffusion or imitation of treatment threat [29]). In both sessions, the researcher gave a brief introduction (15 min) to explain the background and motivation of identifying patterns. For the treatment group, the researcher also explained the different steps of IDAPO. Both groups were given two hours for this task. One participant in the control group had to leave 30 minutes early due to other course obligations (P1). After the two hours, participants were asked to fill out a post-study questionnaire; we used separate post-questionnaires for the two groups, as only the treatment group could be asked about their experiences with IDAPO.

4.6 Analysis and Results

4.6.1 Establishing a Set of Trusted Patterns

In order to be able to determine precision and recall measures, we need to compare the findings to a certain set of “correct” patterns of which we are confident that they are present in the product. In order to establish such a trusted subset of patterns, we used three different sources. Firstly, we used a research report that presents an analysis of the JBoss architecture (v.2.2.4, 2002) [30]. Secondly, we used a technical report (from 2005) that reports on the architecture recovery of JBoss [31]. Thirdly, we used a report from a previous group that had identified patterns in JBoss in the context of the 2009 edition of the Software Patterns course (mentioned in [11]); one of its authors had extensive professional experience as an administrator of JBoss. Table 3 lists patterns identified by the different sources, as well the patterns that we decided to include in the trusted subset.

We made this selection based on the reports, which described the patterns and their location in JBoss, as well as our level of confidence that we had in the presence of these patterns. During our selection, we also considered that the different sources have studied different versions of JBoss. We could not find sufficient justification to include the *Pipes-Filters* and the *Factory* patterns. The column ‘Trusted’ indicates which patterns are included in the trusted subset. We listed all patterns identified (for both control and treatment group) in a spreadsheet. In order to calculate precision and control measures, we counted only those patterns that were listed in the trusted list (Table 3).

Table 3. Derivation of a trusted subset of patterns

Pattern	Liu	Salehie et al.	Report 2009	Trusted
Microkernel	Yes	Yes	Yes	Yes
Layers	Yes	Yes	-	Yes
Pipes & Filters	Yes	-	-	-
Broker	Yes	-	Yes	Yes
Dynamic proxy	Yes	-	Yes	Yes
Proxy	Yes	Yes	-	Yes
Interceptor	Yes	Yes	Yes	Yes
Client/server	-	-	Yes	Yes
Active repository	-	-	Yes	Yes
Factory	-	-	Yes	-

4.6.2 Descriptive Statistics

Table 4 presents the descriptive statistics of the results. We counted the number of identified patterns of the control and treatment group as a whole. The first three columns list the results when counting *all* patterns, disregarding their correctness; column 1 lists the total number of patterns of the control group (18); column 2 lists the total number of candidate patterns identified by the treatment group ('T. candid.', 36), and column 3 lists the total number of identified patterns (as output of step 9 in the process, see Fig. 2) listed by the treatment group ('T. final', 16).

Table 4. Number of patterns per group, mean and standard deviation. Columns 1-3 consider all patterns identified; columns 4-6 only consider the trusted patterns.

	Counting all patterns			Counting trusted patterns only		
	(1) Contr.	**(2) T. candid.**	**(3) T. final**	**(4) Contr.**	**(5) T. candid.**	**(6) T. final**
Total	18	36	16	10	21	10
Mean	2.6	5.1	2.3	1.4	3.0	1.4
Std. dev	2.1	2.3	2.4	0.9	1.1	1.8

Columns 4-6 show the results similarly as column 1-3, but only taking the *trusted* patterns into account (resulting in 10, 21 and 10 patterns, respectively). When counting trusted patterns only, there is no difference between the control group and the final results of the treatment group. The treatment group as a whole identified 21 candidate patterns, which suggests the treatment group was on the right track. Fig. 3 shows boxplot diagrams for the results presented in columns 1-6.

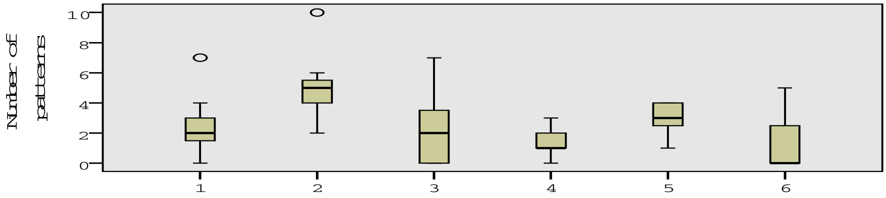


Fig. 3. Distribution of numbers of identified patterns by the control group, treatment group (candidate and final). Boxplots 1-3: counting all patterns, corresponding to columns 1-3 in Table 9. Boxplots 4-6: trusted patterns only (corresponding to columns 4-6 in Table 4).

Table 5 presents the mean and standard deviation values of the *precision* and *recall* rates, calculated for the control group and the treatment group. For the latter, we calculated precision and recall both for the candidate results and the final results. Table 5 shows that the average precision of the control group is 0.56 with an average recall of only 0.18. This suggests that about half of the control group's patterns are correct, but that (on average) less than 20% of the trusted patterns were identified. The *candidate* results of the treatment group score better, with a precision of 0.62 at a recall rate of 0.37, suggesting that (on average) the treatment group identified more correct candidate patterns. However, when looking at the *final* results of the treatment group precision is only 0.30 at a recall of 0.18, worse than the control group. The relative high values for the standard deviations of precision (0.35) and recall (0.23) for the treatment group's final results suggest a large variation among participants. We found that three participants in the treatment group did not list any "final" patterns (as opposed to one participant in the control group).

Table 5. Precision and recall for control, treatment candidate and treatment final results

	Control		**Treatment**			
			Candidate patterns		**Final**	
	Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Mean	0.56	0.18	0.62	0.38	0.30	0.18
Std. dev.	0.32	0.11	0.21	0.13	0.35	0.23

4.6.3 Results of Statistical Analysis

We performed statistical analyses on the number of identified patterns and the calculated precision and recall rates to test the six hypotheses. The assumptions underlying parametric tests such as the t-test were not fulfilled [32], since the data contained outliers and we could not assume that the data have a normal distribution. Therefore, we decided to use the Mann-Whitney U test, which is a non-parametric alternative to the t-test for two independent samples [32]. Table 6 lists the p-values for each of the six hypotheses. The columns 'Candidate' and 'Trusted' indicate whether the hypotheses consider the candidate patterns (of the treatment group) and whether only trusted patterns were counted, respectively. We reject a hypothesis if the p-value is less than the significance level of $\alpha=0.05$. We used SPSS version 18 for all statistical tests.

Table 6. Hypotheses and resulting p-values of the Mann-Whitney U test

Hyp.	**Variable**	**Candidate**	**Trusted**	**P-value**	**Decision**
H_{01}	Number of identified patterns	No	No	0.435	Retain H_{01}
H_{02}	Precision of identified patterns	No	Yes	0.324	Retain H_{02}
H_{03}	Recall of identified patterns	No	Yes	0.597	Retain H_{03}
H_{04}	Number of candidate patterns	Yes	No	**0.051**	Retain H_{04}
H_{05}	Precision of candidate patterns	Yes	Yes	0.555	Retain H_{05}
H_{06}	Recall of candidate patterns.	Yes	Yes	**0.026**	**Reject H_{06}**

Table 6 shows that we could not find compelling evidence to reject hypotheses H_{01}-H_{05} (all p-values > $\alpha=0.05$). In other words, there was not sufficient evidence to conclude that using IDAPO resulted in a higher number of identified patterns (H_{01}), a higher precision of identified patterns (H_{02}), a higher recall of identified patterns (H_{03}), a higher number of candidate patterns (H_{04}), and a higher precision of candidate patterns (H_{05}). With respect to H_{04}, we found that there is some evidence (p=0.051) that using IDAPO results in a higher number of *candidate* patterns, but since the p-value is smaller than our significance level (α=0.05) we do not reject H_{04}. On the other hand, we found evidence ($p=0.026 < 0.05$) to **reject** hypothesis H_{06} ('using IDAPO does not change the recall of candidate patterns'). Together, these results suggest that IDAPO helps to improve the **recall** of candidate patterns.

4.6.4 Results of Post-study Questionnaires

The post-study questionnaire questions were rated using a five-point Likert scale, ranging from Totally Disagree (TD), to Disagree (D), Neutral (N), Agree (A) and Totally Agree (TA).

Results of Treatment Group. Table 7 presents the results for the treatment group. Numbers indicate the number of participants that gave a certain rating, e.g., two subjects answered 'Neutral' on question T1.

Table 7. Post-study questionnaire results for the treatment group. High scores are highlighted

ID	Question	TD	D	N	A	TA
T1	I followed the process step by step in the order prescribed.	1	1	2	1	2
T2	Identifying the type and domain of the software is helpful.	0	1	2	2	2
T3	Identifying the used technologies is helpful to identify patterns.	0	1	1	1	4
T4	The process helped me to identify patterns that I wouldn't have found otherwise.	0	4	2	1	0
T5	The suggested order of steps in the process made sense.	0	2	2	1	2
T6	Storing information per step in the spreadsheet was useful.	0	2	2	1	2

Table 7 shows that the degree to which the subjects followed IDAPO varied (T1). This suggests that participants disliked the process rigid order of steps, and would like to have more flexibility. The results for T2 suggest that most subjects agree that identifying the type and domain of the software is helpful. The results show that identifying used technologies (T3) was considered to be very helpful. Most participants did not think that IDAPO helped to identify patterns that could not have been identified otherwise (T4). Two participants were undecided, and only one participant agreed. Participants were divided on whether the order of IDAPO's steps was sensible (T5). Also, participants were equally divided on whether storing information per step in a spreadsheet was useful (T6).

We also gathered results from a few open questions. Some suggestions were:

- A spreadsheet was not considered suitable to record intermediate information;
- The process should be made less sequential;
- After identifying type and domain, always read documentation to learn about components and used technologies.

The main challenges encountered by the treatment group were:

- To find the right documentation and information;
- Lack of time to read source code, and also to identify patterns;
- Unfamiliarity with JBoss.

Results of Control Group. Table 8 lists the questions and the scores for the control group. The results for C1 show that most participants either disagreed or were undecided on whether they knew what steps to take to identify architecture patterns. Only one participant indicated he knew what approach to take. This confirms our assertion that there is a need to provide some guidance in this task. The second question (C2) was to find out whether participants found sufficient information to identify patterns. Again, most participants indicated disagreement or neutrality. This suggests that, in general, there is a need to identify useful sources of information.

Table 8. Post-study questionnaire results for the control group

ID	Question	TD	D	N	A	TA
C1	I knew what steps to take to perform the assignment.	0	2	4	1	0
C2	I found sufficient information to identify patterns.	1	2	2	2	0

We also asked the participants for suggestions for improvement as well as challenges encountered. Some suggestions included:

- Use of a debugger to trace the execution to find relations among components;
- Search for images of the architecture that may lead to useful sources.

The main challenges encountered by the control group were:

- Unfamiliarity with JBoss; getting to know the system;
- Studying the source code is like finding a needle in a haystack;
- Finding the right information; inconsistent documentation; pattern naming is inconsistent (e.g. a ‘proxy’ component implementing the ‘dispatcher’ pattern)

5 Discussion

In this section we interpret the findings from the experiment, the post-study questionnaires and discuss implications for further research. The overall motivation for the development of IDAPO was to provide guidance to practitioners in identifying architecture patterns in an OSS product. While we did not find evidence that the use of IDAPO helped to identify more architecture patterns than the control group who did not use IDAPO, the results showed some modest advantages. The results of the experiment suggest that the use of IDAPO helped to identify more candidate patterns that were considered correct (based on a set of “trusted patterns” derived in subsection 4.6.1). This means that many potentially present patterns identified based on information about the product’s type, domain and used technologies (steps 1 and 2 in Fig. 2) turned out to be correct. Questions T2 and T3 in Table 7 confirm that participants considered these to be useful steps. These results suggest that IDAPO is helpful to identify candidate patterns. The use of IDAPO also helped to improve the recall of the candidate patterns, which indicates that compared to the control group, a larger number (on average) of correct patterns were recovered.

On the other hand, when considering only the *final* results of the treatment group, we did not find any evidence that the use of IDAPO resulted in more identified patterns than the control group (H_{01}), nor in a higher precision (H_{02}) or recall (H_{03}). This suggests that, while IDAPO helps to identify candidate patterns, the process does not provide sufficient guidance to assess that the patterns are in fact present. Questions T1, T4 and T5 in Table 7 seem to confirm this; participants did not follow the steps in the suggested order (T1), participants did not think that IDAPO exclusively helped to identify certain patterns (T4), and participants were divided on whether the order of steps made sense (T5). Based on these observations as well as the results of the post-study questionnaires of both the control and treatment groups, we point out the strong points of IDAPO as well as points for improvement. IDAPO is useful for guidance, but should not be prescriptive. Rather, a *process framework* seems to be more appropriate, from which a user can select appropriate activities to derive a process that is tailored to the context and needs of the user. This also allows prioritizing tasks in case that time is limited. Furthermore, it is important to investigate ways that a user can get familiar with a system more quickly as well as approaches to find appropriate documentation and information. Better ways to record intermediate information that support the user to manage this information and draw appropriate conclusions should be investigated. The use of tools (e.g., debuggers) could provide additional ways to acquire more information of a system's structure. Through our interviews we found that the use of tools was often not very helpful; therefore, we emphasize that it is important to understand how tools can provide support and what type of information can be acquired.

5.1 Threats to Validity

Conclusion Validity. The number of subjects is a threat to conclusion validity. Fourteen subjects were willing to participate in our experiment, which were divided into two groups (control, treatment) of seven. However, we did not intend to make conclusive statements based on this single experiment only. Rather, our results should be considered exploratory and help us to gain insights in the usefulness of IDAPO. **Construct validity**. There are a few limitations to construct validity. Firstly, we limited the total time for identifying patterns to two hours. This limitation has a direct effect on the amount of work that can be done, and therefore on the number of patterns that can be identified. Participants of the treatment group may have had to spend relatively much time on understanding the steps of IDAPO. However, we chose to limit the time duration in order to be able to recruit a sufficient number of subjects; as the time duration of an experiment increases, fewer participants will be willing to participate. **Internal validity**. We discuss a number of threats to internal validity, which is concerned with the degree to which a change of the dependent variable can be ascribed to a change of the independent variable. The first is instrumentation: the process description and diagram of IDAPO may not have been easy to understand by the treatment group. Though we explained the process to the treatment group, participants may not have fully understood the steps to take. Another instrumentation threat is our marking scheme for assessing "correct" patterns. This instrument (discussed in subsection 4.6.1) was used to perform calculations of precision and recall. Our conclusions depend on the extent to which we correctly confirmed the patterns. We derived this trusted subset of patterns based on three independent sources. However, the three reports do not fully agree on the patterns. In order to decide which patterns to include in the trusted subset, we have (a) studied the description of how the patterns were implemented, thereby assessing the credibility of the description and the pattern's usage, and (b) attempted to find additional information through web searches in order to be able to confirm them. It is noteworthy that the patterns listed by the three sources that we could not confirm (and therefore were not included in the trusted subset) were not identified by either group. Besides this, JBoss may contain patterns that have not been listed by any of the three reports, which means that these are not included in our trusted subset. The second threat is that of selection: the control and treatment groups may not be as equivalent as we intended in terms of work experience and knowledge of related topics (see Table 1). Furthermore, the average age in the control group was 2.5 years higher (26½) than the average age in the treatment group (almost 24); this difference suggests that the control group has a few more years of experience in the field of software engineering. This could have negatively biased the results of the treatment group, which strengthens the decision to reject hypothesis H_{06}. **External validity**. Threats to external validity are those that may limit the applicability of the results to industry practices. The use of master's students as subjects is an important factor that deserves attention, and has been discussed in the literature [26, 33, 34]. We do not consider the use of students to be a major threat, since it is not yet a common practice in industry to identify patterns in an OSS product. Some researchers mention that students are suitable to be used to evaluate new techniques [35]. Furthermore, since the participants were master's students (rather than undergraduates), they can be considered to be 'novice' professionals. A potential threat to external validity is that a treatment is applied on a 'toy' problem. However, we selected the JBoss application server for this experiment, which is an industrial-strength software product.

6 Conclusion and Future Work

In this paper we present and evaluate IDAPO: a process that provides guidance to practitioners who wish to identify architecture patterns in an OSS product. The process design is based on empirically identified steps. We have conducted a quasi-experiment to empirically evaluate IDAPO. We found evidence that the first few steps of IDAPO are particularly helpful to identify candidate patterns (potentially present in the product). We believe that IDAPO can be a valuable contribution to the toolkit of practitioners who need to evaluate OSS products. However, the results also suggested that the other steps of IDAPO could be improved. We believe that the process steps should become more flexible, and become part of a *process framework*, which can be tailored to the user's needs. Furthermore, it would be valuable to investigate how IDAPO can support identification of architectural *tactics*, such as documented in [1]. Tactics support the achievement of quality attributes and can therefore provide valuable insights similar to the information conveyed by patterns.

Acknowledgments. This work is partially funded by IRCSET under grant no. RS/2008/134 and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero.

References

[1] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented Software Architecture - A System of Patterns. J. Wiley and Sons Ltd., Chichester (1996)

[3] Kazman, R., Klein, M., Barbacci, M., Longstaff, T.: The Architecture Tradeoff Analysis Method. In: ICECCS, pp. 68–78 (1998)

[4] Harrison, N.B., Avgeriou, P.: Pattern-Based Architecture Reviews. IEEE Software (2011) (in Press)

[5] Hauge, Ø., Ayala, C., Conradi, R.: Adoption of Open Source Software in Software-Intensive Organizations - A Systematic Literature Review. Inf. Softw. Technol. 52(11), 1133–1154 (2010)

[6] Hauge, Ø., Sørensen, C.-F., Røsdal, A.: Surveying Industrial Roles in Open Source Software Development. In: Int'l Conf. on Open Source Systems, pp. 259–264 (2007)

[7] Ruffin, C., Ebert, C.: Using open source software in product development: A primer. IEEE Software 21(1), 82–86 (2004)

[8] Stol, K., Ali Babar, M.: Challenges in Using Open Source Software in Product Development: A Review of the Literature. In: 3rd FLOSS Workshop, ICSE 2010, pp. 17–22 (2010)

[9] Dong, J., Zhao, Y., Peng, T.: Architecture and Design Pattern Discovery Techniques - A Review. In: Int. Conf. Softw. Eng. Research and Practice, pp. 621–627 (2007)

[10] Tonella, P., Torchiano, M., du Bois, B., Tarja, S.: Empirical studies in reverse engineering: state of the art and future trends. Empir. Software Eng. 12(5) (2007)

[11] Stol, K., Avgeriou, P., Ali Babar, M.: Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges. In: EASE, Keele, UK (2010)

[12] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discpline. Prentice-Hall Inc., Englewood Cliffs (1996)

[13] Harrison, N.B., Avgeriou, P.: Leveraging Architecture Patterns to Satisfy Quality Attributes. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 263–270. Springer, Heidelberg (2007)

[14] Stol, K., Ali Babar, M.: A Comparison Framework for Open Source Software Evaluation Methods. In: Int'l Conf. on Open Source Systems, pp. 389–394 (2010)

[15] Fitzgerald, B.: The transformation of open source software. MISQ 30(3) (2006)

[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

[17] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. Trans. Softw. Eng. 26(1), 70–93 (2000)

[18] Seaman, C.B.: Qualitative methods in empirical studies of software engineering. Trans. Softw. Eng. 25(4), 557–572 (1999)

[19] White, S.A.: Introduction to BPMN, BPTrends (July 2004)

[20] van Heesch, U.: http://www.cs.rug.nl/search/ArchPatn/OpenPatternRepository

[21] Booch, G.: http://www.handbookofsoftwarearchitecture.com (accessed December 5, 2010)

[22] Falessi, D., Ali Babar, M., Cantone, G., Kruchten, P.: Applying empirical software engineering to software architecture: challenges and lessons learned. Empir. Software Eng. 15(3), 250–276 (2010)

[23] Wohlin, C., Höst, M., Henningsson, K.: Empirical Methods and Studies in Software Engineering. LNCS, pp. 145–165 (2008)

[24] Jedlitschka, A., Pfahl, D.: Reporting Guidelines for Controlled Experiments in Software Engineering. In: ISESE, pp. 95–104 (2005)

[25] Olson, D.L., Delen, D.: Advanced Data Mining Techniques. Springer, Heidelberg (2008)

[26] Carver, J.C., Jaccheri, L., Morasca, S., Shull, F.: A checklist for integrating student empirical studies with research and teaching goals. Empir. Software Eng. 15(1) (2010)

[27] Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: A Systematic Review of Quasi-Experiments in Software Engineering. Inf. Softw. Technol. 51(1), 71–82 (2007)

[28] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., el Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. Trans. Softw. Eng. 28(8), 721–734 (2002)

[29] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic, Dordrecht (2000)

[30] Liu, J.: Research Project: An Analysis of JBoss Architecture, http://www.huihoo.org/jboss/jboss.html (accessed March 2, 2011)

[31] Salehie, M., Li, S., Tahvildari, L.: 'Architectural Recovery of JBoss Application Server', Tech. Report no. UW-E&CE#2005-02, University of Waterloo (2005), http://stargroup.uwaterloo.ca/~s7li/publications/ieee_papers/uw-tr-1.pdf

[32] Hollander, M., Wolfe, D.A.: Nonparametric statistical methods, 2nd edn. John Wiley & Sons, Inc., Chichester (1999)

[33] Höst, M., Regnell, B., Wohlin, C.: Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. Empir. Software Eng. 5(3), 201–214 (2000)

[34] Svahnberg, M., Aurum, A.K., Wohlin, C.: Using Students as Subjects – An Empirical Evaluation. In: ESEM, Kaiserslautern, Germany, pp. 288–290 (2008)

[35] Berander, P.: Using students as subjects in requirements prioritization. In: ISESE (2004)

Autonomic Computing Driven by Feature Models and Architecture in FamiWare

Nadia Gamez, Lidia Fuentes, and Miguel A. Aragüez

Dpto de Lenguajes y Ciencias de la Comunicación, Universidad de Málaga
{nadia,lff,m_a_a_r}@lcc.uma.es

Abstract. A wireless sensor network is an example of a system that should be able to adapt its sensor nodes to some context changes with minimum human intervention. This means that the architecture of the middleware for sensors must encapsulate a dynamic mechanism to allow reconfiguration. We present a novel approach to achieve self-adaptation based on software product lines and on the autonomic computing paradigm for the FamiWare middleware. FamiWare uses feature models to represent the potential middleware configurations at runtime. Each configuration is automatically mapped to the corresponding architectural representation of a specific middleware product. Following the autonomic computing principles, FamiWare defines a reconfiguration mechanism that switches from one architectural configuration to another by means of executing a plan. This is possible thanks to the loosely coupled architecture of FamiWare based on an event-based publish and subscribe mechanism. We evaluate our work by showing that the resource consumption and the overhead are not so critical compared with the benefits of providing this self-adaptation mechanism.

Keywords: Autonomic Computing, Middleware, Feature Models, Product Lines Architectures, Event-based Architectures, Models at Runtime.

1 Introduction

Wireless Sensor Networks (WSNs) [1] are currently attracting huge interest due to their potential of applicability in a variety of ubiquitous and ambient intelligence systems. These networks employ from a few sensors to hundreds and thousands of them, linked by a wireless medium, and are able to perform distributed sensing tasks for the purpose of monitoring certain physical phenomena, indoors or in remote environments. Currently, WSNs can support many novel and exciting applications in a wide range of fields such as, smart spaces, traffic control, habitat monitoring or ambient assisted living applications, being considered one of the top technologies that are changing the world.

From the point of view of engineering such systems, the development and deployment of WSNs applications can be considered a very complex task, since such networks pose several distinct requirements, comparing with traditional information systems. Some of these requirements are the critical resource limitations of nodes, the

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 164–179, 2011.

heterogeneity of nodes, the managing of a high variety of routing protocols between the nodes and the low-level programming abstractions normally used. One of the alternatives being proposed recently to face such requirements is the development of a middleware, specific for WSNs. Concretely in our previous work, considering the high variety of hardware and software available, we have defined a *family of middleware* for Ambient Intelligence (AmI) systems, including WSNs (FamiWare [2]), instead of developing a single middleware, normally very specific of a technology or property [3]. We have used the Software Product Line (SPL) [4] approach, to characterize the inherent variability of the WSNs domain by SPL feature models (FM) [5]. In order to facilitate the easy customization and configuration of the middleware Product Line Architecture (PLA), the composition among middleware services themselves and between the services and the final application is performed with a Publish/Subscribe event-based mechanism [2]. We have also defined a mapping between the features and the components and the PLA. These FM and PLA are the base of a model driven process that derives a minimum configuration (i.e. architecture with less numbers of components) adapted to the requirements of each network node, especially those referring to sensor resources, so scarce in WSNs. The input of this process is the list of specific features of a WSN (e.g. number and kind of nodes or routing protocols) and the output is the minimum FamiWare configuration including the service implementations adapted for each node.

However, once the middleware for a WSN is deployed, it normally requires some self-adaptation to context changes. Specifically, a WSN suffers the symptoms of degradation, such as energy loss or failure of some nodes, which requires explicit management action. But considering WSNs are usually deployed in remote unattended or inaccessible areas, the sensor nodes should be able to recover from degradation with minimum human intervention. Therefore, the mechanisms proposed for making some adaptation in response to system degradation or to any context change must avoid the remote administration by technicians and especially must consider the saving of energy to guarantee the system survival. Considering this, such middleware should be as autonomic as possible. In this paper, we present a novel mechanism to achieve self-managing in FamiWare, which applies the Autonomic Computing (AC) paradigm [6] at middleware level. An autonomic middleware should provide mechanisms for automatic reconfiguration of services (self-configuration), automatic detection and correction of faults (self-healing) and automatic configuration of optimal resource parameters according to context variations (self-optimization).

At design time, FamiWare uses feature models and the middleware architectural model to derive middleware configurations. What we propose here, is to use these models also at runtime to drive the reconfiguration of the middleware for failure recovery and/or self-adaptation. So, we use a models@runtime approach, to interpret a plan with the necessary actions that must be executed to move from a configuration of the middleware to a new configuration in response to context changes. The main advantage of implementing the AC by using models@runtime is that our process can ensure that the architectural configurations running in every network node, and at any moment, are correct according to the global network restrictions. By applying AC and models@runtime at the middleware level, our approach promotes an efficient utilization of network resources, which are shared by many applications. Other approaches implement self-managing at code level [7], but the main drawback is that everything is hardcoded, so a correspondence between the reconfigured code and the models used at design time is lost; not being possible to assure that every middleware instance will enter a valid state after a reconfiguration is executed in all nodes.

Finally, we evaluate our work by showing mainly that the overhead is not so critical comparing with the benefits of providing an autonomic behaviour in WSNs.

The remainder of the paper is organized as follows. In Section 2, we discuss the WSNs challenges and how we achieve them. In Section 3, the FamiWare autonomic computing process is described. Furthermore, in this section, we detail the plans and how the dynamic reconfiguration changes are performed in the middleware architecture. The evaluation of our approach is presented in Section 4. In Section 5, we compare our approach with related works. Finally, in Section 6 we outline some conclusions and future works.

2 Motivation

Autonomic Computing was first proposed by IBM in 2001, and its main goal is to endow distributed systems with self-management capacities. In this section we will first discuss the special challenges of WSNs that made them good candidates to take advantage of AC. Finally, we will present our process to derive valid middleware architectural configurations using an SPL approach, and how we reuse the design models also at runtime to implement AC for WSNs.

2.1 Autonomic Computing for Wireless Sensor Networks

We consider that the peculiarities of WSNs made them very appropriate to take advantage of the AC paradigm. This is because, as we already explained in the introduction, WSNs are usually deployed in dynamic and hostile environments with no human presence. So, they must be tolerant to the malfunctioning and loss of connectivity of the nodes and other context changes and furthermore, they must be energy efficient. The goal of AC in WSNs is to achieve an autonomic reconfiguration of nodes in order to have a more optimal network functioning. Now, we detail the specific **challenges** that made the WSNs so suitable to applying the AC paradigm. Our goal is to cover all these challenges using our event-based architecture middleware family together with FMs and models@runtime to put into practice AC.

- **C1 Self-management:** These systems must be able to heal failures and dynamically optimize the functioning of the system in a power efficient way, freeing human administrators from low-level management tasks. WSNs require implement a *self-management* mechanism to guarantee the survival of the application over time.
- **C2 Middleware:** Sometimes a given WSN gives support to many applications. For example, the sensor network installed in the junctions of a city to control the urban traffic can support different control applications (as collect traffic statistics or detect accidents). Consequently, the *self-management* mechanism should be implemented at the middleware level, so that all the applications will benefit from the *self-ities.* Furthermore, the architecture of the middleware has to be the most flexible as possible to allow customization and reconfiguration.

- **C3 Models:** Several WSNs works [7,8] already implement some autonomic behaviour but normally each of them is more focused on providing a specific reconfiguration (e.g. about routing protocol [8]). Also, most of them use rudimentary frameworks where an expert in low level WSNs programming must define the reconfiguration process at code level. However, it would be better that this process can be specified and verified at design level by a domain expert modeller. So, the correspondence between the model of the architectural design and the code installed in every moment is guaranteed.
- **C4 Compatibility:** In these networks, the architectural configurations installed in every node must be compatible. For example, the routing protocol used by a sensor node must be the same as the one used by the sink node. So, the reconfiguration process must ensure that new configurations of different kinds of nodes are compatible.
- **C5 Context variability:** In WSNs the variability of the context situations that triggers a reconfiguration is very high. But after revising real use cases found in the literature and after doing some experiments, we conclude that this variability is bounded. This is because reconfiguration can only be caused by changes inside nodes or in network connections. So, a challenge to make the autonomic mechanism usable and effective is to characterise the context changes as a *bounded set*. This help to define a mapping from context changes to a list of reconfiguration plans.
- **C6 Reconfiguration changes:** Often optimizing the network functioning entails the execution of basic operations in the nodes, such as reducing the monitoring frequency. Also, more complex operations are required such as changing the routing protocol. So, particular attention to fine-grained modifications (e.g. modification of components parameters) must be considered, although coarse-grained ones must also be part of the autonomic mechanism (e.g. adding or removing components).

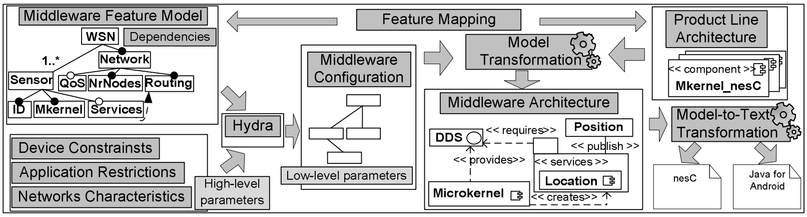


Fig. 1. FamiWare Configuration Process

2.2 Feature Models and Event-Based Architecture for Autonomic Computing

The goal of FamiWare is to cope with the inherent variability of AmI systems, and in particular that of WSNs. We define an automatic process to derive different middleware configurations depending on the hardware and software of the deployed WSN. We apply model-driven and SPL engineering techniques to automate this configuration process. Fig. 1 gives a general overview of it (a complete version can be found in [9]). The first step when creating a SPL is to analyze the variability inherent in the AmI middleware domain. For this task, a *feature model* [5] is constructed. A FM specifies which elements, or *features*, of the family of products are common and which are variable. In addition, it is possible to specify formal constraints or dependencies between these elements. In the next step the PLA of the middleware, which contains both the commonalities and the variabilities specified by the FM is defined. A *Feature Mapping* between the FM and the PLA defines the correspondence between features and middleware components. This correspondence is largely far from being trivial, since each feature is often designed by using more than one component, depending either on the implementation strategies of the service, or on the devices development technology [9]. The customization of the middleware architecture is determined by a set of high-level parameters (e.g. number of sensors or the necessary services). The goal of our process is to reduce the number of high-level parameters the user needs to specify in order to customise a product, and this is achieved by exploiting the *dependencies* between features. This means that the user modeller only has to provide a minimum set of high-level parameters as input features and the rest are automatically inferred by the process using a constraint solver provided by Hydra [10] (our feature modelling tool). A larger set of low-level parameters (e.g. services implementation for TinyOS version) is automatically obtained. Then, our process calculates the particular middleware architecture, by means of model transformations using the feature mapping. This customised architectural model is the input of a model-to-text transformation, which produces 100% of the code for deploying this middleware into the devices.

FamiWare follows a microkernel plus services architecture. The composition among services themselves and between the services and the application is performed with a Publish/Subscribe event-based mechanism, specifically a reduced implementation of the Data Distribution Service (DDS) [12] interfaces. This composition mechanism decouples the services, so it is easier to generate a configuration with variable services and to reconfigure them [9]. In Fig. 2, we show a partial architecture of FamiWare containing some services related with the AC process. The microkernel provides the DDS interface to allow application and services subscribing or publishing events (called *topics*). The microkernel also encapsulates the *Data Delivery* service, which sends data to local and remote components. The microkernel interface is required by both: the application, which subscribes to some topics according to its functionality, and the services that publish and subscribe topics. For example, the service that monitors the battery (*BatteryMon*) publishes the battery level and the *Context-Awareness* service is subscriber to it. For a detailed description of FamiWare architecture we refer the interested reader to [2].

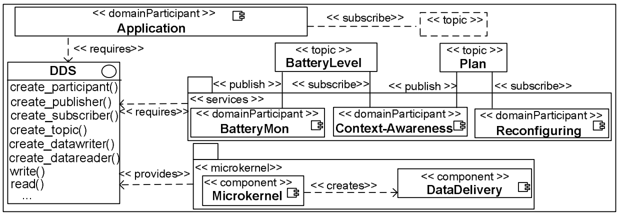


Fig. 2. FamiWare Partial Architecture

The features considered to derive an initial middleware configuration for each network node, are known at design time (e.g. number and location of nodes). The aim of our current work is to generate dynamically different configurations, according to the variable conditions raised at runtime, to achieve autonomic behaviour of our middleware. We address **C1** and **C2**, since we tackle the WSNs *self-management* developing an autonomic middleware family that follows a Publish/Subscribe architecture. Likewise our design process, we propose to model the set of environmental elements that may change during runtime also with an FM and the correspondence between the features and the architectural elements (fulfil **C3**). The rationale behind this is that the set of possible changes that may affect the functioning of a WSN can be seen as a set of variable features that may trigger an autonomic reconfiguration. Note that as we discussed in **C5** the cardinality of this set is bounded. For Hydra at design time, a valid configuration corresponds to a set of features of the FM that satisfies the tree constraints (as mandatory, optional, alternative OR and alternative XOR) and the cross-tree constraints expressed by the dependencies between features. Therefore, we introduce the concept of valid configurations at runtime, where they are generated if one or more features change during middleware execution. The base of our models@runtime mechanism is that the set of actions that must be executed to pass from one valid configuration to another is specified in a plan model compliant with feature and architectural models. We would like to highlight that since we use models@runtime formally verified with Hydra, we can assure, as we mentioned in **C3**, that the middleware architectural configurations running at any given moment in all the network nodes are correct and work properly. Furthermore, we extended Hydra with *clonable features* [11] and we model each node sensor as a clonable feature, where the cloning of these features leads to the cloning of the related structure. Them, the dependencies between the clones guarantee that the configurations of all the nodes of a certain network are compatible, as is required by **C4**. Also, as part of the feature model we specify the elements of the middleware architecture that can be changed after a context change achieving **C6**.

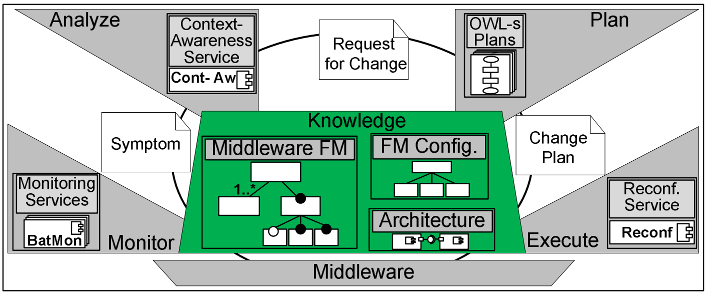


Fig. 3. FamiWare Autonomic Computing Process

3 FamiWare Autonomic Computing Process

AC relies on four functions that share *knowledge*: observes environment details (*monitor*); analyzes them to determine if something needs to change (*analyze*); creates a plan that specifies the necessary changes (*plan*); and performs these actions (*execute*) [6]. But, how these functions are implemented is not a trivial task and is not defined anywhere. We discuss how the realization of the AC functions can be mapped very naturally and in a straightforward manner for the domain of WSN. We have adapted the architecture presented in [6] to our models@runtime AC process, as is shown in Fig. 3. The *monitor*, *analyze* and *execute* functions are implemented as services provided by our middleware (see Section 3.2). Our *plans* are models that contain the tasks to be performed to change from a configuration to another (see Section 3.3). The *knowledge* needed by AC functions is determined by how the reconfiguration is driven for WSNs. We use FM and their corresponding architectural configurations (see Section 3.1).

3.1 Knowledge: Feature Models and Architectural Configurations

After network deployment, a valid configuration, compliant with the FM, is running in each node. Since AC mechanism must define how to go from one configuration to another one, the AC functions must share *knowledge* about the FM and the current architectural middleware configuration. Fig. 4 shows some features of our FM that represent the variability found in two services provided by FamiWare middleware: monitoring and context-awareness. We use FMs to express variability of nodes known at design time and also the variability of context elements, only used at runtime for network reconfigurations. So, we distinguish two levels in our FM:

- **FM at design time (Fig. 4, top):** It mainly models all the possible device characteristics (e.g. OS or radio technologies), network characteristics (e.g. routing protocol) and middleware services (e.g. data fusion or context-awareness). As part of the FM, we define a set of dependencies between features. We can distinguish two types of dependencies: (1) *local*, which are defined between features modelling characteristics of one node (e.g. if coordination service is selected the node must be a cluster-head (CH) or a sink); (2) *network*, which defines dependencies between cloned features (e.g. node sensors), to assure the compatibility between the distributed configurations (e.g. all the nodes have to use the same routing protocol).
- **FM of the context (Fig. 4, down):** We extended the original FM with new features that model all the possible context changes considered by our AC functions. In this FM, we model: (1) for each service the elements of which may be changed at runtime (e.g. frequency of reading the battery); (2) parameters of features already defined at design time, but that can be changed at runtime (e.g. the aggregation function used by the data fusion service); (3) the set of plans and their task and parameters. Also, new *context dependencies* between these features are defined.

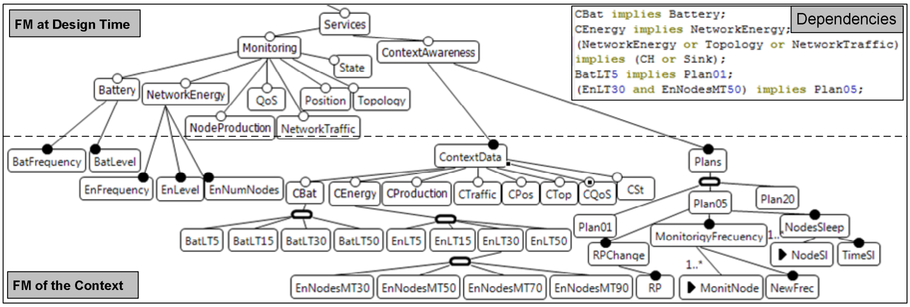


Fig. 4. Partial Feature Model of two FamiWare Services

In order to obtain the subsequent valid configurations generated at runtime, Hydra considers the three types of dependencies (local, network and context).

3.2 Autonomic Computing Services in FamiWare

***Monitor*: Monitoring Services.** FamiWare provides several monitoring services that observe the possible element of the context that may change. We have reviewed the literature on WSNs applications the conclusion was that all of them consider similar context elements to perform dynamic adaptation:

- *Battery*: the level of a sensor node battery.
- *Network Energy*: the energy map of an entire network or of a group of sensors.
- *Node Production*: the amount of data sent by a node to other nodes of the network.
- *Network Traffic*: the amount of traffic in the network.
- *Position*: the position of a node in a network (GPS or relative positioning algorithms).
- *Topology*: the position of all the nodes of the network or of a group of sensors.
- *QoS*: the QoS required by the application.
- *Nodes State*: the state of a node (alive, dead, idle, slept, etc.).

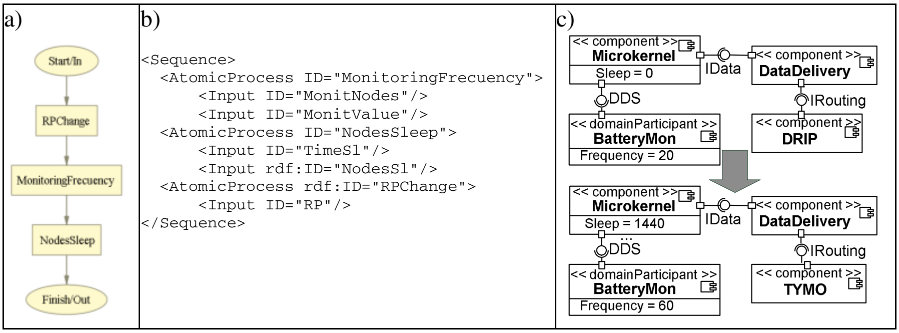
These monitoring services are represented as optional features in the FM (see Fig. 4) corresponding to components in the PLA, so different configurations could include different monitoring services. Concretely, those that collect data from some network nodes are only instantiated in CH or sinks, as is specified in the dependencies of Fig. 4. Each monitoring service feature has several sub-features relevant for our AC process. That is, the attributes that may *trigger* a change and the *parameters* that may change at runtime. For example, the monitoring of the battery is defined by the current level *BatLevel* (*trigger*) and by the frequency *BatFrequency* (*parameter*). This is represented as component parameters in the PLA.

***Analyze*: Context-Awareness Service.** This service is responsible for the awareness of the context, using the data provided by the monitoring services, and initiates the reconfiguration process as a consequence of a context change. It is also defined as part of the FM, since the list of monitored data used to be aware of the context is also variable. For example, in a sensor configuration, it could be useful to consider all the monitoring services observing the context, but in other configurations only the data about the battery is to perform a reconfiguration. Fig. 4 shows that this service is defined by the list of context data appropriate for each node (optional) and the plan that must be chosen by this service when the context changes, both represented as component parameters in the architecture. We define a child feature for each context data, which specifies simple logical expressions that may trigger a context change. For example, the *Context-Aware* service installed in a sink node, is receiving values of energy of the other nodes from the energy network monitoring service, but only when it detects a battery level less than 30% (*EnLT30*) in more than 50% of the nodes (*NodeEnMT50*), it chooses the corresponding plan. The available plans are modelled by a XOR-alternative group in the FM. Also, dependencies between the state of a node (formed by the list of current values of context attributes, e.g. energy_level=28%; number_nodes=51%) and the plan to choose are defined (as is shown in dependencies of Fig. 4, the correct plan for this example will be *Plan05*).

***Reconfigure*: Reconfiguring Service.** This service interprets a plan and executes the tasks to reconfigure the system. Each task implies performing hardware or software modifications, for instance modify the attribute of a software component. Some of these tasks can be more complex, requiring the execution of actions in several nodes. For example, at hardware level, for putting to sleep a set of nodes, the sink or CH sends an event with a request to turn on the sleep mode some nodes for a certain period of time (given as input parameters of the plan). At code level, this is implemented by means of turning off the radio of the node. Also, some software changes can be considered complex, as changing the routing protocol, consists of uploading a new image of code in every node.

3.3 Plan: Models@Runtime

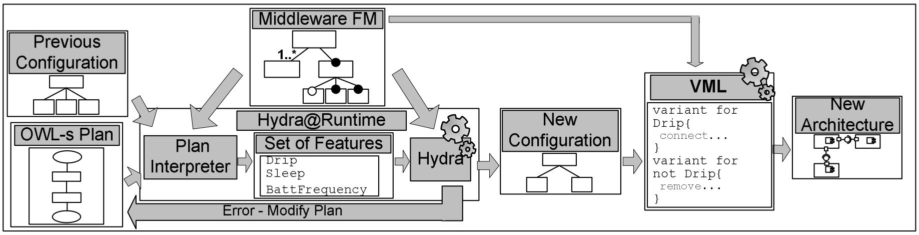
In FamiWare a *Plan*, which consists of a list of tasks, represents the difference between two FM configurations, defined in terms of the *FM of the context*. A *Plan* is specified in OWL-S, an ontology of services for semantic description of Web services. Fig. 5 shows a reconfiguration plan for a sink or a CH to recover from a loss of energy detected in some nodes. Fig. 5.a depicts a graphical representation of the plan generated using the OWL-S editor of Protégé and Fig. 5.b shows a simple extract of the corresponding owl file (our process removes tags and generates a more efficient reduced version for being loaded in sensors). A plan is modelled as a composite process that defines an ordered list of tasks (or atomic process in OWL-S terminology). In our example, three tasks are defined: (1) change the routing protocol; (2) reduce the monitoring frequency; (3) and finally, put some nodes in a sleep mode. As is shown in Fig. 5.b the tasks may have input parameters, as in the case of the *NodesSleep* task, the list of the nodes to sleep and the period of time to be in sleep mode. These parameters are also represented as part of the *FM of the context*, and are provided by the Context-Awareness service as part of the elected plan to the Reconfiguring service. Fig. 5.c, depicts the partial Famiware architecture for a sensor node before and after reconfiguration, showing only the features that were modified by the plan. This example illustrates the two kinds of changes: the coarse-grained modifications, as the addition of the **TYMO** routing protocol component and the removal of the **DRIP** component; and the fine-grained modifications as the change in the values of the **Sleep** period and **Frequency** parameters in two components.



```
<Sequence>
  <AtomicProcess ID="MonitoringFrecuency">
      <Input ID="MonitNodes"/>
      <Input ID="MonitValue"/>
  <AtomicProcess ID="NodesSleep">
      <Input ID="TimeSl"/>
      <Input rdf:ID="NodesSl"/>
  <AtomicProcess rdf:ID="RPChange">
      <Input ID="RP"/>
</Sequence>
```

Fig. 5. Plan example in OWL-S and Reconfiguration of the architecture

We have defined a plan for every possible reconfiguration that we allow in our middleware but new plans for other necessary reconfigurations can be defined. After the definition, using our tool Hydra@Runtime we can check if a applying this plan to the current valid configuration, the new configuration generated is also valid. So, before loading a plan in the middleware the domain modeller can check the correctness of this plan for the specific architectural current configuration as follows (see Fig. 6). Firstly, the OWL-S plan is interpreted taking as input the valid previous middleware configuration and the FM and it calculates the set of features that must be selected and unselected in the new configuration. These features are used together with the FM, to try to construct a valid new configuration. But if some dependencies between features cannot be satisfied for this set of features, an error is reported with the features that produced such error, and the plan must be modified to fix these errors. If the plan works well a new FM configuration is automatically generated. Using this new configuration, the FM and the *mapping* between the FM and the PLA as inputs, the new architectural design is also automatically generated. In this way a correspondence between the models and the code is always maintained.



```
variant for
Drip{
 connect...
}
variant for
not Drip{
 remove...
}
```

Fig. 6. Plans Checking and Generation of the New Middleware Architecture

4 Evaluation

As is described before, FamiWare has microkernel plus services architecture, where the microkernel provides the implementation of the DDS interfaces. The implementation of the microkernel and the AC services has been realized in TinyOS 2.1.1. and the experiments have been performed in TOSSIM and PowerTOSSIM z simulators, and also in real MICAz motes.

4.1 Overhead of the Reconfiguration

The goal of the experiments presented is threefold: (1) they quantify the memory footprint used by FamiWare (including the AC services); (2) they calculate the overhead produced for our middleware, measuring the latency since a context change is detected and the reconfiguration is accomplished; (3) and they evaluate the scalability of FamiWare, regarding the number of services and plans.

Resource Consumption. Since the architecture of FamiWare consists of a microkernel and a variable set of services, the memory usage on a device depends on how many services are in the current configuration and on the size of each one. The minimal core architecture consists of the microkernel and the data delivery service, which includes a routing protocol. Also, since any TinyOS application is compiled together with the OS in one image, we also have to include the OS as part of our measures. Specifically, to perform the autonomic behaviour, the minimal instantiation must have one monitoring service, the context-awareness service and the reconfiguration service. So, for this case, using Drip [13] routing protocol, the memory footprint is very reduced. Considering that typically MICAz nodes with 4Kb of RAM, the FamiWare version for AC consumes the 54.4% (2229 bytes) of RAM. On the other hand, the maximum architectural configuration for AC, considering all monitoring services (8), the microkernel, Drip protocol and a simple application (123 bytes), uses 79.9% (3274 bytes) of the memory, so there is still 20.1% free for other services provided by FamiWare. Note that this maximum configuration is only instantiated in a sink or in a CH, special nodes that normally have many monitoring services to monitor other nodes of the network. For these nodes, device sensors with higher capacities are usually used, so the memory constraints in these devices are not as strict as in ordinary nodes, since normally have much more memory. Summarizing, the resource consumption of FamiWare is satisfactory and very well suited to the memory constraint of real sensors.

Table 1. Latency for the Reconfiguration of the System

Operation	TinyOS
Receive monitored data	1.86 ms
Detect a context change and choose the plan	0.95 ms
Total Context-Awareness time	2.81 ms
Receive plan chosen	0.519 ms
Get and read the plan file	9.926 ms
Interpret and execute the tasks of the plan	565.725 ms
Total Reconfiguration time	576.17 ms

Overhead and Latency. We measure the interval time from which the context-awareness service detects a context change until the reconfiguration is completed. This interval time encompasses: the context-awareness service receives the monitored data, analyzes them for detecting a context change, chooses the suitable plan, and finally the reconfiguration service receives and executes the plan. Table 1 reports these results. The total time of the context-awareness service is 2.8ms, an insignificant time compared with the time taken by the reconfiguration service until the system is reconfigured: 576.10ms. As is observed in the table, the longest time in the reconfiguration service corresponds to the execution of the tasks that comprise the plan. These results are taken for the plan detailed in the previous section, installed in a sink node of a 20 nodes network. Remember, this plan has three tasks and all of them implies the reconfiguration of several remote nodes. So, this entails to send several events to the nodes and the time this takes depends on the routing protocol used. For instance, using Drip [13] protocol in one hop (without intermediate nodes) it takes 8ms but if we consider 3 intermediate nodes, this time increases up to 6.43s. It's worth noting that if we increase the number of sensor nodes, but they are still at one hop of the sink, the time for reconfiguring them does not increase significantly. Nevertheless, if the sink has to use some intermediate nodes to reconfigure other nodes, the time greatly increases, but it depends on the structure of the network and the routing protocol used, as in other WSN platforms. Then, the overhead produced to reconfigure the system is insignificant comparing with the necessary time to send a packet. These times were calculated considering a few plans installed in the sink, but for a higher number of plans, an overhead is produced to search and get the plan from the FLASH memory, as we show in the scalability evaluation. From the user point of view, total time to detect changes and to reconfigure the system is very satisfactory.

Scalability. We assess the scalability performance in terms of the number of plans that an ordinary node can load in its FLASH memory. MICAz motes have 512KB of FLASH. In Table 2 is shown the structure of the FLASH for a sink node with our middleware installed. DYMODATA is the space to save data about the routing protocol. GOLDENIMAGE and the DELUGEs are partitions used by the Deluge TinyOS system. Deluge is a system for reprogramming sensor nodes, so it uses these volumes to load the binary code of the new images. We have reserved 98Kb for loading the plans and still there are 30Kb free, enough for applications or sensed data. The size average for the plans (after Hydra removes the unnecessary tags) is about 500 bytes, so there is enough space to load 200 plans, this is the upper limit of the number of plans. On the other hand, in an ordinary node (not sink or CH) we can reduce the space reserved for plans, since these nodes do not need this high number of plans loaded in memory. However, the time to search a plan in the FLASH memory increases when there are numerous plans loaded. As depicts Fig. 7, the increasing of time is lineal respect to the number of plans. Finally, the maximum time needed to find a plan, if we consider 200 plans loaded is 1298ms, being an insignificant delay, comparing with other sensor tasks. We can conclude that FamiWare works very well, although the number of plans augments considerably.

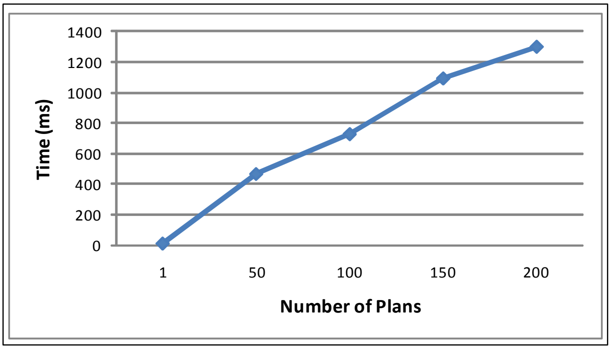


Fig. 7. Evolution of the Time to Read a Plan

Table 2. FLASH memory of MICAz

Volume	Size
DYMODATA	128Kb
GOLDENIMAGE	64Kb
DELUGE1 + DELUGE2 + DELUGE3	192Kb
Plans	**98Kb**
Free	30Kb

4.2 Benefits of the Reconfiguration

In this section we will show the benefits of the reconfiguration in terms of energy saving, one of the goals of our work. The battery capacity of MICAz motes is 2000mAh and lasts about one year. We have simulated the reconfiguration of a 20 nodes network using the plan explained previously, where the energy falls to 30% (600 mAh) in 50% of the nodes and we have to compare the energy expenditure of the same system but without reconfiguration. Fig. 8 shows how the average of the network residual energy starts to decrease when the reconfiguration is being performed (from 0 to 1000 seconds simulation interval). This is due to the extra cost of operations done for reconfiguring the nodes and the sink. For instance, they must be in *Active* state with the radio chip *ON* and they must access their *FLASH* memory. During the reconfiguration, the remaining energy average drops to 599,84mAh for the first 1000 seconds simulated, while in the system without reconfiguration it only drops to 599,911mAh. There is a difference of 0,071mAh, that is the price of reconfiguration in terms of energy, but after it finishes, the energy expenditure is less for the reconfigured system than for the static one. Particularly, the remaining energy of the reconfigured system overcomes the energy of the static one before 3000 seconds after the reconfiguration begins. The difference between the two systems is that the reconfigured one expends about 0.032mAh less than the other in each 1000 second interval. This is due to, after executing our plan, more nodes will be in sleep mode (with their radio chip *OFF*), the monitoring task is performed with less frequency than before (some nodes can be more time in *Idle* or *Power-save* states) and the routing protocol used is more energy saving.

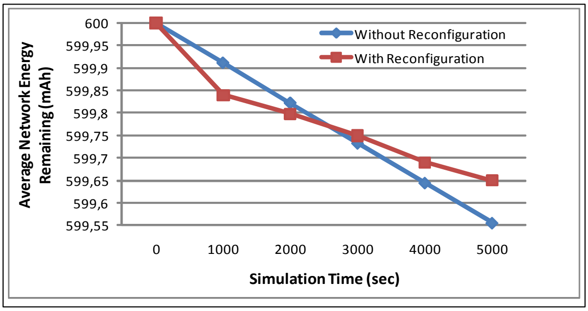


Fig. 8. Energy Expenditure for a System with and without Reconfiguration

The figure only shows some simulation intervals, but if we consider the entire lifetime of the system, the one without reconfiguration will live about 78 days after the network energy drops to 30%, and the reconfigured one, will extend it lifetime in 121 days. So, our reconfiguration allows the network to increase its lifetime, 55 % more than the static system (and about 16% of its total life). Obviously, after this particular reconfiguration, the sensed data could be less accurate, since we increase the period of time for the sensing tasks and several sensors will be periodically slept. But for many systems, it will be much more important to increase their lifespan instead of receiving extremely accurate data. So, we can conclude that the AC behaviour of FamiWare is very useful for satisfying the particular necessities of WSNs such as energy saving. Of course, we define distinct plans for satisfying other network goals.

5 Related Works

In this section we compare our work on autonomic middleware for WSNs with other relevant approaches in Autonomic Computing.

Autonomic WSNs Middlewares. There are a number of existing sensor middleware [3], but only few of them consider an autonomic behaviour. Impala [14] is a middleware that enables application adaptability and reparability in WSNs. But it only explores the reconfigurability of the routing protocol by replacing the entire service. Instead, FamiWare supports also fine-grained modifications with great impact in WSNs (**C6**). Also, in Impala the adaptation is only based on the local state of individual nodes. Finally, they implement a prototype on iPAQ Pocket PC, but not in real sensors as we do. GridKit [15] is a reflective middleware for co-ordinated dynamic reconfiguration of middleware behaviour across nodes in WSNs. As well as FamiWare, it allows dynamic reconfiguration on both, local node and network. But, they do not define what changes they allow and the correspondence between design models and the running code, as our plans and FM do (**C3**). Their implementation is based on high capacity sensors (with Java and Linux), which are not used very often. We tested our AC implementation in MICAz sensors, with more restrictive constraints, so we demonstrated that our approach is feasible. In [16] the benefits of autonomic and semantic WSNs are combined to build a semantic middleware for autonomic WSNs that provides support for Structural Health Monitoring (SHM) applications. They represent the knowledge of AC using ontologies adapted to SHM and we use FMs applicable to any WSN application. Furthermore, they propose a planning service that generates plans, instead our models@runtime approach. WiSeKit [17] is a distributed component-based middleware approach that enables adaptation and reconfiguration of WSN applications. As FamiWare, it adopts dynamic parameter and component adaptation and follows a situation-action rules approach as our context-awareness services. However, their reconfiguration process is driven by code instead that our driven by model process (**C3**).

Autonomic and Adaptative Middlewares. Outside the domain of sensor networks there are a set of approaches that propose autonomic middlewares. Amun [18] is an autonomic middleware for ubiquitous environments. As FamiWare, it is focused on self-configuration, self-healing and self-optimization. They focus, however, only on one kind of system, the Smart Doorplate Project. Furthermore, they do not specify which kind of reconfigurations they allow and how they are performed. ADAMANT [19] is a middleware that uses supervised machine learning to autonomously configure cloud environments with transport protocol. However, they provide only an autonomic configuration of the middleware but they do not consider the autonomic reconfiguration, as is proposed by AC and as we do. DySCAS [20] is a middleware that facilitates context-aware dynamic reconfiguration of automotive control systems. They define *awareness of execution context* and *awareness of architecture context* concepts that are equivalent with the functions performed by our monitoring services and with the FM and the architectural configurations that compound our AC knowledge, respectively. Nevertheless, they use meta-data to embed the knowledge instead of models as we do.

SPLs approaches for Autonomic Computing. There are other proposals that find it useful to use SPLs technologies, as Feature Models to provide autonomic behaviour, but they provide heavyweight solutions for general purpose middleware or applications, not applicable to resource constrained WSNs. In [21] the application of model-driven development and middleware to support the development and operation of dynamically adaptive systems is presented. As does our work, they argue that dynamic systems can be considered as a product family line in which variabilities are bound at runtime instead of at pre-delivery time. Nevertheless, they do not provide a complete implementation of an adaptive middleware as FamiWare. As we argue throughout this paper, in [22] it is suggested that autonomic behaviour can be achieved by leveraging variability models at runtime. Also, in [23] a feature-oriented approach to dynamically develop reconfigurable core assets is proposed. Both of them use FM to guide the reconfiguration process as we do. In contrast to our middleware level approach, however, both of them focus on specific applications.

6 Conclusion and Future Works

We have shown that WSNs have several particular challenges since they are deployed in dynamic environments without human presence. So, these systems should be as autonomic as possible to recover from failures or to reconfigure themselves. To address these challenges this paper has presented the AC behaviour of FamiWare (**C1**). We propose to use FM, the architectural design and plan models at runtime to drive the reconfiguration of the middleware. The main benefits are: The **reconfiguration is tackled at middleware level**, avoiding that every application has to implement the AC process (**C2**); Using **models@runtime** we can assure the correspondence between the model and the code installed in every node and that it works properly, because only valid configurations may be installed (**C3**); Our process ensures that the **new architectural configurations** obtained after the reconfiguration process are **compatible** in all network nodes (**C4**); The variability of context changes is high but limited, so the number of valid configurations is finite. So, using our **feature models and valid configurations** we consider all the possible changes (**C5**); We consider both **fine and coarse-grained adaptations**, applying each depending on the context change (**C6**). Finally, the experimental results show that our process is both feasible and useful.

We are working on the extension of our Hydra@Runtime tool to allow that new plans can be automatically generated, giving two valid configurations: the previous configuration and the new configuration. This Plan Generator will take these configurations and it will find the differences between them and generates a new plan with the tasks that must be performed to go from the previous to the new configuration.

Acknowledgement. This work has been supported by the Spanish Ministry Project RAP TIN2008-01942 and the Junta de Andalucía regional project FamWare P09-TIC-5231.

References

1. Akyildiz, I., Kasimoglu, I.: Wireless Sensor and Actor Networks: Research Challenges. Ad Hoc Networks Journal 2(4), 351–367 (2004)
2. Fuentes, L., Gámez, N.: FamiWare: A Family of Event-based Middleware for Ambient Intelligence. Personal and Ubiquitous Computing 15(4), 329–339 (2011)

3. Wang, M.M., et al.: Middleware for Wireless Sensor Networks: A survey. Journal of Computer Science and Technology 23(3), 305–326 (2008)
4. Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering – Foundations, Principles, and Technique. Springer, Heidelberg (2005)
5. Lee, K., Kang, K., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 62–77. Springer, Heidelberg (2002)
6. IBM (2005), Autonomic computing white paper – An architectural blueprint for autonomic computing, IBM Corp. (2005)
7. Dimitriou, T., Krontiris, I.: Autonomic Communication Security in Sensor Networks. In: Stavrakakis, I., Smirnov, M. (eds.) WAC 2005. LNCS, vol. 3854, pp. 141–152. Springer, Heidelberg (2006)
8. He, Y., et al.: An autonomic routing framework for sensor networks. In: Cluster Computing, vol. 9, pp. 191–200 (2006)
9. Fuentes, L., Gámez, N.: Configuration Process of a Software Product Line for AmI Middleware. Journal of Universal Computer 16(12), 1592–1611 (2010)
10. Hydra: (November 2010), http://caosd.lcc.uma.es/spl/hydra/
11. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice 10, 143–169 (2005)
12. OMG Data Distribution Service for real-time systems, v1.2. (retrieved August 25, 2008)
13. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In: Proc. of the 1st Symp. on Networked Systems Design and Implementation (2004)
14. Liu, T., Martonosi, M.: Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In: ACM PPoPP 2003, pp. 107–118 (2003)
15. Grace, P., Hughes, D., Porter, B., Coulson, G., Blair, G.S.: Middleware Support for Dynamic Reconfiguration in Sensor Networks. In: Proc. IWSNE (2009)
16. Rocha, A., Delicato, F., Souza, N., Gomes, D., Pirmez, L.: A Semantic Middleware for Autonomic Wireless Sensor Networks. In: Proc. Workshop on Middleware for Ubiquitous and Pervasive Systems (2009)
17. Taherkordi, A., et al.: WiSeKit: A Distributed Middleware to Support Application-Level Adaptation in Sensor Networks. In: Senivongse, T., Oliveira, R. (eds.) DAIS 2009. LNCS, vol. 5523, pp. 44–58. Springer, Heidelberg (2009)
18. Trumler, W., Bagci, F., Petzold, J., Ungerer, T.: AMUN: an autonomic middleware for the Smart Doorplate Project. Personal and Ubiquitous Computing 10, 7–11 (2006)
19. Hoffert, J., Schmidt, D., Gokhale, A.: Adapting Distributed Real-time and Embedded Publish/Subscribe Middleware for Cloud-Computing Environments. In: Proc. of the 11th International Middleware Conference (2010)
20. Anthony, R., et al.: Context-Aware Adaptation in DySCAS. In: Proc. of 2nd Int. Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (2009)
21. Bencomo, N., Sawyer, P., Blair, G., Grace, P.: Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In: 2nd Int. Workshop on Dynamic Software Product Lines (2008)
22. Cetina, C., et al.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. IEEE Computer 42(10), 37–43 (2009)
23. Lee, J., Kang, K.: A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In: Proc. of the 10th Int. Software Product Line Conference, pp. 131–140. IEEE Computer Society, Los Alamitos (2006)

An Architecture Analysis Approach for Supporting Black-Box Software Development

Novia Admodisastro and Gerald Kotonya

School of Computing and Communications
InfoLab21, South Drive
Lancaster University
Lancaster LA1 4WA, UK
{admodisa,gerald}@comp.lancs.ac.uk

Abstract. A typical component-based system architecture comprises a set of components that have been purposefully designed and structured to ensure that they have "pluggable" interfaces and an acceptable match with a defined system context. However, the black-box nature of many software components means there is never a clean match between system specifications and concrete software components. Systematic architecture analysis can provide an effective, rapid and relatively low-cost mechanism for addressing risks resulting from architectural adaptation and trade-offs. However, a review of current architecture analysis approaches reveals they differ widely with respect to their ability to support black-box software development. This paper describes an analysis approach that integrates the strengths of current approaches to provide a practical architecture analysis framework for black-box component-based development. The approach is illustrated using a real case study.

Keywords: Architectural analysis, Components, Services, Black-box.

1 Introduction

Features supported by black-box third party software components often vary greatly in quality and complexity. In addition, the contexts in which the components are used may also vary. This complexity together with the variability in application contexts means that the documentation supplied with software components is often incomplete or inadequate. Additional analysis is often required to ensure that an acceptable solution is achieved, and to address situations where unforeseen user needs coincide with a component's undocumented design assumptions [1]. A key challenge in developing black-box software systems is how to provide developers with tools that allow them to derive viable software architectures by balancing aspects of stakeholder concerns with the architectural considerations and capabilities embodied in black-box software components.

Effective architecture analysis can provide the developer with a means to assess design configurations with respect to specific structural and behavioural constraints and to verify the adequacy of compositions with respect to stakeholder concerns. Architectural analysis can also provide a basis for developing "what-if" scenarios to

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 180–189, 2011.

explore the implications of evolving a system [2,3]. However, a study by [1] shows that current architecture analysis approaches differ widely with respect to their underlying models, analytical capabilities and ability to support black-box software development [4,5]. They vary from process-embedded models that derive skeleton architectures by matching non-functional requirements to architectural styles [6], to stakeholder-driven schemes that analyze architectures using multiple quality attributes to identify and improve areas of highest risk [7]. This makes it difficult for developers to ascertain their effectiveness in different application contexts.

Our solution has been to develop, *Component-based Software Architecture analysis FramEwork (CSAFE)*, a scenario-driven architecture analysis approach that combines the strengths of current approaches using a framework that allows different analysis techniques to be integrated. CSAFE is process-pluggable and recognises that negotiation is central to black-box software development. The rest of this paper is organized as follows; section 2 describes the architectural analysis framework. Section 3 describes the case study. Section 4 uses a case study to illustrate our approach. Lastly, Section 5 provides some concluding thoughts.

2 The Analysis Framework

CSAFE is a scenario-driven architecture analysis approach intended to support black-box development. The analysis process is supported by a repository of component specifications that represent concrete components and design templates that embody specific design goals and best practice. CSAFE comprises 4 iterative steps as shown in Fig. 1:

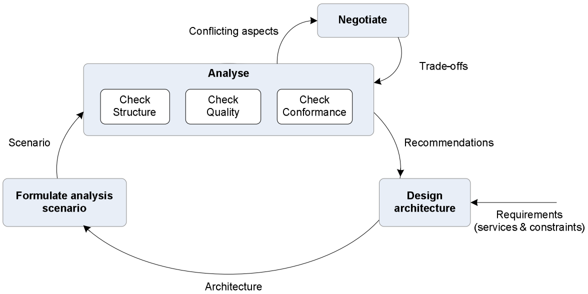


Fig. 1. Architectural analysis process

2.1 Architectural Design

The architecture design stage is concerned with the construction of the system architecture. The CSAFE analysis process accepts architectures expressed in the standard UML component notation [8] or in iXML[1] architecture description language.

[1] iXML is xml-based architectural description language which intended to sustain independent architectural analysis which is not tied to a particular language, toolset or methodology.

iXML is an XML-based architectural description language designed to support analysis in CSAFE. There is not enough space in this paper to provide a detailed description of the ADL. The ADL serves three purposes; first, provides a mechanism for analysing both pre-existing and new architectures. Secondly, it allows for a portable, platform independent description of the system architecture. Lastly, it provides the system designer with a mechanism for conducting "what if" analysis.

CSAFE uses a service-oriented requirements method that maps services and constraints onto iXML to support the derivation architectures from scratch. Discussion of the requirements method is out of the scope of this paper. A detailed description of the requirements method can be found here [11].

2.2 Scenario Formulation

Analysis scenarios are formulated after architectural transformation has taken place. Analysis scenarios allow system designers to tailor the analysis to explore specific design questions by providing a means to augment architectural descriptions with specific quality concerns and other architectural information. Designers can also formulate scenarios to explore "what if" analysis such as assessing the impact of change and competing designs. Table 1 shows the elements of an analysis scenario.

Table 1. Elements of analysis scenario

Aspect	Description
Concern	A desired quality attribute that acts as goal to be addressed and achieved during the process of architectural design. Concerns may be categorized as follows: • Requirement (e.g. performance, security, efficiency availability, maintainability), • Component (e.g. certification, standards, resources etc.) • Business (e.g. nature of support, trust, cost)
Sub-concern	A lower level of concern that allows either qualitative or quantitative measurement.
Refinement	Refinement expresses concern/sub-concern in more details.
Scope	Identifies service or component affected by a concern/sub-concern. Scope also serves as traceability mechanism.
Weighting	Prioritises concerns. Values assigned to quality concerns are likely to vary with application and organization. For the purpose of the example described in this paper, we have adopted a 3-level weighting scheme that relates the value for required features to customer satisfaction and system operation. The weighting scheme of High (H), Medium (M) and Low (L) is associated with quantitative values of 3, 2 and 1: • High denotes core quality concerns. Failure to provide these features means the system will not meet customer needs. • Medium denotes features that are important to the effectiveness and efficiency of the system. Lack of inclusion of an important feature may affect customer satisfaction. • Low denotes features that are useful but not central to the system operation. Lack of inclusion of a useful feature will not have significant impact on customer satisfaction.

2.3 Analysis

The analysis process allows the developer to establish how well a proposed system design satisfies its application and business contexts. CSAFE provides an extensible analysis framework based on a service-oriented architecture and XML that allows the system designer to integrate different analysis methods and techniques (see Fig. 4). Currently the analysis process provides support for:

- *Structure checking*. Identifies mismatches between component interfaces and other incompatibilities in component interconnections.
- *Quality checking*. Identifies inconsistencies and mismatches between desired quality attributes (dependability, component, etc.) and the system context.
- *Conformance checking*. Checks architecture adherence to design heuristics

Fig. 2 shows the CSAFE analysis algorithm.

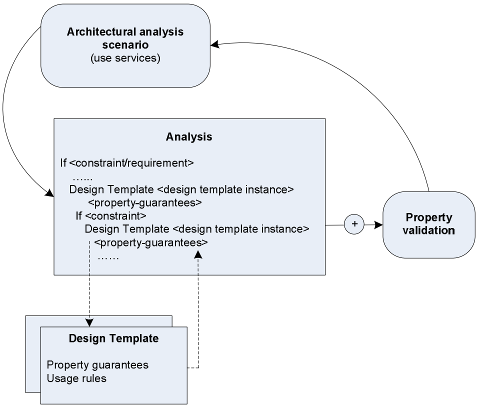


Fig. 2. Mapping analysis scenarios to analysis process

Fig. 3 shows the elements of a design template.

{*Category*} Type, i.e. style, design pattern, local scheme.
{*Name*} The design template name.
{*Also-Known-As*} Other well-known names for the design template if any.
{*Related-Template*} Reference to other closely related design templates.
{*Intent*} The justification for design template
{*Context*} The situation in which the template may apply.
{*Motivation*} Describes template solution.
{Configuration} Specification of the template
{Consequences (Contribution)} Specification of dependency and contribution that template may possess shown in weighting factor.

Fig. 3. Design template structure

2.4 CSAFE Toolset

The CSAFE process is supported with an integrated toolset. The toolset has six main components as shown in Fig. 4. The operation of some of the tools has already been discussed so we will focus the XMI/XML parser, the trade-off analyser and the component repository:

- *XMI/XML parser*. Supports the early stage of the CSAFE process by parsing software architectures transformed from UML to XMI/XML or iXML for analysis.
- *Trade-off analyser (Negotiation)*. Provides the designer with a tool for rating and trading-off competing solutions.

- *Component repository*. Is a searchable respository of black-box component specifications. The components are specified in XML.

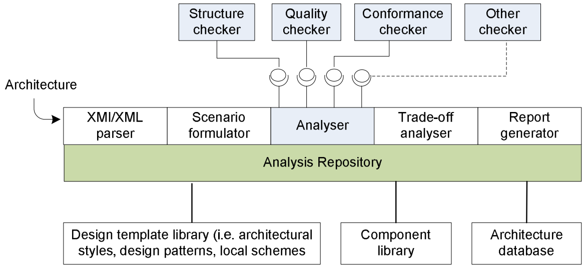


Fig. 4. CSAFE toolset

3 The Case Study

We will illustrate the efficacy of CSAFE with an example derived from a real software project. The *Electronic Document Delivery and Interchange System* (EDDIS) project was concerned with developing an online library system to support the searching, ordering and supplying of electronic documents [10]. Users access the system via a web-based interface using valid usernames and passwords. A user must obtain document and location identifiers from a centralized document registry before placing a document order. Document orders are placed with the document supplier. All document interchange use the Z39.50 document retrieval standard. The EDDIS local administrator is responsible for setting and managing user accounts. A detailed description of the system can be found in [11]. The next sections will provide a summarised description of how the EDDIS requirements are mapped into the initial system architecture and the analysis of the architecture using CSAFE.

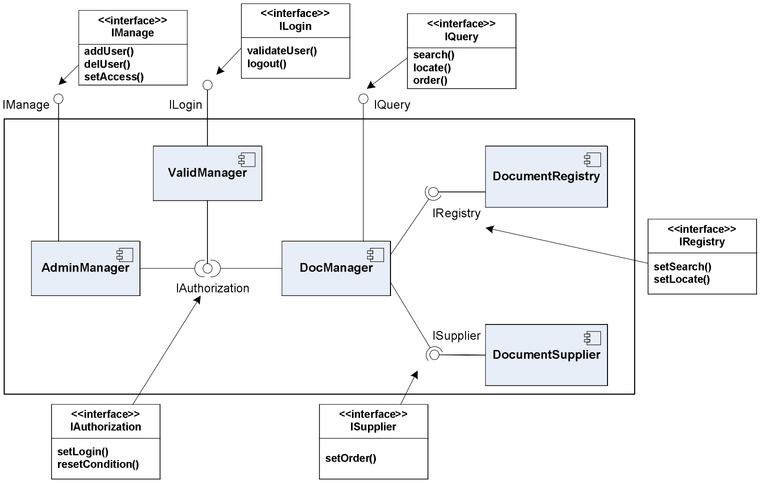


Fig. 5. Initial EDDIS architecture

3.1 Initial EDDIS Architecture

Fig. 5 shows the initial EDDIS component architecture. It was decided that functionality for the *AdminManager*, *ValidManager* and *DocManager* would be provided using off-the-shelf components while *DocumentRegistry* and *DocumentSupplier* would be provided using Web services.

4 The Analysis

4.1 Converting Architecture to XMI Specification

The analysis stage begins with the transformation of the UML architectural design of EDDIS into a processable XMI specification [12]. The XMI/XML parser supports the transformation process by parsing and storing XMI/XML objects in an analysis repository, which is accessible by other CSAFE tools. The parser provides a uniform interface to the underlying XML object model that captures elements of the architecture (i.e. architectural structure with its descriptions, services, constraints and properties).

4.2 Formulating Analysis Scenarios

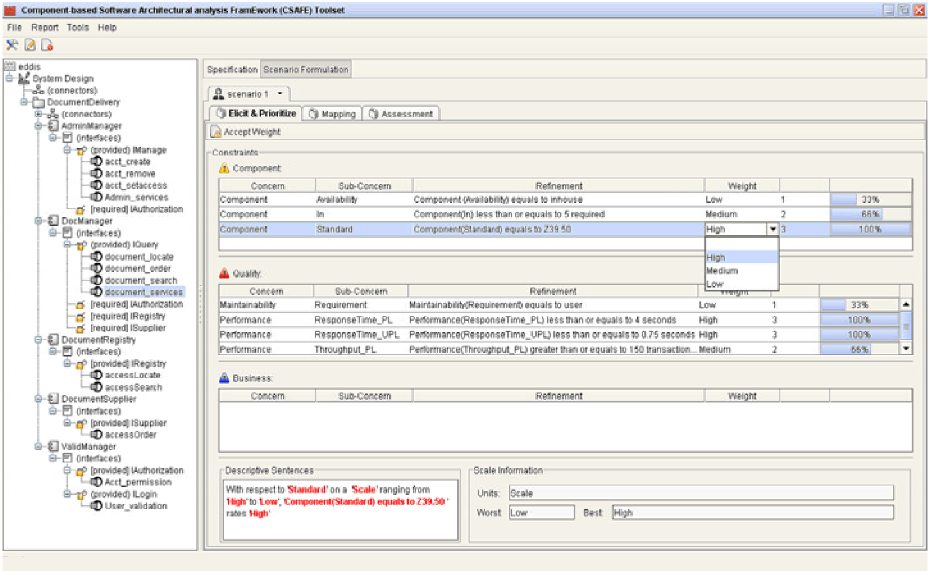
After architectural transformation has taken place, analysis scenarios can be formulated as described in section 2.2. Analysis scenarios are a simple yet effective way to represent quality concerns as goals to be addressed and achieved during the analysis of the architecture. The scope of analysis scenarios can be selective (i.e. associated with specific architectural properties) or global. Fig. 6 shows the example of a selective scenario associated with the *document_services* service. Constraints (shown in centre of the right pane) are expressed in structured natural language in the form:

Concern(SubConcern) <condition><value>unit.

A parser processes the various constraints and their weightings, and generates queries, which search the design template repository for matching design templates. The search result also shows the contribution of the design templates to the quality concerns. Three design templates have been generated for the scenario in Fig. 6:

- *Service-Order Provision*. This template represents a local (in-house) design solution for an online digital library that may require *document search*, *document locate* and *document order* services. The architectural style enforces the separation of *search* and *locate* services, which reside in the same component, from the *order* service. The design strongly supports the maintainability requirement by providing a systematic allocation towards maintenance time for the document main services and allowing the document server to maintain the *order* service more effectively. The design also provides a better way to control availability by allowing a longer duration of the *order* service to be served. However, separation of the services may affect performance of response time and throughput.

- *Cluster-Server pattern.* This design enables the system to maintain good performance while improving availability by using active redundancy and automatic restart during fail over. However, cluster-server complexity is likely to compromise system maintainability.
- *Three-tier proxy server* architectural style [13]. This is a typical reference architecture for a modern web-based system. A tier is a partitioning of functionality that may be allocated to a separate hardware component. This improves maintainability while hiding the complexity of distributed processing. Requests from individual browsers may first arrive at a proxy server, which exists to improve the performance of the Web-based system. They are typically located close to the users, often on the same network, so that they save significant communication and computation resources.



Component:

Concern	Sub-Concern	Refinement	Weight		
Component	Availability	Component (Availability) equals to inhouse	Low	1	33%
Component	In	Component(In) less than or equals to 5 required	Medium	2	66%
Component	Standard	Component(Standard) equals to Z39.50	High	3	100%

Quality:

Concern	Sub-Concern	Refinement	Weight		
Maintainability	Requirement	Maintainability(Requirement) equals to user	Low	1	33%
Performance	ResponseTime_PL	Performance(ResponseTime_PL) less than or equals to 4 seconds	High	3	100%
Performance	ResponseTime_UPL	Performance(ResponseTime_UPL) less than or equals to 0.75 seconds	High	3	100%
Performance	Throughput_PL	Performance(Throughput_PL) greater than or equals to 150 transaction...	Medium	2	66%

Business:

Concern	Sub-Concern	Refinement	Weight		

Descriptive Sentences

With respect to 'Standard' on a 'Scale' ranging from 'High' to 'Low', 'Component(Standard) equals to Z39.50' rates 'High'

Scale Information

Units: Scale

Worst: Low Best: High

Fig. 6. Formulating analysis scenarios

4.3 Modify Architecture or Sub-system Architecture

The last step is to assess how well the three architectural styles contribute to the quality concerns. CSAFE uses the Analytical Hierarchical Process (AHP) [9] to perform the trade-off analysis. Fig. 7 shows the EDDIS architecture based on the three-tier proxy server design template. Modifications to the original architecture are shown in the boxed area.

Fig. 8 shows the weighted contributions of the design alternatives using the AHP. The maximum mean weighting in indicates the highest probability of achieving a particular quality concern. The minimum mean value indicates the least acceptable trade-off. Each sub-concern in a design that has a weighting score below the minimum mean value decreases the chance of the design being chosen as the best design. In addition, a design can be rejected if its overall mean score is below the overall minimum mean value of the quality concerns. As this implies that the design does not meet the stakeholders' expected qualities. S1 offers the poorest solution as its overall quality contribution score of 0.58 is below the overall minimum mean of the quality concerns. Of the remaining two, S3 has the better mean score. However, both the architectures exceed minimum mean value. Although, S3 looks like the best design, it may not necessarily be chosen. For example, the cost of implementing the system using S3 may be beyond the organisation's budget. To decide on the most acceptable architecture, stakeholders need to explore how each design relates to the minimum mean values of critical sub-concerns.

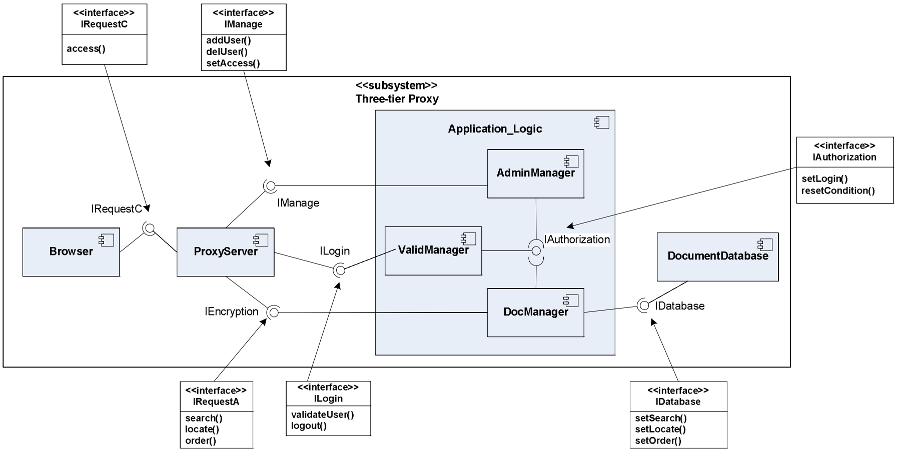


Fig. 7. EDDIS architecture using three-tier proxy server template

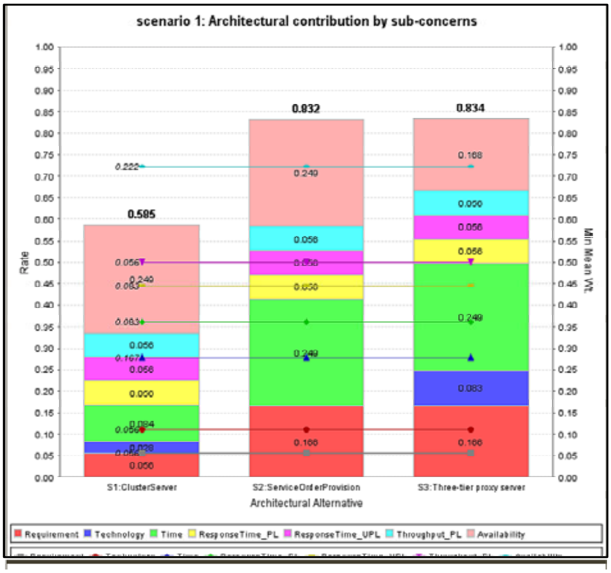


Fig. 8. Architectural contribution by sub-concerns

5 Conclusions

This paper has highlighted the importance of architectural analysis in black-box component-based software development. Systematic architectural analysis can help ensure that risks resulting from architectural adaptations and trade-offs do not adversely affect critical system qualities. The analysis is likely to reveal not only how well an architecture satisfies a particular application context, but also how change to specific quality attributes might affect other quality concerns. Unfortunately, current architectural analysis approaches for component-based vary widely with respect to their analytical capabilities and support for black-box development making it difficult for developers to assess their efficacy in different application contexts.

Our proposed solution aims to address the challenges outlined in Section 1 by providing an analysis approach that:

- Is pluggable to minimise process disruption.
- Allows different analysis techniques and methods to be integrated to leverage their strengths
- Is portable and supports the analysis of architectures expressed in standard software design notations like UML.
- Provides mechanisms defining analysis scenarios that allow stakeholders to explore aspects of the system that interests them and supports for “what-if” analysis under conditions of uncertainty.
- Explicitly recognises that negotiation is central to successful black-box component system development.

Due page limitation we could provide a detailed discussion of the case study. However, we believe we have provided enough evidence demonstrate the efficacy of CSAFE. It is important to mention that CSAFE has been evaluated on a real, but limited case study. The results presented here represent the initial evaluation of CSAFE. We are currently extending CSAFE to include other interesting architectural analyses. We are also exploring better ways of supporting stakeholder analysis and managing the large volume of information generated before evaluating the approach on a larger system.

References

1. Admodisastro, N., Kotonya, G.: Architectural Analysis Approaches: A Component-Based System Development Perspective. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 26–38. Springer, Heidelberg (2008)
2. Kotonya, G., Hutchinson, J.: Managing Change in COTS-Based Systems. In: Proc. of the IEEE ICSM, pp. 69–78, pp. 69–78. IEEE Computer Society, Washington, D.C (2005)
3. Dobrica, L., Eila, N.: A Survey on Software Architecture Analysis Methods. IEEE Trans. on Soft. Eng. 28(7), 638–653 (2002)
4. Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L.: Recommended Best Industrial Prac. for Soft. Arch. Evaluation. Tech. Report, CMU/SEI-96-TR-025 (1997)
5. Vieira, M.E.R., Dias, M.S., Richardson, D.J.: Analyzing Software Architectures with Argus-I. In: Proceedings of the ICSE, pp. 758–761, Limerick, Ireland (2000)

6. Wallnau, K.C.: Volume III: A Technology for Predicable Assembly from Certifiable Components. Technical Rep. CMY/SEI-2003-TR-009. SEI Carnegie Mellon Uni. (2003)
7. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J.: The Architectural Tradeoff Analysis Method. In: Proceeding of IEEE ICECCS, pp. 68–78 (1998)
8. Unified Modeling Language. UML® Resource Page (2010), `http://www.uml.org/` (last updated on October 21, 2010)
9. Hutchinson, J., Kotonya, G.: A Review of Negotiation Techniques in Component-Based Software Engineering. In: Proc. of the EuroMicro Conf. on SEAA, pp. 152–159 (2006)
10. Kotonya, G.: An Architecture-Centric Development Environment for Black-Box Component-Based Systems. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 98–113. Springer, Heidelberg (2008)
11. Kotonya, G., Hutchinson, J.: Analysing the Impact of Change in COTS-Based Systems. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 212–222. Springer, Heidelberg (2005)
12. XML and XMI. CORBA, XML and XMI (2010), `http://www.omg.org/technology/xml/index.htm` (last updated on June 25, 2009)
13. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. SEI Series in Software Engineering. Addison-Wesley, Reading (2005)

Web-Scale Human Task Management

Daniel Schulte

FernUniversität in Hagen, Germany
Daniel.Schulte@FernUni-Hagen.de

Abstract. Today, many professionals work on several projects in different teams at the same time and manage their task in these projects manually or with the help of a task management system. Business portals in larger companies typically provide some kind of task management support. But often they lack facilities to manage tasks beyond organizational boundaries and need technical support by IT departments. To overcome these limitations, especially loosely coupled teams make extensive use of emails and forgo explicit task management features.

As the usage of web applications in all areas of life increases and thus more and more tasks are performed online, new solutions for web-based task management have become necessary. In this paper, we determine requirements for managing tasks in distributed environments without any central supervisory body, and identify related research challenges.

1 Introduction

Teams cooperate and coordinate their work over the internet using web applications increasingly. As professionals are typically involved in several teams and projects at the same time, they have to work with a plethora of web applications.

Within these applications many tasks have to be performed by humans and some of these applications include rudimentary task management facilities for these tasks. However, no application-spanning web-scale human task management has been established yet. Instead, vendors rely on proprietary portals occasionally based on WS-HumanTask [1] to support task management within enterprises and on email notifications for all communication with participants outside enterprises. As emails were originally intended for communication, email applications lack task management facilities and hamper team members — especially those involved in multiple projects— to survey and manage their tasks efficiently.

As the number of human tasks performed in a multitude of web applications increases, new solutions for managing tasks in the web have become necessary. After briefly explaining our human task concept in sec. 2, we will determine key challenges for managing tasks in the web in sec. 3 and identify some resultant research challenges in sec. 4 including our intended research method.

2 Human Tasks

To cover as much manifestations of human tasks as possible, we define *human tasks* broadly as actions that will be carried out by humans. This spans simple

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 190–193, 2011.

data entry tasks controlled by rigid input masks but also creative tasks of knowledge workers. We are interested in those human tasks that are known to online applications as these applications have the potential to provide task descriptions to affected humans in a kind that allows automated task management support (other human tasks can be added manually), and —if performed with help of web applications— to update these metadata automatically.

Sample 1: Review Task. While preparing a workshop, the workshop chair will ask experienced scientist for reviewing submissions. Each submission is —based on expertise and preferences— assigned to a few scientists, who get a copy of the submission. The scientists perform their reviews independently and submit them to the program chair.

Sample 2: Writing Task. Web applications like Google Writer support small teams to prepare workshop papers jointly as they allow concurrent editing of documents and save intermediate results (with versioning). Writing tasks may thus be assigned to several humans in parallel and as a consequence, the state of a task (e. g., "in work" or "submitted to the workshop") will change in the normal course of the task execution potentially triggered by other team members.

These samples show different involved users and web applications:

- users that will/may perform tasks ((potential) task workers) like the reviewers in sample 1,
- users that initiate tasks (task initiators) like the program chair in sample 1,
- applications that help task initiators to initiate and assign tasks (task steering applications) as tools like ConfTool for program chairs,
- applications that help task workers to perform tasks (task execution application) as Google Writer in sample 2,
- applications that help task workers but also task initiators managing tasks (task management applications).

Objective of the planed research is an infrastructure to enable web-scale task management applications.

3 Key Challenges of Web-Scale Human Task Management

Task management comprises collection and maintenance of task metadata provided and possibly updated by task steering & task execution applications (hereinafter collectively referred to as task provision applications) as well as manipulation of metadata by users, scheduling of tasks, and so on. This section identifies main challenges of web-scale human task management.

To survey and manage tasks efficiently, users should be able to access all their tasks by a *single worklist* and have access to *task management facilities* that can handle tasks regardless of the origin. Ultimately, users need only one task management application (not one task management application per organization, (virtual) enterprise, team, task provision application, etc.).

As tasks emerge in task provision applications, these applications should *add tasks to users worklists* incl. some metadata like deadlines directly. As tasks change over time (state, deadlines, etc.), task provision applications need to *update these metadata* automatically.

The assignment of tasks within teams may depend on the roles of team members, on group policies, on current workloads or on any other kinds of policies or agreements. To support individual assignment patterns independently of task provision and task management applications, a self-sufficient *group management* should take on the assignment of tasks.

In web-scale scenarios, applications originate from different organizations and application areas, are written in different programming languages, etc. To enable the development of a rich ecosystem of task provision and task management applications, prevent dependence on central authorities (vendor lock-in, quasi-monopolies like Facebook) as well as single points of failures, and allow the integration of the variety of established web applications already containing and supporting human tasks, a web-scale human task management has to ensure interoperability between applications but also to meet some architectural constraints:

- To bring not only tasks from some selected applications (e. g. those that are supported by some central infrastructure like an Enterprise Service Bus) together but also tasks from any application incl. already used ones, a *decentralized solution* is required.
- Tasks originate in different contexts with different execution steps and states as well as different security requirements. Their types may span routine jobs, adaptive tasks and innovative ones [6], and may be executed by one or several humans cooperatively or competitively, to name a few options. Therefore, *task provision applications should be self-contained*: they control task execution incl. individual lifecycles for tasks, and —in the case of task execution application— access to security-critical applications and data by appropriate self-selected mechanisms.
- The execution of tasks in many scenarios is free to users' choice. Therefore, some management functions like task delegation and in particular the deletion of tasks from worklists should be supported by task management applications independent of task provision applications, so that users remain in control of their worklists. But as work on business-related tasks may be enforced by employers, management functions are more restricted in some cases. To allow different degrees of freedom in task management, *task management applications must be autonomous.*

Furthermore, the architecture of a web-scale human task management should allow the independent evolvability of applications, a common authentication and authorization mechanism, and substitution of applications at runtime.

4 Research Challenges and Research Method

E-mail has no task management facilities, WS-HumanTask based solutions build on centralized and partial proprietary building blocks, and action-centric collaboration solutions like HERMES [7] provide deep integrations at the expense of autonomy of applications (e. g., shared internal data) but do not consider decentralization. Therefore, an interoperable but decentralized web-scale human task management with self-contained task provision applications and autonomous task management applications leads to research questions like:

- Which architectural decisions are necessary to enable web-scale human task management whereas heterogeneous concrete software architectures, programming languages and deployment infrastructures are used and evolved independently from each other?
- As reuse of software is often hard due to architectural mismatches as depicted by [4,5], can we avoid similar problems with the reuse of already existing web applications for human task execution? Or to put it another way: How can we adapt web applications (some kind of legacy systems) to interoperate with such a new infrastructure?
- How can we continue to use established web technologies like OpenID and OAuth?

We will investigate these research questions with the help of architectural prototypes [3], which allow us to compare different architecture styles (based on different architectural decisions), and develop a proof of concept on top of this prototypes. [8] identifies workflow resource patterns capturing the various appearance patterns of resources (e. g. a human) in workflows, and [9] evaluates BPEL4People [2] and WS-HumanTask [1] using these resource patterns. Analogously, we will use them to evaluate the power of different solutions.

References

1. Agrawal, A., et al.: Web Services Human Task (WS-HumanTask), Version 1.0 (2007)
2. Agrawal, A., et al.: WS-BPEL Extension for People (BPEL4People), Version 1.0 (2007)
3. Bardram, J.E., Christensen, H.B., Hansen, K.M.: Architectural Prototyping: An Approach for Grounding Architectural Design and Learning. In: Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004, pp. 15–24. IEEE Computer Society, Washington, DC (2004)
4. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse Is So Hard. IEEE Softw. 12(6), 17–26 (1995)
5. Garlan, D., Allen, R., Ockerbloom, J.: Architectural Mismatch: Why Reuse Is Still So Hard. IEEE Softw. 26(4), 66–69 (2009)
6. Hoffmann, F.: Aufgabe (german). In: Grochla, E. (ed.) Handwörterbuch der Organisation, pp. 200–207. Poeschel, Stuttgart (1980)
7. Kapos, G.-D., Tsalgatidou, A. & Nikolaidou, M.: A Web Service-Based Platform for CSCW over Heterogeneous End-User Applications. In: ISCA 17th International Conference on Parallel and Distributed Computing Systems, PDCS 2004, pp. 462–469 (2004)
8. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow Resource Patterns. Eindhoven University of Technology (2004)
9. Russell, N., van der Aalst, W.M.P.: Evaluation of the BPEL4People and WS-HumanTask extensions to WS-BPEL 2.0 using the workflow resource patterns. BPM Center (2007)

Enhancing Architecture Design Methods for Improved Flexibility in Long-Living Information Systems

Matthias Naab

Fraunhofer Institute for Experimental Software Engineering (IESE),
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
matthias.naab@iese.fraunhofer.de

Abstract. Nearly all organizations in business are highly relying on information systems. As business, business models, organizational structures, and business processes are changing quickly, also information systems have to follow these changes, otherwise they threaten the business success. A key quality attribute of software systems, which is defined to capture the needs for change, is flexibility. Although many of today's IT paradigms like service-oriented architecture or business rule management claim to bring flexibility into information systems, this is often not achieved in practice, as experience shows. This paper explores in more detail the nature of flexibility and proposes an extension to architecture design processes, which allows constructing systems with flexibility directed at the real needs. It makes flexibility more tangible and gives concrete guidance for treating flexibility during architecture design.

Keywords: Software Architecture, Flexibility, Service-Oriented Architecture.

1 Introduction

Flexibility is a key quality attribute of today's information systems. On the one hand, such systems have to keep pace with quick changes in business, business models, organizational structures, and business processes. On the other hand, they are so costly to develop that organizations stick to their information system even for decades. Only if the information systems provide support for effective and efficient realization of changes it can sustainably contribute to an organization's business success. Flexibility is the quality attribute that represents a system's ability to quickly response to particular changes to a system.

Several recent paradigms in IT, like service-oriented architecture (SOA) or business rule management (BRM), are advertised for their strong support for flexibility in information systems. Nevertheless, practice shows that many of the systems in use are not as flexible as expected although all promising paradigms and technologies are in place. There are several causes that contribute to this situation: First, such paradigms and technologies are often expected to deliver systems with inherent flexibility. Second, flexibility as a quality attribute is not as well understood so far as other quality attributes like performance are. Third, architecture design processes don't provide concrete support for the construction of flexible information systems. They neither

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 194–198, 2011.

address the particularities of flexibility nor give concrete guidance how to come from flexibility requirements to an adequate solution.

This paper starts with an exploration of the nature of flexibility in Section 2. It analyzes reasons why systems are often not flexible in practice and proposes a conceptual model for flexibility. Section 3 therefore focuses on introducing methodical guidance for designing flexible architectures. Section 4 gives an overview on initial validation efforts and concludes the paper with discussion and outlook.

2 Characterization of Flexibility

Flexibility Foundations

In general, *flexibility* is “the degree, to which a system supports possible or future changes to its requirements” [11, adapted from 8]. The degree, to which changes are supported, is reflected in time and effort, which are needed to conduct the changes. With flexibility we mean the pure property of a system to allow changes, in contrast to other terms like maintainability, which also covers aspects like analyzability or testability.

Flexibility is not an absolute property of a system. Rather, flexibility can be only quantified with respect to a certain set of requirements to be changed [5]. Mostly, requirements emerging or changing during a system’s life time don’t change totally surprisingly, but can be anticipated upfront. Of course, this is not always possible and most of the time only with a certain degree of fuzziness. Such requirements can be captured as flexibility requirements. Approaches exist that support their elicitation [9].

Why Information Systems in Practice Are Often Not Flexible Enough

In many projects with our industrial customers we observed that systems are often not as flexible as they were expected to be. Information systems based on *Service-Oriented Architecture* (SOA) are a good foundation for an analysis of reasons. SOA is often proclaimed to lead to flexible systems and customers have also adopted the expectation of getting flexible systems [10]. Often, a typical procedure in designing SOA-based systems can be found: Typical architectural mechanisms and technologies of SOA are adopted (separation of process and service, loose coupling in general, technology-independence, …). Then the functionality of the system under development is decomposed and mapped onto the technological elements. The resulting system is expected to be flexible due to the architectural mechanisms used; however, often it is not flexible.

In Fig. 1 a), we analyze in a set notation, which situations can occur and why systems might lack flexibility. (1) be a set of flexibility requirements. A first reason of missing flexibility is a lack of awareness of these flexibility requirements (they occur later, but are not considered during construction). Then, no targeted flexibility can be built in.

To construct for flexibility, architecture mechanisms are selected, which keep the impact of anticipated changes as small and local as possible. (2) be the set of all potential changes, which are well supported by the selected architecture mechanisms,

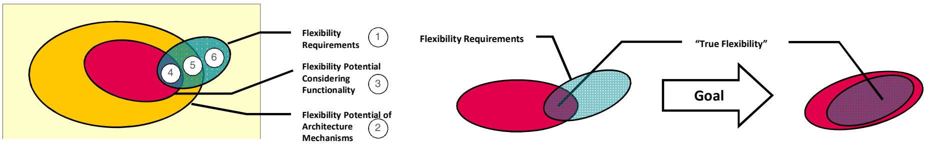

Flexibility Requirements
1
Flexibility Potential Considering Functionality
3
Flexibility Potential of Architecture Mechanisms
2
4
5
6
Flexibility Requirements
"True Flexibility"
Goal

Fig. 1. a) Mismatch of flexibility requirements and flexibility potential | b) "True Flexibility"

the so-called flexibility potential. Typically, not all flexibility requirements are covered in the flexibility potential (see (6)), as recent methods don't provide enough guidance for architects. However, the architecture mechanisms alone don't decide about the flexibility potential. The actual degree of flexibility is mainly determined by the mapping of functionality to the architecture mechanisms. This reduces the flexibility potential as depicted as (3). An example from SOA systems is a suboptimal mapping of functionality to services, which leads to high effort every time a business process is to be changed. Then, only a limited set of flexibility requirements (4) is really covered by the flexibility potential. Further flexibility requirements are not easily achieved any more (5). There is typically little awareness for this situation and architecture design approaches don't provide sufficient support.

To sum up, typically, there is a large flexibility potential in systems but it does not match the required flexibility. This mismatch is the reason why systems are often perceived to be not flexible enough. Thus, we need a new notion to express the desired situation that the flexibility potential matches the required flexibility: We call this "True Flexibility" (see Fig. 1 b)). Typically, people talking about flexibility implicitly mean true flexibility.

3 Architecture Design for Flexibility

Derived from our characterization of flexibility, the goal of our method is to achieve true flexibility. Several sub-goals can be derived: First, flexibility requirements have to be known to a reasonable degree for architecture design (G1). Second, the flexibility potential resulting from selected architectural mechanisms and mapping of functionality should cover all flexibility requirements (G2). Third, as flexibility always comes at a cost (more complicated development, performance impacts, …), the flexibility potential should not be inadequately large but clearly bound to the required flexibility (G3).

State of the Art and Improvement Potential

Flexibility is in the focus of architecture research and practice since a while. Two different trends can be observed: First, there is research focusing on analytical approaches [5, 7]. A commonality of these approaches is that they work with anticipated change situations (described as scenarios) and derive to a more or less formal degree the expected impact or rework necessary for a scenario. Second, there is research focusing on architectural mechanisms enabling flexible systems [3, 4].

What's missing is constructive guidance with focus on flexibility. Existing architecture design approaches can be roughly separated into two categories. The first category mainly focuses on a functional decomposition of the system (e.g. [2]); the second category mainly focuses on addressing the quality attributes of the system (e.g.[4]). Several approaches in the area of service-oriented system design (e.g. [1]) also belong to the first category. Addressing flexibility inherently requires the combination of both types of approaches due to the dual nature of flexibility. In [6], an approach is described, which covers both, functional decomposition and explicit addressing of quality attributes, but only very general and without any specific guidance with respect to the nature of flexibility.

Method Outline

Our approach is intended to enhance existing methods addressing the challenges and goals described above by adding conceptual and methodical parts explicitly addressing flexibility. Concretely, that means that we provide more support for all architectural activities with a focus on flexibility. That starts with support for elicitation of flexibility requirements. We provide classifications, a quality model, and templates that ease gathering and describing of flexibility requirements in form of scenarios. The main enhancement of the design activities themselves is directed towards intertwining aspects of functional decomposition and quality-driven design (application of architectural mechanisms). We introduce finer-grained design steps, classifications of architectural elements and their role in the process, and guidelines how to come up with a mapping of functionality to architectural mechanisms that supports the goals of true flexibility. These enhancements are supposed to make experiences about design for flexibility explicit and to allow architects to benefit from it.

Of course, our approach does not mean that architecture design should be flexibility-centric. Rather, it adds explicit support for flexibility as one challenging quality attribute. Further support for other quality attributes can be added in similar manner resulting in better access to experience for architects. Architecture is still exactly the means to carefully and deliberately balance all quality attributes, but with better methodical support.

4 Initial Validation and Conclusion

We observed and analyzed the challenges in industrial projects with our customers. We explicitly confirmed with customers that practitioners often assume that IT paradigms and technologies lead to the required flexibility. After discussing this point, we asked our customers for their perceived support of flexibility in architecture design processes, which was rated pretty low.

We also started applying our approach in our industrial projects to validate its effectiveness. Here, we present only some first results and impressions:

– Eliciting flexibility requirements without further guidance often leads to results with limited coverage of the real needs. Practitioners are often guided by their expectations what a technology can offer. In contrast, eliciting flexibility requirements with our guidelines and classifications leads on average to higher coverage
– Making the influence of architecture mechanisms and functionality mapping on flexibility explicit is perceived as significant help by practitioners

- Giving guidance with respect to architectural elements to be designed and how flexibility is achieved (in particular with more details on flexibility mechanisms, which are not presented in this paper) is perceived as significant help by practitioners

Flexibility is a key quality attribute of long-living software systems. In this paper we provide a well-founded characterization of flexibility and analyze, why flexibility is often not achieved in practice. First, there is the perception that particular technologies lead to inherent flexibility. Second, there is missing awareness of importance of both, architectural mechanisms and adequate mapping of functionality. Third, there is missing guidance for flexibility in architecture design processes.

We introduce the term *true flexibility* to denote a match between required flexibility and the realized flexibility potential. Based on that, we sketch how existing architecture design processes can be extended with more guidance tailored to the quality attribute flexibility. In projects with customers from industry we already applied the concepts and got positive feedback. In the future, we will make the guidance more concrete and we will also make the flexibility-specific knowledge about certain paradigms like SOA or BRM more tangible. Further, we plan to support architects with direct, automated feedback on architectural design with respect to the achieved degree of flexibility.

Acknowledgement. This paper was supported by the ADiWa project. The ADiWa project was funded by the German Federal Ministry of Education and Research (BMBF) under grant 01IA08006. The authors take the responsibility for the contents.

References

1. Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Gariapathy, S., Holley, K.: SOMA: a method for developing service-oriented solutions. IBM Syst. J. 47(3), 377–396 (2008)
2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, R.: Component-based product line engineering with UML. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2002)
3. Bachmann, F., Bass, L., Nord, R.: Modifiability Tactics. CMU/SEI-2007-TR-002 (2007)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2003)
5. Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H.: Architecture-level modifiability analysis (ALMA). J. Syst. Softw. 69(1-2), 129–147 (2004)
6. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley Publishing Co. (2000)
7. Clements, P., Kazman, R., Klein, M.: Evaluating software architectures: methods and case studies. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2002)
8. IEEE: Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology (1990)
9. John, I., Villela, K.: Evolutionary Product Line Requirements Engineering. In: Proceedings of the 2008 12th International Software Product Line Conference. IEEE Computer Society, Los Alamitos (2008)
10. Martin, W.: SOA Check (2010), `http://www.soa-check.eu/` (last access: June 22, 2011)
11. Naab, M.: Improving the Flexibility of SOA-Based Information Systems by Adopting Practices from Product Line Engineering. In: Doctoral Symposium of SPLC (2009)

On How to Deal with Uncertainty When Architecting Embedded Software and Systems

Jakob Axelsson

School of Innovation, Design and Engineering,
Mälardalen University,
SE-721 23 Västerås, Sweden
jakob.axelsson@mdh.se

Abstract. This paper discusses the topic of uncertainty in the context of architecting embedded software and systems. It presents links between complexity and uncertainty, and identifies different kinds of uncertainty. Based on this, it elaborates why uncertainty arises in the architecting of software-intensive systems, and presents ten different tactics that can be employed to deal with uncertainty and mitigate the associated risks.

1 Introduction

In many companies developing technical products, embedded systems and software play an increasingly important role. From being a small and isolated electronics-based part of a product, the embedded system has developed into a large number of computers with distribution networks and millions of lines of software. This increasing complexity leads to soaring developing costs, and many companies strive to curb this trend by reusing software and hardware between products. Often, a product line approach is applied, where the same platform is used as a basis, with modifications to fit the needs of the individual products and customers.

With a multiplicity of products and variants, the architecture is becoming very important and is a source of increasing interest for companies developing embedded systems. When developing a new platform for embedded systems, much of the architecting occurs 2-4 years before the first product is delivered. At that time, a number of key factors are unknown or known only with large uncertainty, such as the details of the software code, including its structure or execution time; the details of the hardware if new components are to be used (no prototypes available, only specifications); and the exact requirements (only an early estimate is available).

The complexity of these already complex systems increases as a result of this uncertainty, since it leaves open a larger space of possible design alternatives that the architects must consider. Many architects rarely use the tools and methods proposed by academic researchers even if they are aware of them. We have come to suspect that one reason is that many methods and tools are difficult or expensive to use in a situation where only imperfect information is available. In this paper, we therefore investigate the nature of the uncertainty in the architecting situation, and provide some ideas for how this uncertainty can be dealt with.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 199–202, 2011.

2 Types of Uncertainty

A development team's responsibility is to make decisions about the product. *Uncertainty* can be defined as a lack of necessary knowledge to make a decision [1]. This should be compared to *risk*, which is the probability that an unfavorable event occurs (i.e., in this context making the wrong decision). The concept of risk thus assigns a utility or value to an event, whereas uncertainty is free from such valuation. "Uncertainty causes risk which is handled by mitigation and results in outcomes" [2].

Two kinds of uncertainties are very common in engineering. One is *imprecision,* i.e. the exact value of a parameter is unknown (but a range or approximation is known). This typically occurs as a result of insufficient measurement equipment. The other common type is *variability* (or *aleatory* uncertainty) which denotes variations in parameter values between different instances due to a random factor [1].

Another dimension of uncertainty relates to what can be done about it. Sometimes there is a *reducible* uncertainty (alternatively referred to as *epistemic* uncertainty). This is an uncertainty that can be removed or at least reduced by spending an effort in collecting more information. However, there is always a balance if the effort of finding out merits the value of the added knowledge. The opposite is called *irreducible* uncertainty that cannot possibly be discovered [1]. As an example, it is an irreducible question if it will rain on a certain spot on Earth tomorrow (only prognoses are possible), but it is a reducible question if it rained there yesterday.

Variability is often described by statistical distributions, capturing properties of a large population of objects. This is sometimes referred to as the *frequentist* interpretation of probabilities. However, sometimes distributions are also used to describe one's beliefs in some event, such as our certainty that it will rain tomorrow. This is called *subjective* probabilities, and is useful in situations where statistics is meaningless (such as when talking about unique events rather than a large number of similar events). Many tools for statistical analysis that are based on probability distributions are also useful to reason about subjective beliefs.

So far, we have discussed uncertainties in the value of a known factor. Another level of uncertainty is whether all the relevant factors have actually been found, or if there are yet unknown factors that should also be considered. An example of this is the uncertainty if all relevant requirements have actually been elicited. Such factors are sometimes referred to as *unknown unknowns*.

3 Causes for Uncertainty

Architecting is characterized by many parallel activities and only few firm decisions. There are a number of factors that contribute to a certain level of inherent uncertainty. Some of them are caused by the nature of the development process and methods:

- *Sequential decision making* where the first decisions must be made under uncertainty regarding the consequences on later decisions.
- *Parallel decision making*. Several teams are refining their systems in parallel, and they must all make decisions based on uncertain knowledge about what decisions the neighboring teams will make in the future.

- *Expert judgment*. Sometimes, decisions need to be made based on experts giving imprecise statements such as "large", "fast", or "unlikely".
- *Abstract models*. It is a common engineering practice to build models of a future system. However, such models are always simplifications of reality, and factors that are believed to have minor influence are often removed.

Other uncertainties are due to the nature of the products. Three categories of noise, i.e. variability that cause quality loss, have been identified:

- *Environment*, or *outer noise,* includes changes to the environment, operating conditions, and effects of the system's interactions with different people. For architecting, only limited statistical data is available about the environment due to finite sampling. The environment available for investigation at the time of decision may vary from the one at the time of deployment.
- *Aging*, or *inner noise*. For physical products, aging leads to wear and deterioration, and this is true for the physical parts of the embedded system. For the software, architects may need to consider changes done over the system's installed life that can lead to variations.
- *Manufacturing*, or *product, noise*. Again, this is evident for the physical parts, and often there is a need to also include mechanisms in software (such as calibration or feedback) to control the effects of part-to-part variations in material and dimensions that are hard, or too expensive, to fix.

All these causes of uncertainty are general for engineering of complex systems, and are particularly strong during architecting. Two special cases of environment noise are of particular interest for architects. When the system of interest is a subsystem in a larger product, as is often the case for software, unintended interaction with neighboring subsystems often take place and cause noise that falls in this category. Also, uncertainties about the market for the products spill over on the architects, such as what functionality and variants will be requested by different customers.

4 Mitigation

We will now discuss a number of "tactics" that can be employed to reduce the unavoidable uncertainties and mitigate risks in architecting.

- *Focus on evolution rather than revolution*. When a new system is developed from scratch, everything is uncertain and remains so for a long time. By instead focusing on step-wise evolution of the architecture, only a few unknowns at a time must be dealt with, and effects are quickly seen.
- *Create feedback loops*. In architecting, feedback from downstream engineering is needed to quickly see when estimates prove to be wrong.
- *Track quality attributes*. Many architects focus on functionality, and have little data on actual values on the architecture's quality attributes. By continuously measuring attributes, it becomes possible to base decisions on trends and extrapolation from real data on things like cost and performance.

- *Make uncertainty explicit*. Often, engineers use point data in their estimates. By instead describing estimates using probability distributions, analyses such as Monte Carlo simulation can provide a richer picture of alternative scenarios, complemented with sensitivity analyses.
- *Divide and conquer using tolerances*. To ensure that the desired state of the system-wide quality attributes are reached during subsystem development, tolerances for attributes should be made explicit for each subsystem. The tolerances can give architects early warnings of problems later on.
- *Perform cost-benefit analysis of added certainty*. Usually, reducing uncertainty comes at a cost for performing additional analysis, and it should be considered whether buying more information in this way really changes the decisions being made.
- *Use options thinking*. A difficult trade-off for architects is between product cost now, and future flexibility, and real options can be used for this.
- *Apply robust design principles*. In situations where it is not practical to reduce uncertainty, the solution should be protected through robustness.
- *Use wide sampling and saturation*. Dealing with unknown unknowns is a particular challenge. One remedy is to gather information from a wide range of sources. Saturation should be tracked over time to see when the rate of new information slows down, and investigation can stop.

5 Conclusions

As we see it, uncertainty is a fact of life for architects of complex systems. However, we also believe that a deeper understanding of the nature of this uncertainty can help reducing the associated risks and improving the efficiency of the architecting process.

Apart from new evaluation methods and models that take uncertainty into account, there is also a need to review the current practices when it comes to the architecting process and the role of the architect. Too much effort is often spent on trying to get better information about events that are by necessity occurring in the future, and this search is usually fruitless. Instead, more focus should be placed on quality attributes and their relations, and finding architectures that can handle various scenarios.

It is also important to always bear in mind when looking for a solution that all additions to the architect’s work come at a cost, and future research must focus on evaluating that the benefit is really motivating this additional spending.

References

1. Aughenbaugh, J.: Managing uncertainty in engineering design using imprecise probabilities and principles of information economics. PhD thesis, Georgia Inst. of Tech. (August 2006)
2. McManus, H., Hastings, D.: A Framework for Understanding Uncertainty and Its Mitigation and Exploitation in Complex Systems. In: Proc. 15th Symposium of the International Council on Systems Engineering, INCOSE (July 2005)

Runtime Performance Management of Information Broker-Based Adaptive Applications

Anu Purhonen and Sakari Stenudd

VTT Technical Research Centre of Finland, Oulu, Finland
firstname.lastname@vtt.fi

Abstract. The increasing number of devices that surround us in everyday life requires additional means to handle the information overload they cause. In addition to the heterogeneity of devices, the smart environment is challenging because of user mobility, fluctuating resources and changing user needs. In this kind of dynamic environment, the applications need to be adaptive in order to maintain the user-perceived quality at the required level. This work proposes a solution for runtime performance management in a smart environment, where devices exchange information using semantic information brokers.

Keywords: self-management, ontologies, smart environment.

1 Motivation and Background

In a smart environment, the technologies should disappear into the background [1]. Our environment may include devices from various domains such as home appliances, building automation, and personal mobile devices. In addition to heterogeneity, the smart environment is challenging because of dynamism of the environment, such as user mobility, fluctuating resources and changing user needs. In this kind of dynamic environment, it is not possible to guarantee the quality of the operation at design time. Consequently, the applications need to be able to adapt to these changing conditions in order to maintain the user-perceived quality at an acceptable level.

The interoperability platform (IOP) [2], [3], [4] is a solution for interoperability of heterogeneous devices. It is realised by Semantic Information Brokers (SIB) that are used for sharing information between agents at heterogeneous devices. A smart space application (SSA) consists of at least one agent (consumer) that performs some useful service to a user and requires co-operation with at least one agent (producer) in order to realize that service. The specification of IOP does not address runtime quality management that is needed to be able to fulfil the needs of dynamic environments. This work addresses the performance management of applications utilising IOP.

The purpose of the runtime performance management (RPM) is to provide the mechanisms for the self-management of adaptive applications facing timing requirements and resource constraints. Self-management is realised using the MAPE-K loop [5] that consists of monitoring, analysis, planning and execution phases, which all use a common knowledge of the system and its environment. In general, performance engineering ensures that the timing requirements of applications are met

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 203–206, 2011.

within the resource constraints enforced by the underlying platform. Runtime performance management may be needed when (1) applications are downloaded onto devices where they are not necessarily tested before, (2) the overall performance is a result of co-operation of devices that are emerging and disappearing, and (3) the user preferences may change at runtime.

2 Research Questions

The purpose of the runtime performance management is to maintain performance of adaptive applications in a smart environment. The research questions for the RPM system are summarised as follows:

- **Q1. How to take into account devices that are not known at the development time?** The composition of emerging and disappearing devices cannot be predicted. Different manufacturers rely also on diverse technologies in their devices. Furthermore, different application domains may have diverse standards they have to follow.
- **Q2. How to let the application user decide if she wants to have performance management or not?** In order to increase reusability of the application components it should be up to the user of the smart application if she wants the application to be adaptive or not.
- **Q3. How to utilize devices that do not support performance management themselves?** It is up to the manufacturer to decide what features a device possesses. However, they may provide anyway useful service.
- **Q4. How to make RPM scalable from resource-poor to resource-rich devices?** The amount of resources on the devices varies so the RPM system should be able to support applications in devices with varying capabilities.
- **Q5. How to make RPM scalable to the size of the application.** In addition to restricted applications, performance management should be able to handle large applications that need multiple measurements and complicated reasoning.

3 Runtime Performance Management of Information Broker-Based Smart Environment

If a SSA is not useful unless it meets the performance requirements set to it then runtime performance management is needed. The runtime performance management consists of performance agents and the information that is transferred between the agents (see **Fig. 1**).

The quality information includes advertised, required, and measured qualities. Advertised quality is the advertised performance values of the produced information. Required quality refers to the performance requirements to the consumed information. Measured quality is the actually provided performance of the information. Additionally, there could be predicted quality that would give an estimation of the performance level in the future. Notification of quality violations, either measured or predicted, is used for triggering adaptations. Like any other information stored in and retrieved from SIB the exchanged quality information follows, the defined quality ontology.

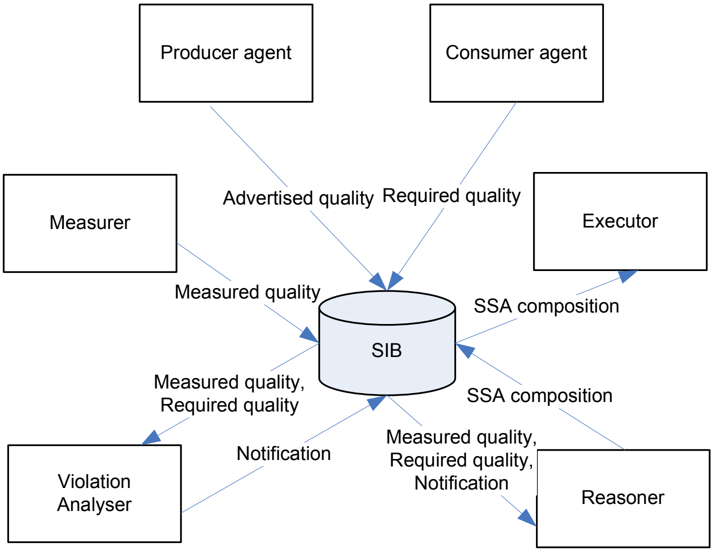


Fig. 1. Quality information exchanged during runtime performance management

Runtime performance management consists of four types of agents that realize the MAPE-K loop. **Measurers** produce values to the base or derived measures of performance attributes. The basic aggregation activities for the raw measurements are also included in this category. Measures are always related to a particular instance of information. On the other hand, information may have one or more measurers attached.

Analyser refers to all the activities that are required for generating the status of the performance from the base or derived measures. The analysers perform, for example, violation analysis and performance prediction. Violation analyser finds out if the measured or predicted quality does not fulfil the required quality of a performance measure and makes notifications of violations.

Reasoner decides the composition of agents that best fulfils the smart space application requirements. The quality requirements of SSA are budgeted into quality requirements for individual agents. Each consumer agent could have a reasoner attached to it. Reasoner searches for suitable candidates that could produce the information that the consumer needs and selects the one that best suits the performance requirements. It also updates the SSA composition in case the situation changes.

Executor is the mechanism that is capable of realizing the adaptation activities required by the SSA composition including activating and deactivating agents. The actual implementation depends on the platforms that the co-operating agents are running on.

We experimented with the IOP and quality management in a demonstrator that consisted of a greenhouse with sensors, actuators and interfaces for the gardener and customers [6]. This work elaborates the interfaces and division of responsibilities of the co-operating agents as well as defines the first set of supporting ontologies needed in runtime performance management. The RPM ontology relates the quality properties (e.g. provided, measured) to the IOP concepts (e.g. agent, information). The prototypes made indicate that the selected approach is a viable solution for runtime performance management in a semantic information broker-based smart environment. A summary of the results related to research questions:

- The communication of information broker based applications happens via SIBs. Consequently, also the quality information is exchanged via SIBs facilitating performance management also in dynamic and heterogeneous environment. (Q1)
- Like other agents in a smart space, performance management agents are separated from application-specific agents so that performance management is an optional and adaptable feature. (Q2)
- The reasoner makes the decision which information producer best fits the requirements of the consumer. Consequently, it can also select information without any quality properties in case that fits the requirements. (Q3)
- The performance management agents have each their own role and they do not need to be located at the same device. What kind of measurement and processing is needed depends on the application. The composition of the performance management can be distributed making it easier for resource-poor devices. (Q4)
- Each consumer may have its own reasoner. Consequently, when each consumer has its own performance management then performance management is actually distributed allowing handling of large applications. (Q5)

Acknowledgements. This work has been carried out in TIVIT-DIEM project – partially funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

References

1. Weiser, M.: The Computer for the 21st Century. Scientific American (International Edition) 265, 66–75 (1991)
2. DIEM/TIVIT project, `http://www.diem.fi/`
3. Sofia/Artemis project, `http://www.sofia-project.eu/`
4. Smart-M3, `http://sourceforge.net/projects/smart-m3/`
5. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36, 41–50 (2003)
6. Evesti, A., Eteläperä, M., Kiljander, J., Kuusijärvi, J., Purhonen, A., Stenudd, S.: Semantic Information Interoperability in Smart Spaces. In: 8th International Conference on Mobile and Ubiquitous Multimedia, pp. 158–159 (2009)

Reference Architecture and Product Line Architecture: A Subtle But Critical Difference

Elisa Yumi Nakagawa[1], Pablo Oliveira Antonino[2], and Martin Becker[2]

[1] Dept. of Computer Systems, University of São Paulo - USP
PO Box 668, 13560-970, São Carlos, SP, Brazil
elisa@icmc.usp.br

[2] Fraunhofer Institute for Experimental Software Engineering
Fraunhofer-Platz 1, 67663, Kaiserslautern, Germany
{pablo.antonino,martin.becker}@iese.fraunhofer.de

Abstract. Currently, the size and complexity of software systems, as well as critical time to market, demand new approaches from Software Engineering discipline for building such systems. In this context, the use of reference architectures and product line architectures is becoming a common practice. However, both of these concepts are sometimes mistakenly seen as the same thing; it is also not clearly established how they can be explored in a complementary way in order to contribute to software development. The main contribution of this paper is to make a clear differentiation between these architectures, by investigating and establishing definitions for each of them. Based on this, we also propose the use of reference architectures as a basis for product line architectures. As a result, a better understanding of both reference architectures and product line architectures, as well as an understanding of how to explore them jointly, can contribute to promoting more effective reuse in the development of software systems.

1 Introduction

In the face of increasing complexity, diversity, scope, and size of software systems, as well as the necessity of dynamic integration and adaptation of systems and the challenges encountered by managing families of systems, new ways to facilitate the development and evolution of such systems are required. Software Engineering has proposed a number of different ways for that. In particular, software architectures are being increasingly investigated as the main artifact that plays a pivotal role in determining system quality, since they form the backbone of any successful software-intensive system. Motivated by that, the research area of Software Architecture has grown up and has accumulated important knowledge that has contributed to facilitating the achievement of software quality aspects. In this context, two special types of software architecture can be found: Reference Architecture and Product Line Architecture. Both aim at improving software system development by standardizing the architectures of a set of software systems. In general, a reference architecture has been considered

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 207–211, 2011.

as a structure that, besides aggregating behavior, is the basis from which the software architectures of systems of a given domain are built. Considering the relevance of reference architectures, various application domains, such as automotive, avionics, and robotics, have proposed and used these architectures. In parallel, product line architectures have been used in the Software Product Line (SPL) approach to enable reuse in the large. This approach aims to efficiently derive specific products for a given domain based on reusable core assets that have common features and variable parts [1]. In this context, the product line architecture refers to a structure that also encompasses the behavior from which software products are developed. It is important to highlight that more and more organizations are adopting SPL[1] and, as a consequence, product line architectures, as a solution to improve time to market, productivity, flexibility, and mass customization needs. Despite the fact that reference architectures and product line architectures have been widely discussed and used, there exists no consensus within the software architecture community about the effective relationship between them. Besides that, since they seem to be similar, they are sometimes considered to be the same thing. It is also not clear how these architectures can be explored together in order to make an effective contribution to software development. Moreover, understanding them well seems to be relevant for the establishment and evolution of such architectures.

The main objective of this paper is to make a clear distinction between reference architecture and product line architecture. For this, we have investigated and established definitions for both architectures. We based our work on the more influential and recent works in the software architecture literature, as well as on our experience in proposing, using, and managing these architectures. The main results of our work is that, despite common characteristics between them, they present a subtle but critical difference. Considering the particular characteristics of each, we also propose the use of reference architectures as a basis for product line architectures.

2 Reference Architecture and Product Line Architecture

In spite of the diversity of available reference architectures and the general understanding regarding what a reference architecture is, there exists no consensus about its definition, as also stated in [2]. Regarding product line architecture, on the other hand, we have observed that its definition is more consolidated. In the follow, we will discuss definitions of both.

Reference Architecture: A study of the main works in the software architecture literature allow us to identify some important definitions for reference architecture. One of the first ones was stated by Kruchten [3], who said that "A reference architecture is, in essence, a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed and proven for use in particular business and technical contexts, together with supporting

[1] http://www.sei.cmu.edu/productlines/casestudies/index.cfm

artifacts to enable their use. Often, these artifacts are harvested from previous projects." Another definition was proposed by Bass et al. [4] and states that "A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them." A reference model can be considered as an abstract framework that presents a minimal set of unifying concepts, axioms, and relationships within a particular problem domain, independent of specific standards, technologies, implementations, or other concrete details. From the perspective of a specific domain, Rosen et al. [5] said that "A reference architecture is a working example of a critical aspect of your enterprise architecture, such as (...) how to work with your organization's message bus or (...) how to work with your business rules engine." In the same perspective, Angelov et al. [2] stated that "A reference architecture is a generic architecture for a class of information systems that is used as a foundation for the design of concrete architectures from this class." To complete these definitions, Reed[2] thinks that "A reference architecture consists of information accessible to all project team members that provides a consistent set of architectural best practices." Even though syntactically different, these definitions present the same essence: the reuse of knowledge about software development in a given domain, in particular with regard to architectural design. Other definitions are also found, but the ones presented above are sufficient for our purpose. Based on these works, we believe that a reference architecture refers to an architecture that encompasses the knowledge about how to design concrete architectures of systems of a given application domain; therefore, it must address the business rules, architectural styles (sometimes also defined as architectural patterns that address quality attributes in the reference architecture), best practices of software development (for instance, architectural decisions, domain constraints, legislation, and standards), and the software elements that support development of systems for that domain. All of this must be supported by a unified, unambiguous, and widely understood domain terminology.

Product Line Architecture: Various synonyms are used for the term "product line architecture", such as software product line architecture, domain-specific software architecture, domain architecture, and even reference architecture in [6,7]. However, it seems that "product line architecture" is a term that fits best in the SPL context. In parallel, there are also different definitions of this term. DeBaud et al. [7] say that it is an architecture with a required degree of flexibility and is shared by all the members of a product line, ensuring their conceptual integrity. According to Pohl et al. [6], "product line architecture is a core architecture that captures the high level design for the products of the SPL, including the variation points and variants documented in the variability model." In the same perspective, Gomaa [8] stated that "product line architecture is an architecture for a family of products, which describes the kernel, optional, and variable components in the SPL, and their interconnections." In a more complete definition, SEI[3] declares that "The product line architecture is an early and prominent

[2] http://www.ibm.com/developerworks/rational/library/2774.html

[3] http://www.sei.cmu.edu/productlines/frame_report/arch_def.htm

member in the collection of core assets. (...) The architecture defines the set of software components (...) that populates the core asset base. The product line architecture – together with the production plan – provides the prescription (...) for how products are built from core assets". It is worth highlighting that, in general, others definitions are derivations of those presented herein. Moreover, it is important to observe that the common essence in these definitions is "variability on a common core". Thus, we can define product line architecture as a special type of software architecture used to build a product line; it explicitly describes commonality and variability and is the basis for the architectures of all product line members.

3 Conclusion and Future Work

The contribution of this paper is a better understanding of what differentiates reference architectures and product line architectures. While reference architectures deal with the range of knowledge of an application domain, providing standardized solutions for a broader domain, product line architectures are more specialized, focusing sometimes on a specific subset of the software systems of a domain and providing standardized solutions for a smaller family of systems. Another essential difference is that product line architectures are concerned with the variabilities among products. Furthermore, reference architectures are generally on a higher level of abstraction compared to product line architectures. We have therefore first investigated reference architectures as a basis for product line architectures. In future work, we will also investigate how reference architectures can be explored in the development of other product line artifacts, such as the variability model. Thus, we intend to achieve reuse of the domain knowledge contained in the reference architecture, as well as time and effort reduction and productivity improvement, when establishing a SPL.

Acknowledgments. This work is supported by the Brazilian funding agencies FAPESP, CNPq, and CAPES and partially supported by the OptimAAL-Project (Competence Platform for the development and introduction of AAL-Solutions), funded by the German Federal Ministry of Education and Research (BMBF), and the CESAR-Project (Cost-efficient methods and processes for safety relevant embedded systems), funded by the European ARTEMIS Joint Undertaking. We would like to thank Sonnhild Namingha from Fraunhofer IESE for linguistic support.

References

1. Clements, P., Northrop, L., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)
2. Angelov, S., Grefen, P.W.P.J., Greefhorst, D.: A classification of software reference architectures: Analyzing their success and effectiveness. In: WICSA 2009, Cambridge, UK, pp. 141–150 (September 2009)

3. Kruchten, P.: The Rational Unified Process: An Introduction, 2nd edn. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (2000)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (2003)
5. Rosen, M., Ambler, S.W., Hazra, T.K., Ulrich, W., Watson, J.: Enterprise architecture trends. Enterprise Architecture 10(1) (2007); Cutter Consortium
6. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
7. DeBaud, J.M., Flege, O., Knauber, P.: PuLSE-DSSA - a method for the development of software reference architectures. In: ISA 1998, Orlando, USA, pp. 25–28 (1998)
8. Gomaa, H.: Designing Software Product Lines with UML. Object Technology Series. Addison-Wesley, Reading (2004)

Dynamically Reconfigurable Resource-Aware Component Framework: Architecture and Concepts

Bojan Orlic, Ionut David, Rudolf H. Mak, and Johan J. Lukkien

Eindhoven University of Technology,
Eindhoven, The Netherlands
{b.orlic,i.david,r.mak,j.j.lukkien}@tue.nl

Abstract. Applications executed on a shared distributed platform compete for resources provided by the platform. In case these applications have highly fluctuating resource demands, a software architecture is required that provides support for runtime resource management. In position paper [1], we have proposed such architecture and have introduced its key concepts and entities. In this paper, we introduce a metamodel that captures the key concepts and we identify lifecycle models for both applications and individual components. A set of dynamic reconfiguration strategies is introduced and their relationship to the stages of the application lifecycle is given.

Keywords. Component framework, networked services, resource management, dynamic reconfiguration, application lifecycle, component lifecycle.

1 Introduction

The key idea behind the component-based software engineering (CBSE) approach to software development is to enable 3rd party composition in the process of system engineering and in that way facilitate building complex systems from predefined building blocks [2]. Composition of components is supported by component frameworks which address the component model, as well static (design time) and dynamic (run time) aspects of composition. We are interested in CBSE for distributed systems with limited resources, where resources concern processing power, memory capacity and network bandwidth. Resource management, however, is rarely addressed by a software component framework, and if it is, it is limited to static resource reservations. We target applications with highly fluctuating resource demands (e.g. video processing applications) in which static reservations are not suitable as they result in inefficient resource usage. For the combination of efficient runtime resource management and dynamic component composition a special architecture is needed. In this paper we further develop the architecture proposed in [1]. We choose to realize applications using the SOA style by connecting networked services, which allows us to include structural reconfiguration in our resource management strategies.

2 Architecture Description

Figure 1 gives a metamodel that specifies key abstractions of our architecture. The system layer contains a single resource manager that is in charge of all resources

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 212–215, 2011.

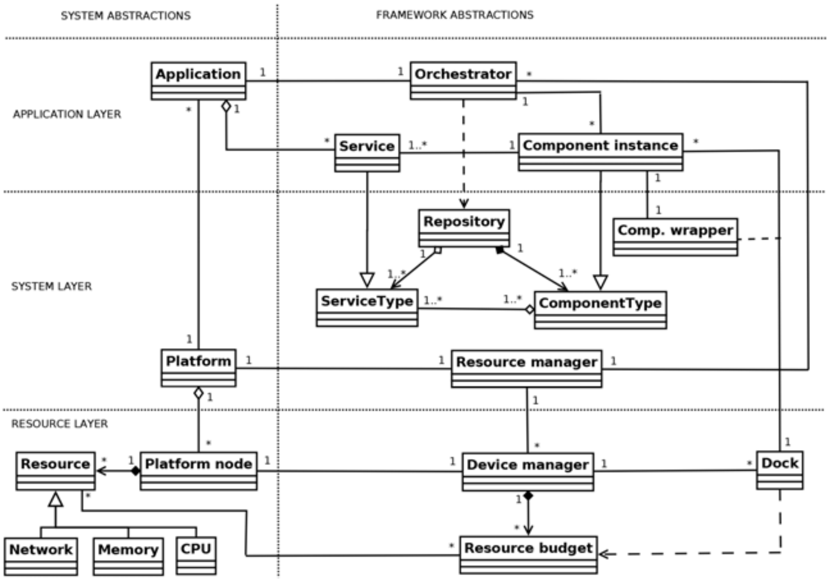


Fig. 1. Metamodel defining basic concepts, entities and interconnection mechanisms

provided by the nodes of the distributed platform. To perform its management tasks it deploys the services of local device managers, as shown in the resource layer. These services comprise installation and instantiation of components, monitoring their resource usage, and enforcement of resource reservations (budgets). In the application layer, orchestrator entities are responsible for the composition, deployment and operation of an application. Each application in the system will have a dedicated orchestrator. Since our applications are built from reusable and discoverable components whose instantiation, composition, and deployment can be performed in the SOA style, the system layer of our architecture also contains a repository of services and corresponding components that are available in the system. Applications are composed from services through binding provided and required interfaces. Docks are units of the deployment of the system that can host one or more components.

The lifecycles of applications and components, depicted in Figure 2, involve two main stakeholders – application designer and application operator. A designer can create applications either ad-hoc at runtime or he can define and save an application as a set of possible configurations. In the latter case, the operator chooses when to start the application, and the actual configuration is selected at runtime through negotiation between the resource manager, the orchestrator and the operator. The application lifecycle consists of composition, deployment and operation phases. In general, resource management decisions taken in the operation phase (marked 1 to 5) require revision of activities in the other two phases (steps A to E). In step A, the application designer composes an application from services available in the repository. In step B, he selects components that implement those services. During the composition phase, the application designer is also allowed to enter constraints for steps of the deployment phase, such as mapping of components to nodes (C), setting of QoS levels (D) and allocation of budgets to components (E). Thus, prior to runtime, it is possible to specify a number of different configurations as well as the reconfiguration strategy (the logic that stipulates when to select which configuration).

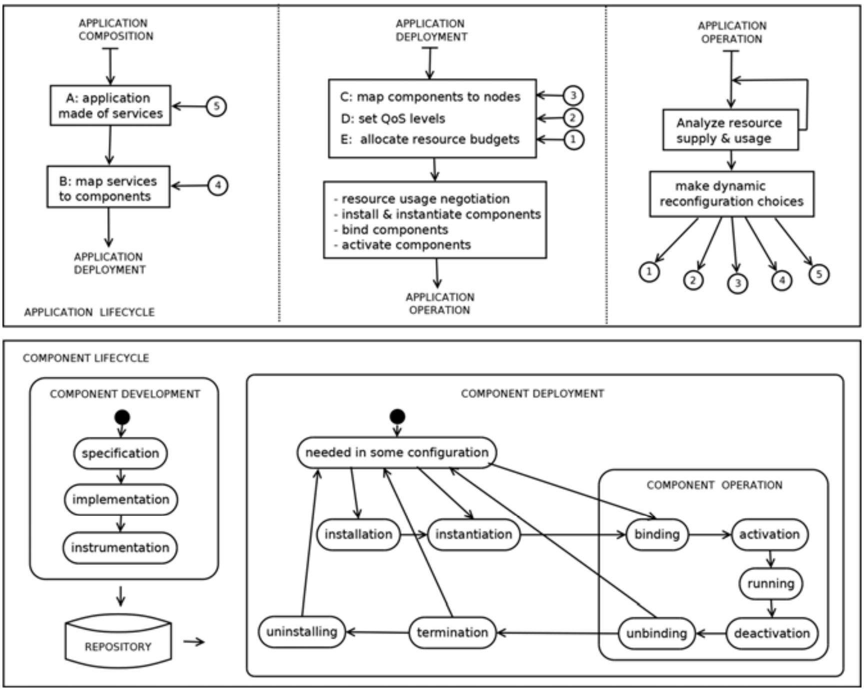


Fig. 2. Application and component lifecycles

In the operation phase, resource supply and usage is monitored and analyzed. If needed, dynamic reconfiguration is applied. In order of increasing severity we distinguish: 1) Realigning resource budgets. 2) QoS adaptation, in which service levels are adjusted to resource availability. It includes realigning local resource reservations and resolving dependencies between QoS levels of an application's components, e.g., the frame rate. 3) Reallocation, where components are migrated to nodes that better fit their resource demands. 4) Replacement of components by others that offer the same service but at a different resource demand. 5) Reconstruction of application structure, in which services are added, replicated, replaced, or removed.

The component lifecycle is specific to our framework and in accordance with the application lifecycle. The component development phase includes specification, implementation and instrumentation. The outcome of the instrumentation is a component deployable on a specific OS and compatible with our framework. Such a wrapped component is stored in the repository. The component deployment phase starts when a component is needed in some application and includes activities such as installation, instantiation, termination and uninstalling. In the component operation phase, the instantiated component can undergo binding, activation, deactivation and unbinding.

3 Conclusions and Future Work

In this paper, we have described a resource-aware component based architecture. Similar to others we identify three layers [3], and distinguish between local and global resource management [4], [5] and [6]. As in [7] and [8] resource management is combined with plugin framework and SOA technologies. In contrast to all frameworks cited, our framework allows structural changes as part of resource management strategies, and defines application and component lifecycles.

Current work involves implementing the framework and definition of models for resource usage prediction. Future work includes introducing resource management policies and assigning those activities to framework entities.

References

1. David, I., Orlic, B., Mak, R., Lukkien, J.J.: Towards Resource-Aware Runtime Reconfigurable Component-Based Systems. In: Proceedings of the 6th World Congress on Services, SERVICES 2010, Miami FL, USA, pp. 465–466 (2010)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
3. Koulamas, C., Prayati, A., Papadopoulos, G.: A Framework for the implementation of Adaptive Streaming Systems. In: Proceedings of the 3rd ACM Workshop on Wireless Multimedia Networking and Performance Modeling (WMuNeP 2007), pp. 23–26 (2007)
4. Sachs, D.G.: A New Framework for Hierarchical Cross-layer Adaptation, Ph.D. dissertation, University of Illinois at Urbana-Champaign (2006)
5. Rizvanovic, L., Fohler, G.: The MATRIX: A Framework for Streaming in Heterogeneous Systems. In: RTMM - International Workshop on Real-Time for Multimedia, Italy (2004)
6. Kersten, B., van Rens, K., Mak, R.: ViFramework: A framework for networked video streaming components. In: Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2011 (2011)
7. Korostelev, A., Lukkien, J.J., Nesvadba J., Qian Y.: QoS Management in Distributed Service Oriented Systems. In: Parallel and Distributed Computing and Networks (2007)
8. Chen, S., Lukkien, J.J., Verhoeven, R., Vullers, P., Petrovic, G.: Context- Aware Resource Management for End-to-End QoS Provision in Service Oriented Applications. In: Proceedings of the Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments (SUPE 2008), pp. 1–6 (2008)

A Reusable Business Tier Component with a Single Wide Range Static Interface

Oscar M. Pereira[1], Rui L. Aguiar[1], and Maribel Yasmina Santos[2]

[1] Instituto de Telecomunicações, University of Aveiro
3810-193 Aveiro, Portugal
{omp,ruilaa}@ua.pt
[2] Algoritmi Research Center, University of Minho
4800 Guimarães, Portugal
{maribel}@dsi.uminho.pt

Abstract. This research proposes an architecture for reusable components aimed at bridging the object-oriented and the relational paradigms. The component, referred to here as Business Tier Component, provides a single wide range static interface able to manage a set of Create, Read, Update and Delete (CRUD) expressions, deployed at runtime and of any complexity, on behalf of application tiers. The only constraint is that the required interface to manage each CRUD expression must be a super-interface of the provided wide range interface. The main research challenge of this paper is the definition of an architecture for reusable components aimed at managing dynamically a set of CRUD expressions, deployed at runtime, on behalf of application tiers.

Keywords: reusable component, business tier, databases, impedance mismatch.

1 Introduction

Object-oriented and relational paradigms are simply too different to bridge seamlessly, leading to a set of difficulties informally known as *impedance mismatch* [1], which is an open issue for more than 50 years [2]. To tackle the impedance mismatch, several solutions have been proposed, including Call-Level Interfaces (CLI) [3, 4], Embedded SQL [5], object-to-relational mapping techniques (O/RM) [6, 7], language extensions [7] and persistent frameworks [8, 9]. These solutions are mostly tailored to cope with the principle of "new project implies new development process" staying in opposite to approaches based on component-based software engineering (CBSE) [10]. CBSE is a sub-discipline of software engineering aimed at promoting the reuse of components to build software systems.

This paper presents the basis for a reusable-component-based approach aimed at bridging the object-oriented and the relational paradigms hereafter known as Business Tier Component (BTC). Each BTC is developed to provide a single wide range static interface able to accommodate a set of CRUD expressions that are needed to implement a business tier. Among their main features, each BTC has the capacity to accept, at runtime, the information regarding the business logic to be implemented.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 216–219, 2011.

This paper is structured as follows: Section 2 presents the Business Tier Component and Section 3 presents the conclusions.

2 Business Tier Component

At development time, each BTC is built to implement a service interface known as a wide business tier interface (BTI) and have no knowledge about the CRUD expressions they will have to manage. CRUD expressions are only deployed at runtime. The only constraint is that each CRUD expression must only require any sub-set of the implemented BTI services. In order to avoid any maintenance activity, the development of BTC must be carefully planned to forecast all current and future services to be implemented by the BTI. BTI comprises four distinct types of interfaces (ISelect, IInsert, IUpate, IDelete), see Fig. 1, each one tailored to deal with one specific type of CRUD expression, Select, Insert, Update and Delete, respectively.

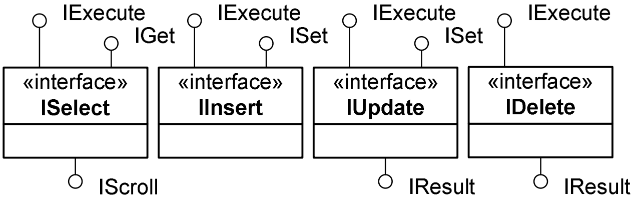


Fig. 1. Interfaces types

Fig. 2 provides additional detail to help the understanding of each interface type: IGet interface implements services to read all the attributes from the returned relations; IExecute interface provides a service to execute any CRUD expression with any number of clause conditions parameters; ISet implements a service to set the values for the attributes used on any Insert and Update CRUD expression; IScroll interface implements services to scroll on returned relations; IResult implements services to get results derived from the execution of CRUD expressions. Interfaces IExecute, IResult, IScroll and ISet are shared by all BTC. IGet is the only interface that is specifically built for each BTC. The generalization achieved by IExecute and ISet interfaces relies on the strategy followed to define their arguments. Being an array of objects there is no restriction on the data type and number of arguments to be used. At runtime, if required, BTC checks their data type and use the correct methods to interact with the host relational database management system. Fig. 3 presents a simplified class diagram for BTC. Each interface type is implemented by a class herein know as Business Tier Entity (BTE): BTE_S, BTE_I, BTE_U and BTE_D for ISelect, IInsert, IUpdate and IDelete, respectively. BTC entry point is the public static method *createBTC*, which creates a new instance of a BTC and returns IBTC interface. From this interface actors may access BTC functionalities, namely for playing the two implemented roles: administrator and developer. The administrator role is played through the IAdm interface and is used to supervise and control the set of CRUD expressions to be made available in each BTC running instance. The

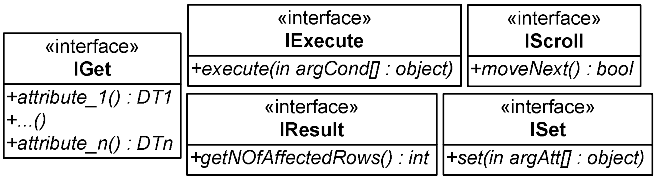


Fig. 2. Details of interfaces types

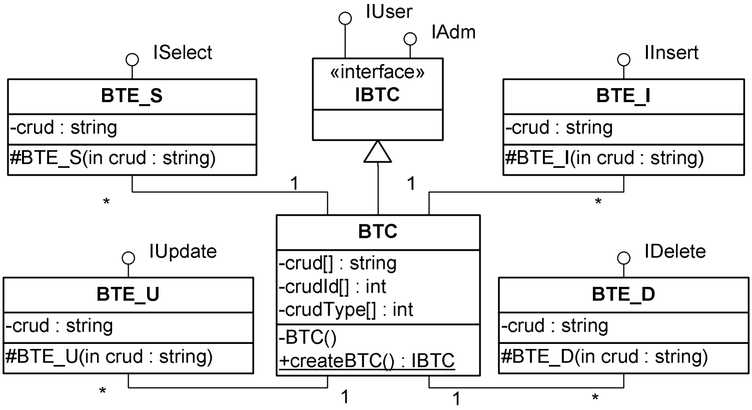


Fig. 3. Class diagram of BTC architecture

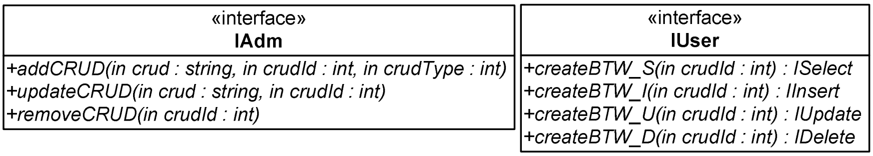

«interface»
IAdm
+addCRUD(in crud : string, in crudId : int, in crudType : int)
+updateCRUD(in crud : string, in crudId : int)
+removeCRUD(in crudId : int)
«interface»
IUser
+createBTW_S(in crudId : int) : ISelect
+createBTW_I(in crudId : int) : IInsert
+createBTW_U(in crudId : int) : IUpdate
+createBTW_D(in crudId : int) : IDelete

Fig. 4. IAdm and User interfaces

developer role is played through the IUser interface, which is used by developers to instantiate BTE and to execute CRUD expressions. Fig. 4 presents IAdm and IUser intefaces: 1) IAdm interface comprises 3 methods to manage the set of CRUD expressions to be made available in each BTC running instance. Thus, different running instances of the same BTC may manage a different set of CRUD expressions. This assertion emphasizes that the presented architecture promotes the reuse of BTC to build business tiers. 2) IUser interface comprises four methods: one for each sub-interface type. From Fig. 3 and Fig. 4 it is easily perceived that users select from the pool one of the CRUD expressions that are available, by their ids, and instantiates a BTE. This way, each BTE have no knowledge about the CRUD expressions to be managed until their instantiation. Moreover, each BTE may be reused to managed not one but any number of CRUD expressions, promoting this way the reuse of computation [11].

3 Conclusions

This paper presents the result of a research focused to define an architecture to reusable components aimed at bridging the object-oriented and the relational paradigms. The proposed architecture allows BTC to cope with the following features: 1) The source code to execute CRUD expressions is internally implemented by BTE, relieving, this way, programmers from writing the correspondent source code, 2) The defined model seamlessly translates the object-oriented and the relational paradigms tackling this way the impedance mismatch hindrance; 3) Pools of CRUD expressions are maintained in runtime revealing BTC capability to dynamically adapt to new CRUD expressions; 4) Each running instance of the same BTC may manage a different set of CRUD expressions promoting this way BTC reusability.

It is expected that the outcome of this research may contribute to open new perspectives to develop reusable components to build business tiers based on CRUD expressions.

Future work will be centered on extending the BTC architecture and then assess it, if possible, in a real scenario.

References

1. David, M.: Representing database programs as objects. In: Bancilhon, F., Buneman, P. (eds.) Advances in Database Programming Languages, pp. 377–386. ACM, N.Y (1990)
2. Cook, W., Ibrahim, A.: Integrating programming languages and databases: what is the problem? (May 2011), `http://www.odbms.org/experts.aspx#article10`
3. Microsystems, S.: JDBC Overview (May 2011), `http://www.oracle.com/technetwork/java/overview-141217.html`
4. Microsoft. Microsoft Open Database Connectivity (May 2011), `http://msdn.microsoft.com/en-us/library/ms710252VS.85.aspx`
5. Eisenberg, A., Melton, J.: Part 1: SQL Routines using the Java (TM) Programming Language. In: American National Standard for Information for Technology Database Languages - SQLJ, International Committee for Information Technolgy: dff (1999)
6. Christian, B., Gavin, K.: Hibernate in Action. Manning Publications Co. (2004)
7. Kulkarni, D., et al.: LINQ to SQL: NET Language-Integrated Query for Relational Data, Microsoft
8. Rusell, C.: JDO Specification, JSR-12 (June 2010), `http://jcp.org/aboutJava/communityprocess/final/jsr012/index.html`
9. Oracle. JPA - Java Persistent API (May 2011), `http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html`
10. Heineman, G.T., Councill, W.T.: Component-Based Software Engineering: Putting the Pieces Together, 1st edn. Addison-Wesley, Reading (2001)
11. Elizondo, P.V., Lau, K.-K.: A catalogue of component connectors to support development with reuse. Journal of Systems and Software 83(7), 1165–1178 (2010)

Reverse Engineering Architectural Feature Models

Mathieu Acher[1], Anthony Cleve[2], Philippe Collet[1],
Philippe Merle[3], Laurence Duchien[3], and Philippe Lahire[1]

[1] Université de Nice Sophia Antipolis - I3S (CNRS UMR 6070), France
{acher,collet,lahire}@i3s.unice.fr
[2] PReCISE Research Centre, University of Namur, Belgium
acl@info.fundp.ac.be
[3] INRIA Lille-Nord Europe, Univ. Lille 1 - CNRS UMR 8022, France
{philippe.merle,laurence.duchien}@inria.fr

Abstract. Reverse engineering the variability of an existing system is a challenging activity. The architect knowledge is essential to identify variation points and explicit constraints between features, for instance in feature models (FMs), but the manual creation of FMs is both time-consuming and error-prone. On a large scale, it is very difficult for an architect to guarantee that the resulting FM is consistent with the architecture it is associated with. In this paper, we present a comprehensive, tool supported process for reverse engineering architectural FMs. We develop automated techniques to extract and combine different variability descriptions of an architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. We illustrate the process when applied to a representative software system and we report on our experience in this context.

1 Introduction

As a majority of software applications are now large-scale, business-critical, operated 24/7, distributed and ubiquitous, their complexity is increasing at a rate that outpaces all major software engineering advances. To tame it, *Software Product Line* (SPL) engineering is one of the major trends of the last decade. An SPL can be defined as "*a set of software-intensive systems that share a common, managed set of features and that are developed from a common set of core assets in a prescribed way*" [8]. SPL engineering aims at generating tailor-made variants for the needs of particular customers or environments and promotes the systematic reuse of software artifacts. An SPL development starts with an analysis of the domain to identify commonalities and variabilities between the members of the SPL. It is common to express SPL variability in terms of features, which are domain abstractions relevant to stakeholders. A *Feature Model* is used to compactly define all features in an SPL and their valid combinations [18,10].

When SPL engineering principles are followed from the start, it is feasible to manage variability through one or more *architectural* feature models and then

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 220–235, 2011.

associate them to the architecture [17]. The major architectural variations are then mapped to given features, allowing for automated composition of the architecture when features are selected to configure a software product from the line. A resulting property of crucial importance is to guarantee that the variability is not only preserved but also kept consistent across all artefacts [9,4,15].

In many cases, however, one has to deal with (legacy) software systems, that were not initially designed as SPLs. When the system becomes more complex, with many configuration and extension points, its variability must be handled according to SPL techniques. In this context, the task of building an architectural feature model, is very arduous for software architects. It is then necessary to recover a consistent feature model from the actual architecture. On a large scale both automatic extraction from existing parts and the architect knowledge should be ideally combined to achieve this goal.

1.1 FraSCAti: The Need for Handling Variability

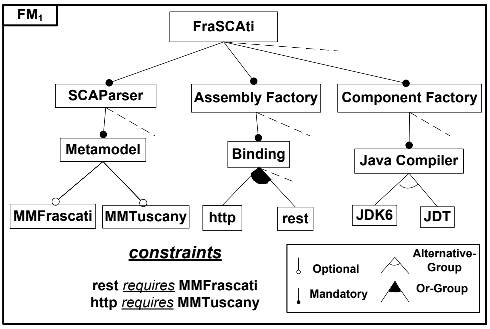
In this paper, we illustrate our proposal and report on a case study on the FraSCAti platform [12], an open-source implementation of the Service Component Architecture (SCA) standard [21], which allows for building hierarchical component architectures with the support of many component and service technologies.

Started in 2008, the development of FraSCAti begun with a framework based on a basic implementation of the standard, and then incrementally enhanced. After four major releases, it now supports several SCA specifications (Assembly Model, Transaction Policy, Java Common Annotations & APIs, Java Component Implementation, Spring Component Implementation, BPEL Client & Implementation, Web Services Binding, JMS Binding), and provides a set of extensions to the standard, including binding implementation types (Java RMI, SOAP, REST, JSON-RPC, JNA, UPnP, etc.), component implementation types (Java, OSGi, Java supported scripting languages, Scala, Fractal), interface description types (Java, C headers, WSDL, UPnP), runtime API for assembly and component introspection/reconfiguration [19]. As its capabilities grew, FraSCAti has itself been refactored and completely architected with SCA components.

With all these capabilities, the platform has become highly (re-)configurable in many parts of its own architecture. It exposes a larger number of extensions that can be activated throughout the platform, creating numerous variants of a FraSCAti deployment. For example, some variations consist in specific components bound to many other mandatory or optional parts of the platform architecture. It then became obvious to FraSCAti technical leads that the variability [22] of the platform should be managed to pilot and control its evolution as an SPL.

1.2 Feature Modeling

Variability modelling is a central activity in SPL engineering. We chose to rely on a particular kind of variability model, *Feature Models* (FMs), based on their wide adoption, the existence of formal semantics, reasoning techniques and tool support [18,10,6]. FMs compactly represent product commonalities and variabilities in terms of features. FMs can be used to describe features at different



(a) an excerpt of a possible architectural FM

$$\begin{aligned}
\phi_1 \ &= \ FraSCAti \\
&\wedge FraSCAti \Leftrightarrow AssemblyFactory \\
&\wedge FraSCAti \Leftrightarrow ComponentFactory \\
&\wedge FraSCAti \Leftrightarrow SCAParser \\
&\wedge SCAParser \Leftrightarrow Metamodel \\
&\wedge AssemblyFactory \Leftrightarrow Binding \\
&\wedge ComponentFactory \Leftrightarrow JavaCompiler \\
&\wedge JavaCompiler \Rightarrow JDK6 \vee \ JDT \\
&\wedge \neg \ JDK6 \vee \ \neg JDT \\
&\wedge MMFrascati \Rightarrow Metamodel \\
&\wedge MMTuscany \Rightarrow Metamodel \\
&\wedge http \Rightarrow Binding \\
&\wedge rest \Rightarrow Binding \\
&\wedge Binding \Rightarrow rest \vee http \\
&\wedge rest \Rightarrow MMFrascati \\
&\wedge http \Rightarrow MMTuscany
\end{aligned}$$

(b) propositional logic encoding

Fig. 1. Feature Model and Propositional Logic Encoding

abstraction levels, at different phases of the software lifecycle, and related to various software artefacts [9,14,4]. The FMs that we consider all along this paper express *architectural* variability.

An FM hierarchically structures features into multiple levels of detail. As an example, Fig. 1(a) shows a excerpt of the *architectural FM* of FraSCAti. As in typical SPLs, not all combinations of features or *configurations* (see Definition 1) are valid. When decomposing a feature into subfeatures, the subfeatures may be *optional*, *mandatory* or may form *Or* or *Alternative*-groups (e.g., $JDK6$ and JDT form an Alternative-group, *http* and *rest* form an Or-group). An additional mechanism to specify variability is to add constraints (expressed in propositional logic), which may cut across the feature hierarchy (e.g., *rest* requires $MMFrascati$). The validity of a configuration is determined by the semantics of FMs, e.g. $JDK6$ and JDT are mutually exclusive and cannot be selected at the same time. A valid configuration is obtained by selecting some features from parents to children while following the rules imposed by the operators (e.g., exactly one subfeature must be selected in an Alternative) and the constraints.

Definition 1 (SPL, Feature Model). *A software product line SPL_i is a set of products described by a feature model FM_i. The set of features of FM_i is denoted $\mathcal{F}_{FM_i}$. Each product of SPL_i is a combination of features and corresponds to a valid configuration of FM_i. A configuration c of FM_i is defined as a set of selected features $c = \ \{f_1, f_2, \ldots, f_m\} \subseteq \mathcal{F}_{FM_i}$. $[\![FM_i]\!]$ denotes the set of valid configurations of FM_i. We note ϕ_i the propositional formula of FM_i.*

The set of configurations represented by an FM can be described by a propositional formula defined over a set of Boolean variables, where each variable corresponds to a feature [10]. Figure 1(b) also shows the mapping of the FM to a propositional formula. The propositional formula can be used to automatically reason about properties of an FM (e.g., see [6]). In particular, if an assignment to its Boolean variables is satisfiable, then the selection/deselection of the corresponding features respects the rules evoked above.

1.3 Reverse Engineering FraSCAti as an SPL

In order to manage FraSCAti as an SPL, we needed to capture its variability from the actual architecture. Several software artefacts (SCA composite files, Maven descriptors, unformal documents) describe the FraSCAti architecture, but its variability is not explicitly represented. As the FraSCAti main *software architect* (SA) had an extensive expertise of the architecture and of its evolution, he was asked to model the architecture he has in mind with variation points (see left part of Fig. 2). As a domain expert, he had the ability to elicitate those variation points and explain the rationale behind them. We decided to separate the variability description from the architectural model itself. The idea is to represent the variation points of the architecture as features in an architectural FM, and then to related those features to the architectural elements.

The task of manually creating the architectural FM is daunting, time-consuming and error-prone, requiring substantial effort from the SA. In this case, as in all large scale architectures, it is very difficult for an architect to guarantee that the resulting FM is consistent with the architecture.

Another complementary approach consists of the automated extraction of the architectural FM from existing software artefacts (see right part of Fig. 2). This operation clearly saves time and reduces accidental complexity, but the accuracy of the results directly depends on the quality of the available documents and of the extraction procedure. This approach is notably followed in a recent work involving large scale variability extraction from the Linux kernel [20].

The main challenge is then to reconcile these two architectural FMs into a final FM being compatible with both the SA view and the actual architecture. It must also be noted that we could have tried to somehow *integrate* the SA knowledge in the extraction process or to let him edit an extracted FM, but we

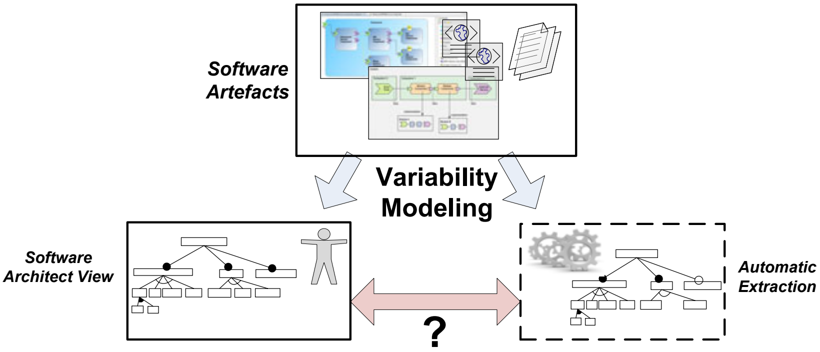


Fig. 2. Variability Modeling from Software Artefacts

argue that keeping the first two activities separated is better. This lets an highly experienced SA focus on her own variability scoping, and then compare it to the extracted version. Moreover, this allows for explicitly separating the required variability of the SA from the supported variability of the actual software system, as advocated in [16].

In this paper, we present a comprehensive, tool supported process for reverse engineering architectural FMs. We develop automated techniques to extract and combine different variability descriptions of an architecture. Then, alignment and reasoning techniques are applied to integrate the architect knowledge and reinforce the extracted FM. In the remainder of this paper we describe the automated extraction process that we have applied to FraSCAti FM(Section 2). We then show how the process is completed by refinement steps that enable the architect to compare and integrate her knowledge, with the aim to obtain a consistent architectural FM (Section 3). This process is validated by application to the FraSCAti architecture and some lessons learned are presented. A related work discussion is provided in Section 4 while Section 5 concludes the paper.

2 Automatic Extraction of Architectural Feature Model

Overview. Fig. 3 summarizes the steps needed to realize the process. First, a raw *architectural feature model*, noted $FM_{Arch_{150}}$, is extracted from a *150% architecture* of the system (see ①). The latter consists of the composition of the architecture fragments of *all* the system plugins. We call it a 150% architecture because it is not likely that the system may contain them all. Consequently, $FM_{Arch_{150}}$ does include all the *features* provided by the system, but it still constitutes an over approximation of the set of *valid combinations* of features of the system family. Indeed, some features may actually *require* or *exclude* other features, which is not always detectable in the architecture. Hence the need for considering an additional source of information. We therefore also analyze the specification of the system plugins and the dependencies declared between them, with the ultimate goal of deriving inter-feature constraints from inter-plugin constraints. To this end, we extract a *plugin feature model* FM_{Plug}, that represents the system plugins and their dependencies (see ②). Then, we automatically reconstruct the bidirectional mapping that holds between the features of FM_{Plug} and those of $FM_{Arch_{150}}$ (see ③). Finally, we exploit this mapping as a basis to derive a richer architectural FM, noted FM_{Arch}, where additional feature constraints have been added. As compared to $FM_{Arch_{150}}$, FM_{Arch} more accurately represents the architectural variability provided by the system.

2.1 Extracting $FM_{Arch_{150}}$

The architectural FM extraction process starts from a set of n system plugins (or *modules*), each defining an architecture fragment. In order to extract an architectural FM representing the entire product family, we need to consider *all* the system plugins at the same time. We therefore produce a *150% architecture*

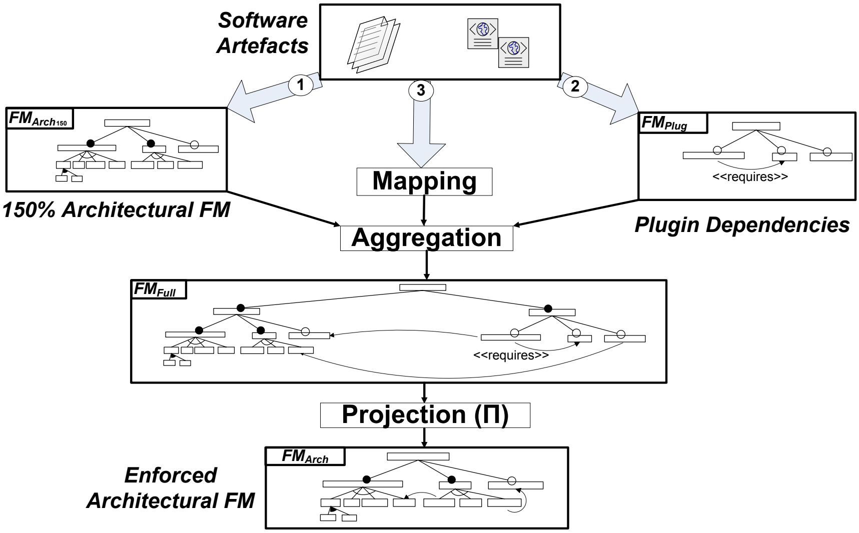


Fig. 3. Process for Extracting FM_{Arch}

of the system, noted $Arch_{150}$. It consists of a hierarchy of components. In the SCA vocabulary, each component may be a composite, itself further decomposed into other components. Each component may provide a set of *services*, and may specify a set of *references* to other services. Services and references having compatible *interfaces* may be bound together via *wires*. Each wire has a reference as *source* and a service as *target*. Each reference r has a *multiplicity*, specifying the minimal and maximal number of services that can be bound to r. A reference having a 0..1 or 0..N multiplicity is *optional*.

Note that $Arch_{150}$ may not correspond to the architecture of a *legal* product in the system family. For instance, several components may exclude each other because they all define a service matching the same 0..1 reference r. In this case, the composition algorithm binds only one service to r, while the other ones are left unbound in the architecture.

Since the extracted architectural FM should represent the *variability* of the system of interest, we focus on its *extension points*, typically materialized by *optional* references. Algorithm 1 summarizes the behavior of the FM extractor. The root feature of the extracted FM (f_{root}) corresponds to the main composite (*root*) of $Arch_{150}$. The child features of f_{root} are the first-level components of *root*, the latter being considered as the main system features. The lower-level child features are produced by the $AddChildFeatures$ function (Algorithm 2). This recursive function looks for all the optional references r of component c and, for each of them, creates an optional child feature f_r, itself further decomposed through a *XOR* or an *OR* group (depending on the multiplicity of r). The child features f_{c_s} of the group correspond to all components c_s providing a service compatible with r.

Algorithm 1. $ExtractArchitecturalFM_{150}(Arch_{150})$

```
Require: A 150% architecture of the plugin-based system ($Arch_{150}$).
Ensure: A feature model approximating the system family ($FM_{Arch_{150}}$).
$root \leftarrow MainComposite(Arch_{150})$
$f_{root} \leftarrow CreateFeature(root)$
$FM_{Arch_{150}} \leftarrow SetRootFeature(FM_{Arch_{150}}, f_{root})$
for all $c \in FirstLevelComponents(root)$ do
    $f_c \leftarrow CreateFeature(c)$
    $FM_{Arch_{150}} \leftarrow AddMandatoryChildFeature(FM_{Arch_{150}}, f_{root}, f_c)$
    $FM_{Arch_{150}} \leftarrow AddChildFeatures(FM_{Arch_{150}}, c, f_c, Arch_{150})$
end for
```

Algorithm 2. $AddChildFeatures(FM, c, f_p, Arch_{150})$

```
Require: A feature model ($FM$), a component ($c$), a parent feature ($f_p$), a 150% architecture ($Arch_{150}$).
Ensure: $FM$ enriched with the child features of $f_p$, if any.
for all $r \in OptionalReferences(c)$ do
    $MC \leftarrow FindMatchingComponents(Arch_{150}, r)$
    if $MC \neq \emptyset$ then
        $f_r \leftarrow CreateFeature(r)$
        $FM \leftarrow AddOptionalChildFeature(FM, f_p, f_r)$
        if $Multiplicy(r) = 0..1$ then
            $g \leftarrow CreateXORGroup()$
        else if $Multiplicy(r) = 0..N$ then
            $g \leftarrow CreateORGroup()$
        end if
        $FM \leftarrow AddGroup(FM, f_r, g)$
        for all $c_s \in MC$ do
            $f_{c_s} \leftarrow CreateFeature(c_s)$
            $FM \leftarrow AddChildFeatureOfGroup(FM, g, f_{c_s})$
            $FM \leftarrow AddChildFeatures(FM, c_s, f_{c_s}, Arch_{150})$
        end for
    end if
end for
```

2.2 Extracting FM_{Plug}

The extraction of the plugin feature model FM_{Plug} starts from the set of plugins $P = \{p_1, p_2, \ldots, p_n\}$ composing the system. This extraction is straightforward: each plugin p_i becomes a feature f_{p_i} of FM_{Plug}. If a plugin p_i is part of the system core, f_{p_i} is a mandatory feature, otherwise it is an optional feature. Each dependency of the form p_i *depends on* p_j is translated as an inter-feature dependency $f_{p_i} requires f_{p_j}$. Similarly, each p_i *excludes* p_j constraint is rewritten as an *excludes* dependency between f_{p_i} and f_{p_j}.

2.3 Mapping $FM_{Arch_{150}}$ and FM_{Plug}

When producing $Arch_{150}$, we keep track of the relationship between the input plugins and the architectural elements they define, and vice versa. On this basis, we specify a bidirectional mapping between the features of $FM_{Arch_{150}}$ and those of FM_{Plug} by means of *requires* constraints. This mapping allows us to determine (1) which plugin provides a given architectural feature, and (2) which architectural features are provided by a given plugin.

2.4 Deriving $\boldsymbol{FM_{Arch}}$

We now illustrate how we derive FM_{Arch} using $FM_{Arch_{150}}$, FM_{Plug}, the mapping between FM_{Plug} and $FM_{Arch_{150}}$, and an operation called *projection* using the example of Fig. 4.

First FM_{Plug} and $FM_{Arch_{150}}$ are *aggregated* under a synthetic root $FtAggregation$ so that root features of input FMs are mandatory child features of $FtAggregation$. The aggregation operation produces a new FM, called FM_{Full} (see Fig. 4). The propositional constraints relating features of FM_{Plug} to features of $FM_{Arch_{150}}$ are also added to FM_{Full}.

Definition 1 (Projection) *The projection is a unary operation on FM written as* $\Pi_{ft_1,ft_2,...,ft_n}\ (FM_i)$ *where* $ft_1, ft_2, ..., ft_n$ *is a set of features. The result of a projection applied to an FM,* FM_i*, is a new FM,* FM_{proj}*, such that:* $[\![FM_{proj}]\!] = \{\ x \in [\![FM_i]\!] \mid x \cap \{ft_1, ft_2, ..., ft_n\}\ \}$

Second, we compute the projection (see Definition 1) of FM_{Full} onto the set of features of $FM_{Arch_{150}}$ (i.e., $\mathcal{F}_{FM_{Arch_{150}}} = \{Arch,\ Ar1,\ \ldots,\ Ar6\}$). The projection produces a new FM, called FM_{Arch} (see Fig. 4). Formally:

$$\Pi_{\mathcal{F}_{FM_{Arch_{150}}}}\ (FM_{Full}) =\ FM_{Arch}$$

In the example of Fig. 4, the relationship between $[\![FM_{Full}]\!]$ and $[\![FM_{Arch}]\!]$ truly holds. We can notice that one configuration of the original $FM_{Arch_{150}}$ has been removed, i.e., $[\![FM_{Arch_{150}}]\!]\ \setminus\ [\![FM_{Arch}]\!] =\ \{Ar1, Ar2, Ar3, Ar6, Arch\}$. Indeed the projected FM_{Arch} contains an additional constraint $Ar3 \Rightarrow Ar5$, that was not present in $FM_{Arch_{150}}$. Similarly, the constraint $Ar4 \Rightarrow Ar6$ (grey tint in Fig. 4) can been derived but is redundant with $Ar3 \Rightarrow Ar5$. As we will see below, such constraint derivation can dramatically reduce the set of configurations of $FM_{Arch_{150}}$.

Implementation of the projection. Our previous experience in the composition of FMs has shown that *syntactical* strategies have severe limitations to accurately represent the set of configurations expected, especially in the presence of cross-tree constraints [1]. The same observation applies for the projection operation so that reasoning directly at the *semantic* level is required. The key ideas of our implementation are to *i)* compute the propositional formula representing the projected set of configurations and then *ii)* reuse the reasoning techniques proposed in [10] to construct an FM from the propositional formula. *Formula Computation.* For a projection $FM_{proj} =\ \Pi_{ft_1,ft_2,...,ft_n}\ (FM_i)$, the propositional formula corresponding to FM_{proj} is defined as follows :

$$\phi_{proj} \equiv \exists\ ftx_1, ftx_2, \ldots ftx_{m'}\ \phi_i$$

where $ftx_1, ftx_2, \ldots ftx_{m'} \in\ (\mathcal{F}_{FM_i} \setminus \{ft_1, ft_2, \ldots ft_m\}) =\ \mathcal{F}_{removed}$.

The propositional formula ϕ_{proj} is obtained from ϕ_i by *existentially quantifying* out variables in $\mathcal{F}_{removed}$. Intuitively, all occurrences of features that are not present in any configuration of FM_{proj} are removed by existential quantification in ϕ_i. The projection can be seen as a safe removal of a set of features

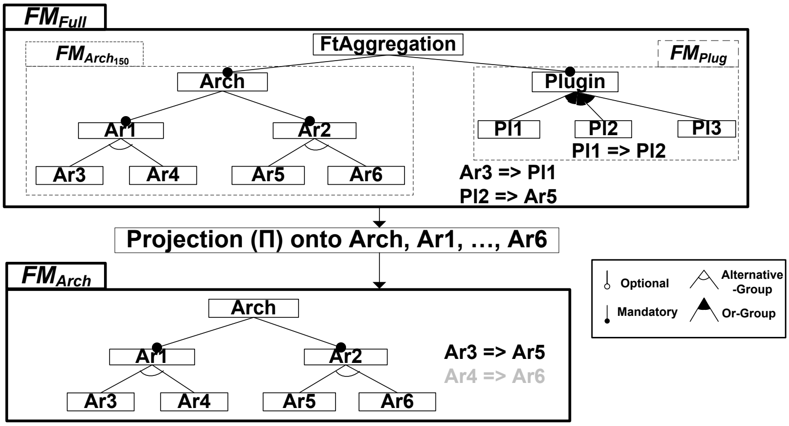


Fig. 4. Enforcing architectural FM using aggregation and projection

(i.e., existential quantification removes a variable from a propositional formula without affecting its satisfiability).

We rely on Binary Decision Diagrams (BDDs) to compute ϕ_{proj} [7]. A BDD can be seen as a compact representation of a propositional formula. BDDs have interesting properties, for example, computing the existential quantification of BDDs can be performed in at most polynomial time with respect to the sizes of the BDDs involved [7]. The techniques described in [10] (see below) also relies on a BDD-based implementation.

From Formula to FM. We use the algorithm presented in [10] to transform ϕ_{proj} into an FM. More precisely, the algorithm builds a tree with additional nodes for feature groups that can be translated into a basic *feature diagram*. Importantly, the algorithm indicates parent-child relationships (i.e., mandatory or optional features) and Alternative- or Or-groups. We use information of the original FM to favor features that were initially grouped.

The feature diagram, however, may be an over approximation of the original formula in the sense that if we translate the synthesized feature diagram to a propositional formula, noted $\phi_{proj_{diagram}}$, then some valid assignments of $\phi_{proj_{diagram}}$ may be invalid in ϕ_{proj}. We thus need to further constrain the feature diagram (as we did for the example of Fig. 4) in case ϕ_{proj} is not logically equal to $\phi_{proj_{diagram}}$. We propose to decompose the FM FM_{proj} as a feature diagram, a set of required constraints and a set of other propositional constraints (that cut across the hierarchy of the diagram). We compute the set of required constraints by first computing the implication graph, noted I_{proj}, of the formula ϕ_{proj} over $ft_1, ft_2, ..., ft_n$. I_{proj} is a directed graph $G = (V, E)$ formally defined as:

$$V = \{ft_1, ft_2, ..., ft_n\} \qquad E = \{(f_i, f_j) \mid \phi_{proj} \wedge f_i \Rightarrow f_j\}$$

Then, the set of require constraints $I_{requires}$ can be deduced by removing edges from I_{proj} being already expressed in the feature diagram (e.g., parent-child relations). The feature diagram *plus* the require constraints may still be an over approximation of ϕ_{proj} (it is not the case in Fig. 4). It is detected by checking the logical equality between ϕ_{proj} and $\phi_{proj_{diagram}} \wedge \phi_{proj_{requires}}$ ($\phi_{proj_{requires}}$ being the logical conjunction of all require constraints of $I_{requires}$). If needs be, an additional formula $\phi_{proj_{other}}$ is computed so that ϕ_{proj} is logically equal to $\phi_{proj_{diagram}} \wedge \phi_{proj_{requires}} \wedge \phi_{proj_{other}}$. We implement all the logical operations using BDDs.

3 Refining the Architectural Feature Model: Application

We conduct a study to *i)* determine if the architectural FM designed by the SA[1], noted FM_{SA}, is *consistent* with the extracted FM FM_{Arch} (and vice-versa) ; *ii)* step-wise refine FM_{SA} based on the previous observations. We describe the techniques developed for the case study and analyze the results[2].

3.1 Tool Support

We rely on FAMILIAR (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) [11,2], a domain-specific language dedicated to the management of FMs. FAMILIAR is an executable, textual language that supports manipulating and reasoning about FMs and is integrated in a comprehensive Eclipse-based environment (including graphical editors). We use FAMILIAR for two main purposes. Firstly, the extraction procedure generates FAMILIAR code to compute FM_{Arch}. Secondly, FAMILIAR provides the SA with a dedicated approach for easily manipulating FMs during the refinement process.

3.2 Results

Automatic Extraction. The $FM_{Arch_{150}}$ produced by the extraction procedure contains 50 features while the FM_{Plug} contains 41 features. The aggregated FM, FM_{Full}, resulting from $FM_{Arch_{150}}$, FM_{Plug} and the bidirectional mapping contains 92 features and 158 cross-tree constraints.

We first verify some properties of FM_{Full}. By construction, we know that the projection of FM_{Full} onto $\mathcal{F}_{FM_{Arch_{150}}}$ is either a refactoring or a specialization [23] of $FM_{Arch_{150}}$ (see Definition 2).

Definition 2 (Specialization, Refactoring, Generalization, Arbitrary Edit). *Let f and g be two feature models. f is a specialization of g if $[\![f]\!] \subset [\![g]\!]$ f is a generalization of g if $[\![g]\!] \subset [\![f]\!]$ f is a refactoring of g if $[\![g]\!] = [\![f]\!]$ f is an arbitrary edit of g if f is neither a specialization, a generalization nor a refactoring of g.*

[1] P. Merle, principal FraSCAti developer, plays the role of the SA in this study.

[2] See [24] for further details about the case study.

We observe that FM_{Arch} is a specialization of $FM_{Arch_{150}}$. More precisely, $FM_{Arch_{150}}$ admits 13958643712 possible configurations ($\approx 10^{11}$), while FM_{Arch} represents 936576 distinct products ($\approx 10^{6}$). As expected, the projection technique significantly reduces the over approximation of $FM_{Arch_{150}}$.

To improve the understanding of the difference between two FMs, we use a *diff* operator, denoted as $FM_1 \oplus_{\setminus} FM_2 = FM_r$. The following defines the semantics of this operator.

$$[\![FM_1]\!] \setminus [\![FM_2]\!] = \{x \in [\![FM_1]\!] \,|\, x \notin [\![FM_2]\!]\} = [\![FM_r]\!]$$

The formula ϕ_r of FM_r is used to reason about properties of FM_r (e.g., satisfiability) and is computed as follows:

$$\phi_r = (\phi_{FM_1} \wedge \ not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \ \wedge \ \neg(\phi_{FM_2} \wedge \ not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$$

For example, determining the kind of relationship between two FMs (see Definition 2) can be done by reusing the algorithm presented in [23] or by using the diff operator (see Definition 3).

Definition 3 (Diff and Specialization/Refactoring) *Let f and g be FMs. f is a specialization or a refactoring of g if* $(f \oplus_{\setminus} g)$ *has no valid configuration since* $[\![f]\!] \subseteq [\![g]\!]$ *is equivalent to* $[\![f]\!] \setminus [\![g]\!] = \emptyset$.

Moreover, the diff operator can compute the difference (if any) between two FMs in terms of set of configurations. In particular, we can compute the cardinality of this set. For example, we correctly check the following relationship[3] using the tool support: $|FM_{Arch_{150}}|$ - $\left|FM_{Arch_{150}} \oplus_{\setminus} FM_{Arch}\right| = |FM_{Arch}|$.

Refining Architectural FMs. The goal of the reverse engineering process is to elaborate an FM which *accurately* represents the valid combinations of features of the SPL architecture. The absence of a *ground truth* FM (i.e., an FM for which we are certain that each combination of features is supported by the SPL architecture) makes uncertain the accuracy of variability specification expressed in FM_{Arch} as well as in FM_{SA}. It is the role of the SA to determine if the variability choices in FM_{SA} (resp. FM_{Arch}) are coherent regarding FM_{Arch} (resp. FM_{SA}). In case variability choices are conflicting, the SA can *refine* the architectural FM.

We now report the problems encountered when reasoning about the relationship between FM_{Arch} and FM_{SA}. We also describe the advanced techniques we developed to assist the SA.

Reconciling FM_{Arch} ***and*** FM_{SA}***.*** A first obstacle concerned the need to *reconcile* FM_{Arch} and FM_{SA} (see Fig. 5). Both FMs come from difference sources and a preliminary work is needed before reasoning about their relationship. Firstly, the *vocabulary* (i.e., names of features) used differs in both FMs and should be aligned consequently. To resolve this issue, we rely on string matching techniques (i.e., Levenshtein distance) to automatically identify features of FM_{Arch}

[3] where $|FM_i|$ denotes the number of configurations of FM_i, i.e., $|FM_i| = |[\![FM_i]\!]|$.

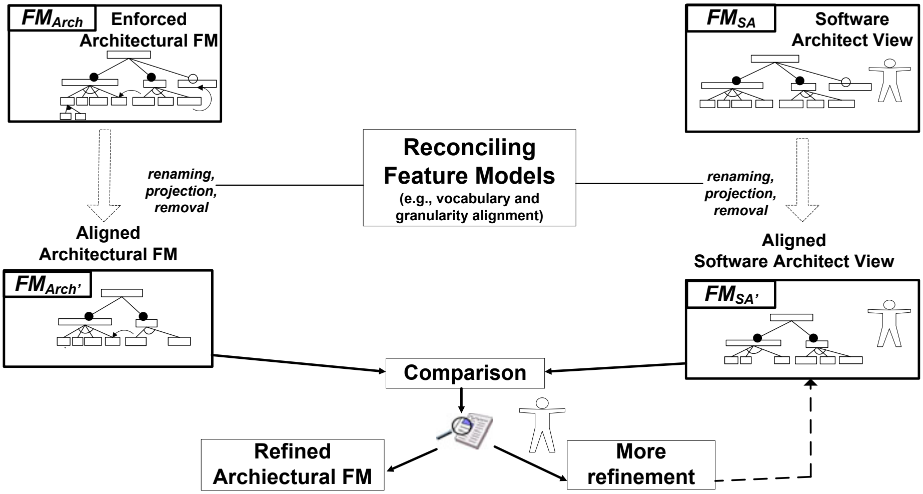

FMArch Enforced Architectural FM
FMSA Software Architect View
renaming, projection, removal
Reconciling Feature Models
(e.g., vocabulary and granularity alignment)
renaming, projection, removal
Aligned Architectural FM
FMArch'
Aligned Software Architect View
FMSA'
Comparison
Refined Archiectural FM
More refinement

Fig. 5. Process for Refining FM_{Arch}

that correspond to features of FM_{SA}. Then a renaming is applied on all corresponding features in FM_{Arch}. As an example, "MMFraSCAti" of FM_{SA} has been identified to correspond to "sca_metamodel_frascati" of FM_{Arch} and after the renaming FM_{Arch} contains the feature "MMFraSCAti". We automatically detect 32 features. The SA manually specifies the correspondence for 5 features in which the automated detection does not succeed (e.g., "MembraneFactory" corresponding to "fractal_bootstrap_class_providers"). Secondly, *granularity* details differ, (i.e., some features in one FM are not present in the other FM): FM_{SA} only contains 39 features whereas FM_{Arch} contains 50 features.

In FM_{SA} ***but not in*** FM_{Arch}**.** Two exclusive features $Felix$ and $Equinox$ are present in FM_{SA} but not in FM_{Arch}. We also observed that the two features are present in FM_{Plug} but not in $FM_{Arch_{150}}$ (and hence not in FM_{Arch}). A discussion with the SA reveals that these two plugins do not explicitly define architecture fragments in SCA. As a consequence, this variability point can simply not be identified in the architecture by the automatic extraction procedure.

In FM_{Arch} ***but not in*** FM_{SA}**.** We identified 13 features that are present in FM_{Arch} but not in FM_{SA}. Among others, two metamodels used by the SCA parser, three Bindings, two SCA properties, two Implementations and one Interface were missing. Given the complexity of the FraSCAti project, this is not surprising that the SA forgets some features. Hence, for most of the features, the SA considers the missing features as relevant and thus adds them in FM_{SA}. For one of the missing feature, "sca_interface_java", the SA reveals that he *intentionally* ignored it in FM_{SA}, arguing that it is a mandatory feature (i.e., every FraSCAti configuration has a Java interface) and that his focus was on variability rather than commonality. We indeed verify the mandatory nature of "sca_interface_java" in FM_{Arch}. Nevertheless, the SA decides to add "sca_interface_java" in FM_{SA}. Similarly, two first-level mandatory features, "binding_factory" and "services", were missing in FM_{SA}. The SA intentionally did not include the two features since they do not convey any further variation points, but he decides to edit FM_{SA} by adding those features. Another example concerns a feature of FM_{Arch}, "juliac", that adds unnecessary details (so that the way features are organized in FM_{SA} and FM_{Arch} slightly differ). Here the SA decides to remove "juliac" by projection.

***Reasoning about** FM_{Arch} **and** FM_{SA}.* At this step, we can *compare* FM_{Arch} and FM_{SA}. A first comparison is to determine the kind of relationship between FM_{Arch} and FM_{SA} (see Definition 2). We obtain an arbitrary edit, that is, some configurations of FM_{Arch} are not valid in FM_{SA} (and vice-versa). To go further, we use the diff operator (see Definition 3) and the merge in intersection mode (see [1]). We enumerate and count the unique configurations of FM_{Arch} and FM_{SA} as well as the common configurations of FM_{Arch} and FM_{SA}. Nevertheless, the techniques appear to be insufficient to really understand the difference between the two FMs. Intuitively, we need to identify more *local* differences. A first technique is to compare the variability associated to features of FM_{Arch} and FM_{SA} that have the same name. In particular, we detect that *i)* four features are optional in FM_{Arch} but mandatory in FM_{SA} and *ii)* three sets of features belong to Or-groups in FM_{Arch} whereas in FM_{SA}, the features are all optional. A second technique is to compute the intersection and the difference of the sets of require constraints in FM_{Arch} and FM_{SA} (based on their implication graphs, see page 228).

***Step-wise Refinement of** FM_{SA}.* The comparison techniques have been reiterated until having a satisfying architectural FM. Based on the comparison results, the SA had several attitudes. Firstly, he used FM_{Arch} to *verify* the coherence of his original variability specification in FM_{SA}. Secondly, he considered that some variability decisions in FM_{SA} are correct despite their differences with FM_{Arch}. The SA imposed five variability decisions not identified by the extraction procedure. Thirdly, he edits FM_{SA}, for example, by adding some constraints only present in FM_{Arch} or by setting optional a feature originally mandatory. The extracted FM notably identifies nine "obvious" constraints not expressed in FM_{SA} and allows the SA to incrementally *correct* FM_{Arch}.

3.3 Lessons Learned

The FraSCAti case study provides us with interesting insights into the reverse engineering of architectural FMs. First, the gap between FM_{SA} and FM_{Arch} appears to be manageable, due to an important similarity between the two FMs. However, it remains helpful to assist the SA with automated support, in particular, to establish correspondences between features of the two FMs. The most time-consuming task was to reconcile the granularity levels of both FMs. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature by projection, are not desirable but mandatory (i.e., basic manual edits of FMs are not sufficient).

Second, our extraction procedure (Section 2) yields very promising results. It recovers most of the variability expressed in FM_{SA} and encourages the SA to correct his initial model. A manual checking of the five variability decisions imposed by the SA shows that the extraction is not faulty. It correctly reproduces the information as described in the software artefacts of the project.

Third, the SA *knowledge* is required *i)* to scope the SPL architecture (e.g., by restricting the set of configurations of the extracted FM), especially when software artefacts do not correctly document the variability of the system and *ii)* to control the accuracy of the automated procedure. An open issue is then to provide a mechanism and a systematic process to reuse the SA knowledge, for example, for another version of the architectural FM of FraSCAti.

4 Related Work

Despite the importance of variability management in software engineering in general, and in software architectures in particular [5], the problem of reverse engineering the variability of *existing* systems has definitely not received sufficient attention from the research community. While our work takes an architectural perspective, the other existing approaches in the field consider different input artifacts including legacy system documentation [13] or textual requirements [3]. In their recent work, She *et al.* [20] propose a reverse engineering approach combining two distinct sources of information: textual feature descriptions and feature dependencies. Our approach also benefits from the combination of two (other) sources of information, namely plugin dependencies and architecture fragments. They mostly focus on the retrieval of the feature diagram (heuristics for identifying the most likely parent feature candidates of each feature, group detection, etc.) and assume that the set of valid configurations is correctly restituted, which is clearly not the case in our work. We also support the identification of feature groups (based on architectural extension points), of the right parent feature of each feature (based on architectural hierarchy) and of inter-feature dependencies (through projection of plugin dependencies).

The FM analysis and reasoning techniques used in this paper reuse and extend previous work in SPL and requirement engineering [6]. Metzger *et al.* [16] propose an approach to cross-checking *product-line* variability and *software* variability models, thus assuming that such models (or views) are available. Our approach is complementary since it allows the recovering of the *actually supported* variability of a software system, and since it involves the cross-analysis of architectural and plugin FM. One of the key component and original contribution of our work is the projection operator we have defined and realized (see Section 2). Lopez and Eyged [14] address a related problem in the context of safe composition by checking the consistency of multi-view variability models. In particular, they check whether an FM developed by a domain expert is a specialization or a refactoring of an FM representing the variability of multiple models. Thüm *et al.* [23] reason on the nature of FM edits, and provide a classification that we rely on when comparing the extracted FM with the software architect view. As we have shown, reasoning about the relationship of two FMs is inappropriate until FMs are not reconciled, i.e., pre-directives (e.g., safe removal of unnecessary details) have to be applied before. Benavides et al. compared the performance of CSP, SAT solvers and BDD solvers for some reasoning operations on FMs [6]. As future work, we will investigate the use of SAT or CSP solvers to realize the diff/projection operators. A comparison with BDD-based implementations is planned to determine the most scalable solution. Another research direction is to consider feature *attributes* [6], for example, to model quality attributes of the FraSCAti architecture.

5 Conclusion

Variability management is of crucial importance in the management of large families of software systems. While feature models have long been recognized as expressive means to compactly represent software variability from different perspectives, building one of them for a large system is a complex, time-consuming and error-prone activity. In this paper, we presented a tool-supported approach to reverse engineer software variability from an architectural perspective. The reverse engineering process involves the automatically supported extraction, aggregation, alignment and projection of architectural feature models. It has the merit of combining several sources of information, namely software architecture, plugin dependencies and software architect knowledge. We successfully evaluated the proposed approach when applied to FraSCAti, a large and highly configurable plugin-based system. We showed that our automated procedures allow for producing both correct and useful results, thereby significantly reducing manual effort. We learned, however, that fully automating the process is not realistic nor desirable, since the intervention of the software architect remains highly beneficial. The ongoing evolution of the FraSCAti project will bring us an opportunity to study how to reuse the accumulated knowledge of the software architect. As the validation was only conducted on a single case study, we need, on the long term, to adapt the proposed process to show its applicability to other forms of architecture (e.g., OSGi) and other architectural concepts. This should make significant steps to the provision of a validated, systematic process for extracting architectural variability models.

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Comparing Approaches to Implement Feature Model Composition. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 3–19. Springer, Heidelberg (2010)
2. Acher, M., Collet, P., Lahire, P., France, R.: A Domain-Specific Language for Managing Feature Models. In: SAC 2011, pp. 1333–1340. ACM, PL Track (2011)
3. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: SPLC 2008, pp. 67–76. IEEE, Los Alamitos (2008)

4. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology (JOT) 8(5), 49–84 (2009)
5. Bachmann, F., Bass, L.: Managing variability in software architectures. SIGSOFT Softw. Eng. Notes 26, 126–132 (2001)
6. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated Analysis of Feature Models 20 years Later: a Literature Review. Information Systems (2010)
7. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a bdd package. In: DAC 1990, pp. 40–45. ACM, New York (1990)
8. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, Reading (2001)
9. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE 2006, pp. 211–220. ACM, New York (2006)
10. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC 2007, pp. 23–34 (2007)
11. FAMILIAR, http://nyx.unice.fr/projects/familiar/
12. FraSCAti, http://frascati.ow2.org
13. John, I.: Capturing product line information from legacy user documentation. In: Software Product Lines, pp. 127–159. Springer, Heidelberg (2006)
14. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 217–232. Springer, Heidelberg (2010)
15. Lopez-Herrejon, R.E., Egyed, A.: On the need of safe software product line architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 493–496. Springer, Heidelberg (2010)
16. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.-Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: RE 2007, pp. 243–253 (2007)
17. Parra, C.A., Cleve, A., Blanc, X., Duchien, L.: Feature-based composition of software architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 230–245. Springer, Heidelberg (2010)
18. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. 51(2), 456–479 (2007)
19. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.-B.: Reconfigurable SCA Applications with the FraSCAti Platform. In: SCC 2009, pp. 268–275. IEEE, Los Alamitos (2009)
20. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: ICSE 2011, pp. 461–470. ACM, New York (2011)
21. SCA standard, http://www.osoa.org/
22. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques: Research articles. Softw. Pract. Exper. 35(8), 705–754 (2005)
23. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE 2009, pp. 254–264. IEEE, Los Alamitos (2009)
24. Companion webpage, https://nyx.unice.fr/projects/familiar/wiki/ArchFm

Supporting Communication and Cooperation in Global Software Development with Agile Service Networks

Damian Andrew Tamburri and Patricia Lago

VU University Amsterdam
The Netherlands
{d.a.tamburri,p.lago}@vu.nl

Abstract. Current IT markets exhibit many constraints (e.g. budget, staff shortage, etc.). These constraints force IT companies to increase productivity using globally distributed manpower. Literature shows that global software development (GSD) indeed raises productivity but reduces communication and collaboration between teams. Consequently, the risk of failure increases. To ease communication and collaboration among teams, novel engineering methods must be provided. To address this problem, we propose using Agile Service Networks (ASNs). ASNs are an emergent paradigm in which service oriented applications (network nodes) collaborate through agile and dynamic service interactions (network edges). Agile interaction among ASN nodes, allow mitigating distance (typical of GSD) by dynamically adapting communication and collaboration as needed. Through ASNs, GSD can be seen as a global network of resources (teams, documentation, knowledge, etc.) among which agile interactions allow flexible knowledge exchange and team collaboration. To establish feasibility of our proposal, we investigated how ASNs can support GSD. Based on existing works in the fields of both ASNs and GSD, we mapped GSD challenges on ASNs key features and devised a meta-model showing how ASNs are used to support GSD requirements.

1 Introduction

Our global economy is constantly challenged by time-to-market and budget issues. Moreover, the availability and cost of manpower rapidly change. To maximize productivity in these conditions, IT companies carry out software development globally. Ideally, by using teams in different sites and timezones, all 24 hours in a working day can be rendered productive. Unfortunately, when doing so, the issues in knowledge exchange and synchronization among teams are often underestimated. These problems regard people rather than technology, and hence they are very difficult to study. In addition, costs inevitably raise because of increased travel needs (e.g. for management and architects' meetings etc.). Consequently, workforce becomes ineffective, costs prohibitive and ultimately, projects fail [6,13]. The problem we want to address is the lack of practices and tools to support these issues in GSD.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 236–243, 2011.

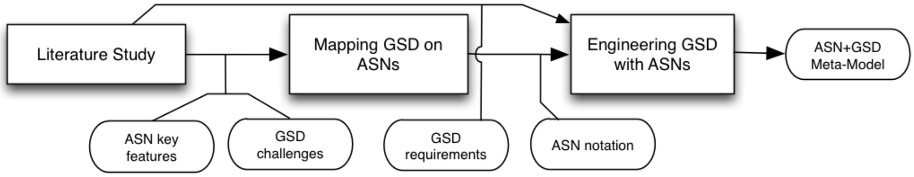


Fig. 1. Research Approach

ASNs are networks of service-oriented applications (network nodes) created by collaborative service interactions (network edges) among many cooperating industrial parties. Through ASNs, complex yet agile and adaptable business transactions take place on a global scale.

Similarly to GSD processes, ASNs stem from collaborative business processes [4,2], distributed on a global scale. Since GSD is indeed a business process (complying with the definition in [10]) ASNs can be used to model the business process of developing software globally. Their networked and agile nature can be enriched to support both social and technical requirements of GSD. In this paper we investigate how can ASNs support GSD processes. ASNs were only recently introduced, and using them to support GSD was never researched so far. Therefore our investigation faces challenges such as limited literature on ASNs and no related work. Another interesting challenge regards the social aspects of GSD: these must be represented and supported through ASNs, which are defined as a technical system. Two main contributions are offered: *(i)* a mapping of GSD challenges on ASN key features, showing that ASNs can support GSD; *(ii)* a meta-model that shows ASNs supporting GSD requirements. Figure 1 shows our research steps (rectangles) as well as inputs and outputs (rounded rectangles). First we carried out a literature study obtaining GSD challenges, GSD requirements and ASNs' key features. Then we showed feasibility of our proposal by mapping GSD challenges on ASNs' key features. Finally, we devised a meta-model to show how GSD can be supported by ASNs. This meta-model was obtained extending an existing ASN notation [15,1] to support GSD requirements.

2 Literature Study

This section surveys Agile Service Networks and Global Software Development. To gather clear-cut literature for ASNs, we applied the topic search string (i.e. "Agile Service Networks") to major scholarly search engines (Google Scholar, IEEExplore, ACM Digital Library, Wiley Interscience, Microsoft Academic Research). For GSD, we consulted experts in the field. The resulting publications were [4,12,14,2] for ASNs and [7,3,6,8,13,5] for GSD. To these publications we added [15], a publication from the S-Cube consortium (available at www.s-cube-network.eu) discussing Service Networks.

2.1 Agile Service Networks

Analyzing the selected papers we have identified the following key features exposed by ASNs.

ASNs are dynamic: All the papers describe ASNs as being highly dynamic entities. In [4,14,15] dynamism is seen as essential part of service interactions in collaborative industrial networks (i.e. industrial value networks [11]). Dynamic agility in this context is regarded as the immediate ability to adapt to dynamic changes in demand and offer.

ASNs are business-oriented: All papers promote the concept of ASNs from a business perspective. ASNs emerge from business corporative collaborations [4] and represent complex service applications interacting in a networked business scenario involving multiple corporations or partners in different sites (i.e. different geolocations) [2]. Within ASNs, business value can be computed, analyzed and maximized [4,12].

ASNs are collaborative: In all papers, ASNs are defined as interoperating business alliances. Each member cooperates with others to achieve a common goal (e.g. service level, value increase). Therefore, ASNs are collaborative.

ASNs are emergent: There are no engineering and design methods specific to ASNs. They form spontaneously as a consequence of business alliances teaming-up to collaboratively increase business value through corporative partnership [4,14,15,2].

In addition to these key features, we used the ASN notation in Figure 2, taken from [15]. The main architectural elements for ASNs in the notation are **Participant**s (ASN nodes) and **Relation**s (ASN edges). For the sake of space, we do not further discuss this notation and urge the reader to refer to [15] and [1] for further details.

2.2 Global Software Development

Analyzing the suggested papers we have identified the following challenges in GSD.

Social Aspects are important to enable teams to integrate and exchange knowledge correctly [5,7]. In [5] GSDs are comprised of globally distributed teams

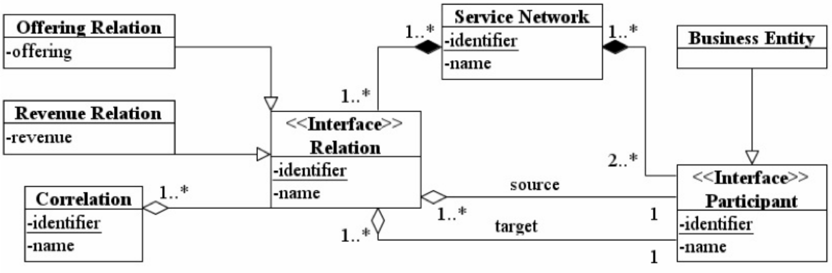


Fig. 2. Service Networks Notation from [15] and [1]

carrying out an objective collaboratively. Collaboration is increased by socialization in teams and social networking [7].

Collaboration increases productivity by raising team interaction, awareness and responsibility on the project [7]. In [6] the key issue for GSD is coordination in dynamic contexts. Collaborative effort, is required for GSD to succeed.

Flexibility in management, to coordinate multisite development [8,6]. Ideally GSD should be able to use all available resources regardless of geographical location and coordinate these collaboratively. Management should be flexible enough to provide fine grained control over all types of resources (e.g. documentation, people down to individual skills). Knowledge localization is challenging since granularity of management and control over resources and people is limited [8].

Reduced dependency among teams, so that productivity of one team is not impacted by productivity of others. Distance can be compensated with tactics to increase communication, loosen teams dependency and limit participants' cultural difference [3].

Coordination of all resources available, i.e. manpower, tools, document artifacts, knowledge, to timely allocate resources and maximize productivity [3,6]. GSD often fails because many of the mechanisms that coordinate work in co-located projects (e.g. stand up or colloquial meetings, informal "water-cooler" talk etc.) are absent or disrupted.

Geolocalization to allow project awareness among teams. Since teams are geographically dispersed and often unknown to each other, they need intercommunication and awareness infrastructures to actively participate on the project [13,9,7].

Finally, from these papers we elicited requirements for GSD processes (for the sake of space the list is not present here and is availableonline[1]). We obtained these requirements by: *(a)* scanning through the literature, coding text describing requirements or needs for GSD processes; *(b)* analyzing industrial case studies from [13]. In total, we obtained 17 Requirements from literature coding, and 12 requirements from the real-life industrial GSD scenarios in [13].

The entities and relationships occurring in GSD processes (according to requirements) can be summarized in the following scenario:

*"Company X develops software globally by using N globally distributed **teams**. Each team is made of **engineers** with individual **skills, social background, roles**, etc. A **global team map** is used to track **location, timezone** and **knowledge** of every team (e.g. skills, documentation available, progress made on artifacts, etc). One or more teams are **core** teams since their task is managing the whole process, checking **shared documentation**, deciding a project-wide **technical space** and planning **travel budget**. Travel budget is needed for the frequent "**awareness**" meetings among teams. **Shared documentation** is needed to document the project and also to increase awareness of every member. A common **technical space** is needed to ease communicability (e.g. common formats) and knowledge exchange (e.g. common platforms). As soon as **requirements** are agreed with the **stakeholders** they are used to generate a **global***

[1] http://www.picfront.org/d/87GP

architecture. *Once the global architecture is defined, it is split into* ***project units***. *Project units are allocated to* ***engineering*** *teams, responsible for their development.* ***Service*** *teams update shared documentation to allow consistency and further increase* ***project awareness***”.

Words in bold in the scenario represent the entities taking part in GSD processes. This scenario and the GSD requirements it represents, are used in the definition of our meta-model in section 4.

3 Mapping GSD to ASNs

This section shows that ASNs can be used to support GSD processes. To this aim, GSD challenges were matched with ASNs key features (both presented in Section 2). Table 1 summarizes results. Column 1 represents GSD challenges, while column 2 shows ASNs' matching key feature. Column 3 provides rationale. The table shows that all GSD challenges found can be supported by ASNs key features.

Table 1. Mapping of ASN characteristics on Global Software Development

GSD needs...	***... ASNs are...***	***Rationale***
social aspects	***business-oriented***	ASNs stem from the business strategies for collaborative value increase. These strategies are modeled around social demands and user profiles (social context, background, social extraction, etc.). This means that ASN nodes are modeled to satisfy customers' (social) characteristics[4,15].
collaboration and awareness	***collaborative***	Agile Service Networks are generated through collaboration of networked service applications[4]. Formally, collaboration terms are stated in service level agreements [2,14]. This means that every ASN node must collaborate with other to achieve the network's goal (similarly to GSD Collaboration needs). In so doing, formal service level agreements must be in place so that collaborating nodes know what are the terms of the collaboration (similarly to GSD awareness needs).
collaborative coordination		ASNs are collaborative and adaptable to context change. Service applications coordinate spontaneously to achieve results in accordance to fixed service level needs[2,14]. Dynamic adaptation of both nodes and interactions allows dynamic coordination.
reduced dependency among teams		ASNs provide clear-cut definitions of network nodes (i.e. service applications)[14]. Agile interactions between nodes enable loose dependency: if one (service) node is not available, another node can be called up [2].
management flexibility	***dynamic***	Agile Service Networks provide a dynamic infrastructure, adaptable to context change. Agile interactions among nodes allow for flexible management of the network.[2].
geo-localization of resources	***emergent***	agile service networks are emergent through service discovery, localization and management of serving nodes [4,15,14]

4 Engineering GSD with ASNs

To show how ASNs can support GSD, requirements for GSD processes must be satisfied through ASNs. In this section we show a meta-model in which entities

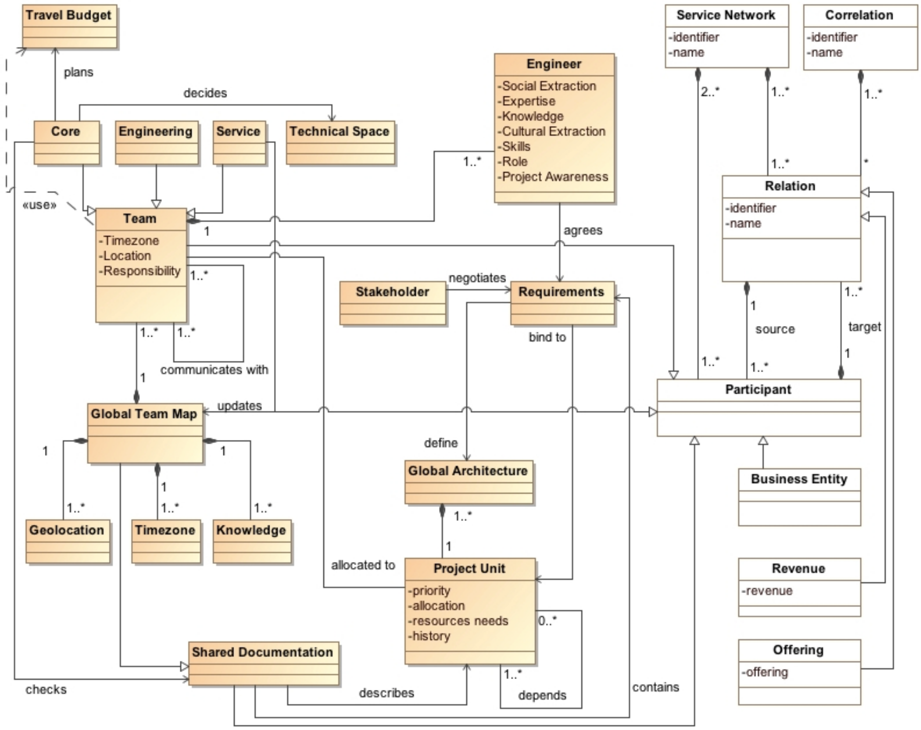


Fig. 3. ASN notation for GSD

and relations from GSD requirements (as summarized in the scenario closing section 2.2) are modeled through an ASNs notation (see Figure 2). To build this meta-model, we first reproduced entities and relationships stemming from GSD requirements. Then we reproduced the ASN notation in the meta-model. Finally, we extended the ASN notation by specialization (i.e. by drawing a generalization from GSD specific concepts to ASN generic concepts). More formally, the following "merging" rule was applied:

"*specialize the* **Participant** *class from Figure 2 with all entities that take active part in GSD according to requirements (i.e. that are participants in an ASN). Specialize the* **Relation** *class with all relations among resulting* **Participant***s*".

This rule is both necessary, and sufficient. It is necessary since all the active contributors in GSD must be **Participant**s in an ASN; it is sufficient, since all remaining elements to be merged (i.e. relations between **Participant**s) are **Relation**s in the ASN.

Therefore, the concept model in Figure 3 was obtained by drawing the entities and relationships required for GSD (i.e. stemming from the requirements we elicited) and then applying the rule defined above.

On the left hand side, The model shows the entities and relationships stemming from the requirements (filled), while the ASN notation (originally in Figure 2) is on the right hand side (non filled). The two are merged by specializing the

Participant class on the right, with **Team**s, **Global Team Map** and **Shared Documentation** classes on the left. Since these three entities carry out (either directly or indirectly) the software development, indeed they are the active participants in GSD, according to requirements. Relations taking place among these elements are ASN transactions (i.e. **Relation**s). For the sake of clarity in Figure 3 we do not show the relations on the GSD side (left, filled) specializing the **Relation** class on the ASN side (right, non-filled).

Indeed this meta-model shows that an ASN to support GSD processes can be created by modeling **Team***s,* **Global Team Map***s and* **Shared Documentation** *as active* **Participant***s within the ASN. Consequently, the relations between these are ASN collaborative transactions (i.e.* **Relation***s).*

5 Conclusions and Future Work

In this paper we wanted to establish if and how ASNs supported GSD. To this aim we systematically searched for literature in ASNs and GSD. From the gathered literature we obtained ASN key features, GSD challenges and GSD requirements. Mapping GSD challenges on ASNs' key features led us to conclude that ASNs indeed support GSD. Moreover, extending an ASN notation to meet GSD requirements, we have shown how this support can be concretized.
This notwithstanding, it can be noticed that ASNs are still missing some important architectural elements, e.g. social aspects of GSD. These aspects are key to provide added-value support tools. Since GSD actors are teams part of organizational structures (i.e. corporations, software companies etc.), a systematic literature review into Organizational Social Structures is being carried out. From this study we hope to develop a socio-organizational context model to enrich ASNs. Moreover, since Figure 3 is a meta-model, i.e. a model for a model, further exploration of model-driven engineering methods for GSD through our ASN_{GSD} meta-model is in order. Moreover, validation of this meta-model should be put in place to make its support to GSD meaningful. For this, industrial case studies should be developed and results should be analyzed against industrial expectations (e.g. a focus group). Further on, more experimentation should be invested in simplifying / improving the model in Figure 3 (e.g. action research)[2].

References

1. Bitsaki, M., Danylevych, O., Heuvel, W.-J., Koutras, G., Leymann, F., Mancioppi, M., Nikolaou, C., Papazoglou, M.: An architecture for managing the lifecycle of business goals for partners in a service network. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 196–207. Springer, Heidelberg (2008)

[2] The authors acknowledge the support of the European Community's Seventh Programme FP7/2007- 2013, grant agreement 215483 (S-Cube), for partially funding this project.

2. Bucchiarone, A., Cappiello, C., Di Nitto, E., Kazhamiakin, R., Mazza, V., Pistore, M.: Design for adaptation of service-based applications: main issues and requirements. In: Proceedings of the 2009 International Conference on Service-Oriented Computing, ICSOC/ServiceWave 2009, pp. 467–476. Springer, Heidelberg (2009)
3. Carmel, E., Agarwal, R.: Tactical approaches for alleviating distance in global software development. IEEE Software 2(18), 22–29 (2001)
4. Carroll, N., Whelan, E., Richardson, I.: Applying social network analysis to discover service innovation within agile service networks. Service Science 2, 225–244 (2010)
5. Ebert, C., De Neve, P.: Surviving global software development. IEEE Software 18(2), 62–69 (2001)
6. Herbsleb, J.D.: Global software engineering: The future of socio-technical coordination. In: Briand, L.C., Wolf, A.L. (eds.) FOSE, pp. 188–198 (2007)
7. Herbsleb, J.D., Mockus, A., Finholt, T.A., Grinter, R.E.: An empirical study of global software development: distance and speed. In: ICSE 2001: Proceedings of the 23rd International Conference on Software Engineering, pp. 81–90. IEEE Computer Society, Washington, DC, USA (2001)
8. Herbsleb, J.D., Moitra, D.: Guest editors' introduction: Global software development. IEEE Software 18, 16–20 (2001)
9. Herbsleb, J.D., Moitra, D.: Global software development. IEEE Software 18(2), 16–20 (2001)
10. http://www.opengroup.org/projects/soabook/ Soa source book
11. Jetter, M., Satzger, G., Neus, A.: Technological innovation and its impact on business model, organization and corporate culture - ibm's transformation into a globally integrated, service-oriented enterprise. Business & Information Systems Engineering 1(1), 37–45 (2009)
12. Metzger, A., Pohl, K.: Towards the next generation of service-based systems: The S-cube research framework. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 11–16. Springer, Heidelberg (2009)
13. Sangwan, R., Bass, M., Mullick, N., Paulish, D.J., Kazmeier, J.: Global Software Development Handbook. Auerbach Series on Applied Software Engineering Series. Auerbach Publications, Boston (2006)
14. van den Heuvel, W.-J., Zimmermann, O., Leymann, F., Lago, P., Schieferdecker, I., Zdun, U., Avgeriou, P.: Software service engineering: Tenets and challenges. In: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, PESOS 2009, pp. 26–33. IEEE Computer Society, Washington, DC, USA (2009)
15. Andrikopoulos, V., Benbernou, S., Bitsaki, M., Danylevych, O., Hacid, M.S., van den Heuvel, W.J., Karastoyanova, D., Kratz, B., Leymann, F., Mancioppi, M., Mokhtari, K., Nikolaou, C.N., Papazoglou, M.P., Wetzstein, B.: Survey on business process management. Deliverable 1. S-Cube Consortium (July 14, 2008)

Reducing Architectural Knowledge Vaporization by Applying the Repertory Grid Technique

Dan Tofan, Matthias Galster, and Paris Avgeriou

Department of Mathematics and Computing Science, University of Groningen
The Netherlands
{d.c.tofan,m.r.galster}@rug.nl, paris@cs.rug.nl

Abstract. The architecture of a software-intensive system is the composition of architectural design decisions. These decisions are an important part of Architectural Knowledge (AK). Failure to document architectural design decisions can lead to AK vaporization and higher maintenance costs. To reduce AK vaporization, we propose to apply the Repertory Grid Technique (RGT) to make tacit knowledge about architecture decisions explicit. An architect can use the RGT to elicit decision alternatives and concerns, and to rank each alternative against concerns. To validate our approach, we conducted a survey with graduate students. In the survey, participants documented decisions using the RGT. We compared these decisions with decisions documented using a basic decision template. Our results suggest that RGT leads to less AK vaporization, compared to conventional ways of documenting decisions.

Keywords: architectural knowledge, repertory grid, AK vaporization, survey.

1 Introduction

Bosch [2] considers the software architecture of a system as the composition of a set of architectural decisions. Kruchten et al. [9] define Architectural Knowledge (AK) about a system as design decisions and design. De Boer and Farenhorst [1] investigated various definitions of AK in literature. They obtained evidence for the importance of decisions in AK and argued that decisions represent a link from the problem domain to the solution domain.

Bosch [2] considers AK vaporization as a key problem in software architecture. AK vaporization contributes to expensive system evolution, lack of stakeholders' communication and reduced reusability [7]. Given the importance of AK vaporization, the architecting community has proposed various approaches for reducing it. For example, Bosch [2] argued for the representation of design decisions as first class entities in software architecture. Kruchten et al. [9] discussed an AK repository, and its underlying ontology. Jansen [7] studied decisions recovery and their dependencies. Kruchten et al. [8] described a decision view that incorporates design decisions in the '4+1' view model. Furthermore, the ISO/IEC 42010 standard provides recommendations for incorporating design decisions in architecture descriptions.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 244–251, 2011.

In this paper we propose to use the Repertory Grid Technique (RGT) for capturing architectural decisions. In general, the RGT follows four steps: choose a decision topic, obtain alternatives, get concerns, and rate alternatives against each concern. The resulting grids can be used for hierarchical cluster analysis of alternatives and concerns. For details on the RGT please refer to [4]. In a previous study [10], we already identified some advantages and disadvantages of RGT. In this paper, we investigate how RGT may reduce AK vaporization.

2 Applying the RG Technique

We conducted a descriptive survey with the goal of investigating the reduction of AK vaporization by applying RGT. As sample, we invited graduate students enrolled in a Software Architecture course at the University of Groningen in 2010. Participation was voluntary and had no influence on grades. The study was scheduled to take place as part of an optional 2-hour seminar.

In the session, we first trained the students to use the RGT and then asked them to apply it to some architectural decisions. Participants used the RGT to describe decisions that occurred in the context of the course project. The course project required students to act as architects, and design a complex home automation system that interacts with the Smart Grid in order to sell and buy electric power. Throughout the semester, students worked in groups of five persons to architect the system. As our study took place halfway through the project, students had already made the important architectural decisions and thus possessed AK about the home automation system. Therefore, we captured the AK of students about some of the decision topics that occurred in their course project. Even though students worked as teams, we decided to apply the RGT at an individual level, for simplicity reasons.

To structure our study, we used Basili's Goal-Question-Metric (GQM) method. Our *goal* is to reduce AK vaporization from architects' viewpoint. This is important to both practitioners and researchers, because knowledge vaporization leads to increased maintenance costs. Our *question* is: Does the RG technique reduce AK vaporization, compared to a basic approach to document architectural decisions?

Selecting *metrics* for AK vaporization was more difficult. In our context, the decisions documented in architectural documents in the course project were based on a predefined template. Given their lack of experience, students were asked to describe the decision topic, alternatives, their pros and cons, decision outcome, and rationale. Moreover, they did not use any established approach for documenting decisions. We speculate that architectural documentations created by students are similar to documentations produced by practitioners who do not use systematic approaches for decision documentation. From our experience, we consider that decision documentation in industry is rarely systematic, even compared to the templates filled by students. On the other hand, the RG output is the result of a systematic, but unproven approach.

Existing work on AK vaporization has yet to offer techniques for measuring vaporization and to facilitate the evaluation of new approaches that claim to reduce AK vaporization. Thus, we defined the following three *metrics* to compare how RG-based decisions differ from decisions based on a basic decision template. First, we considered the number of explicit decision alternatives, followed by the number of concerns. The third metric was the ratio of the number of expressed rankings compared to the maximum of possible rankings. These metrics could be obtained from the RGT output as well as from the basic architectural documentations created by students. A higher number of explicit alternatives, concerns or rankings may suggest a reduction in AK vaporization. To increase reliability, two researchers conducted this analysis. For the ratio of rankings, we divided the number of explicit rankings by the maximum number of possible rankings. For example, if a decision has 4 explicit rankings, 2 alternatives and 3 concerns, then the ratio is 0.66.

Next, we describe the preparations for our data collection efforts. Due to time constraints for the study, we needed a RG tool that participants could use for the AK acquisition task. After piloting a few tools, we selected Idiogrid [6], because it supports self-administering the RGT. Next, we describe the four steps of RGT as performed in the study.

Choose Decision Topic. We prepared some decision topics for students to use in the survey. To apply RGT, participants needed to be experts on the decision topic, or be at least knowledgeable about the topic. To satisfy this condition, we analyzed the course project reports, delivered by students. For each of the six groups, we compiled a list of decision topics and alternatives. Next, we analyzed which decision topics appeared across all groups, to see which topics are more common. We identified four such topics: choice of user interface, programming language, communication technology, and operating system. To satisfy the expertise prerequisite, we asked each student to choose two out of the four decision topics, based on his/her familiarity.

Get Alternatives. According to Edwards et al. [3], the alternatives (or elements) used in the RGT can either be supplied to the participant or elicited from him/her. The former is suitable for investigations on a specific set of elements [3]. As we aimed to elicit AK from participants, we decided to ask participants to specify the alternatives for each selected topic.

Get Concerns. Similar to the previous step, we decided to elicit characteristics (or constructs) from participants, rather than to supply them. However, in our previous study [10], we used the triadic elicitation approach in individual interviews: we asked the expert in what way two of three alternatives (elements) are alike, but different from the third. For this study, we doubted that most students could successfully use the triadic approach in a self-administered session, through a dialog with a tool, instead of an interviewer. According to Grice et al. [5], grids based on sentence completion are suitable for any domain of experience, and are easy to complete. Following our pilot sessions, we concluded that it may be more intuitive for students to generate constructs through sentence completions, compared to the triadic elicitations. Therefore, we decided to use the sentence completion approach.

Rate Alternatives. For this step, we configured Idiogrid to use a 5-point rating scale, ranging from -2 to 2. When rating an alternative against a concern, lower values indicated agreement with the concern (i.e., affordable), while higher values indicated agreement with its opposite (i.e., expensive). The middle value indicated neutrality, uncertainty or lack of applicability.

When executing the survey, we first introduced the participants to the RG technique. Afterwards, we asked the participants to do an example grid session, for training purposes, with the topic of choosing between bars in town. Next, participants applied RGT, on decision topics of their choice. We asked the participants not to use the internet, or talk with each other during the study. We did not impose a time limit for the RG sessions. Due to the limited duration of the session, we skipped grids analysis and refinement. Instead, we decided to send each student an email with an analysis of his/her grids. Furthermore, two researchers were available to answer questions from participants during the study.

At the end, each participant filled in a questionnaire. We were interested in the profile of the participants. Moreover, we added questions on the study itself, e.g., to check whether the participants understood the instructions and questions, and the perceived difficulty of performing the RGT for the decision topics.

4 Analysis of Survey Results

Each of the 20 participants delivered two grids, except for two persons who had to leave earlier and produced only one grid. One participant delivered no grid at all. Overall, we obtained 36 grids, as well as paper-based post questionnaires from attendants. Additionally, students delivered 6 architectural documents, at the end of the course. As described in the survey design, we collected measurements for both, the grids and architectural documents.

To measure the number of explicit concerns, two authors of this paper conducted a content analysis on each decision description. Each researcher individually assigned a concern to every sentence of a decision's description. Next, we reviewed every sentence, and compared the assigned concerns. We considered an agreement if both researchers meant the same thing, but used different words. For example, if one assigned to a fragment the concern *cost* and the other one assigned *affordability,* then it is an agreement. Upon disagreements, we either agreed to use an existing concern from one of us, or we negotiated the assignment of a new concern. For a few sentences, we asked a third researcher to mediate. Initial inter-rater agreement was 51.8%. After negotiations, we achieved full agreement.

For the third metric, we needed to determine the ratio of explicit rankings, against the maximum number of possible rankings. We multiplied the number of explicit alternatives and the number of distinct concerns to obtain the maximum number of possible rankings.

4.2 Analyzing Metrics for All Decisions

Some students from the same course project group used the RGT on identical decision topics. We obtained 12 grids for which only one student from a course project group addressed that decision topic (single grids). Additionally, we obtained 7 double grids: two participants from the same group produced individually two grids on the same decision. Also, we got 2 triple grids, from three members of the same group capturing the same decision. Similarly, we obtained 1 quadruple grid, from four members of the same group.

To ensure a suitable comparison of the metrics obtained from the grids and the project reports, the grid must have been produced by a student who also co-authored the architectural document. Our raw data consisted of 12 data points of single grids, 7 data points of double grids, 2 data points of triple grids, and 1 data point of quadruple grids.

To analyze the data, we needed to filter outliers. We noticed that one data point of double grids and the one with quadruple grids had poor decisions descriptions in the architectural document, i.e., no alternatives considered. Therefore, we eliminated the two data points. In one of the triple grids, we eliminated a grid due to poorly phrased concerns and obtained a new data point with double grids. Similarly, we converted the other data point with triple grids into a new one with a single grid, by removing two poor quality grids. After filtering, we obtained 13 single-grid data points, and 7 double-grid data points. The numbers of alternatives and concerns in a double-grid data point were calculated by counting the distinct alternatives and concerns from the two grids.

The boxplots in figure 1 summarize the collected metrics. The median of the numbers of alternatives obtained with RGT is 4 for single-grid data points, and 6 for double-grid data points. Half of the data points for numbers of concerns in student reports (architectural documents), for single-grid data points, were equal to 5, 6, or 7. All ratios of rankings for grids are equal to 1.

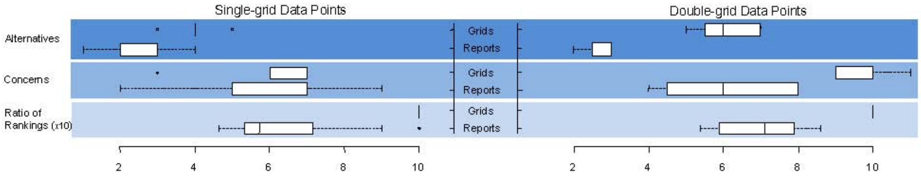


Fig. 1. Boxplots for each metric, for the two types of data points

To answer our research question, we define the following null hypotheses.

H_{0a}: The RGT does not influence the number of explicit alternatives.
H_{0b}: The RGT does not influence the number of explicit concerns.
H_{0c}: The RGT does not influence the ratio of explicit rankings.

The alternative hypotheses are the following.

H_{1a}: The RGT influences the number of explicit alternatives.
H_{1b}: The RGT influences the number of explicit concerns.
H_{1c}: The RGT influences the ratio of explicit rankings.

Next, we compare the means of each metric, for the decisions in grids and architectural documents, by using the Wilcoxon signed ranks test. We test each hypothesis on the single-grid data points and the double-grid ones. Table 1 summarizes the results, for the numbers (#) of alternatives, concerns, and ration, of grids (G) and reports (R).

Table 1. Hypotheses, metrics, means, standard deviations, and p-values for the two samples

		13 Single-grid Data Points			7 Double–grid Data Points		
H	Metric	Mean	Std. Dev.	*p*-value	Mean	Std. Dev.	*p*-value
H_a	# Alternatives G	4.00	0.41	0.002	6.14	0.90	0.016
	# Alternatives R	2.62	0.77		2.71	0.49	
H_b	# Concerns G	6.00	1.41	0.720	9.57	0.79	0.017
	# Concerns R	6.23	1.88		6.14	1.86	
H_c	Ratio G	1.00	0.00	0.003	1.00	0.00	0.018
	Ratio R	0.66	0.19		0.70	0.13	

Given the *p*-values less than 0.05 and the means values, we can reject H_{0a}, and accept that applying the RGT increased the number of explicit alternatives, for both single- and double-grid data points. Regarding H_{0b}, we cannot find any influence of RGT, for single grids, due to the high *p*-value. However, we can reject H_{0b} for the sample with double grids. Additionally, we reject the third null hypothesis (H_{0c}), as the *p*-values are low.

The results strongly suggest that one grid contains more explicit alternatives and higher ratio of explicit rankings than the equivalent description from a basic architectural document. Two grids seem to contain not only more explicit alternatives and rankings, but also more concerns.

4.2 Post Questionnaires

From post questionnaires, we learned that participants had a bachelor degree in Computer Science or a related field (e.g., Information Technology). Half of the participants had an average of around 2 years of work experience in software industry. Students needed an average of 24 minutes for each grid session, with a standard deviation of 11 minutes.

We asked students to rate some statements on a Likert scale from 0 to 4 (strongly disagree, disagree, neutral, agree, strongly agree) that referred to their understanding of the study. We learned that participants understood the presentation on the RG technique. They also understood the directions for creating the grids, and perceived the Idiogrid tool [6] as easy to use. Students perceived the first grid as easier to do, than the second one. A possible explanation for the difference may be that students applied RGT on the most familiar topic first, followed by the less familiar one. Participants indicated that they clearly understood the tasks. Also, they partly enjoyed the assignment.

Overall, post questionnaire results suggest that participants did not face significant issues in using RGT in a self-administrated manner, after a short introduction to it. We believe that the smooth learning curve and low time cost may facilitate RGT adoption by practitioners.

5 Discussion

RGT tends to elicit a higher number of alternatives, compared to basic architectural documents, which usually mention 2-3 alternatives. Additionally, RGT delivers 100%

of possible rankings for a decision, due to the systematic steps for capturing decisions. In contrast, architectural documents seem to make explicit only around 70% of rankings, as participants used no systematic technique for capturing decisions.

On average, one grid contained around 6 concerns, similar to decisions from the architectural documents. However, double grids contained around 9 concerns. Combining RGT-elicited concerns by 2 out of the 5 members of a team increased significantly the number of explicit concerns. Therefore, we believe RGT may be useful for architectural reviews, to help uncover more concerns from stakeholders.

We found out that participants spent an average of 24 minutes to capture a decision with RGT. In our previous study [10], participants captured a decision in 57 minutes, on average. We believe the difference is mainly due to the approach for eliciting concerns. As reported by Grice et al. [5], we also noticed that sentence completion seems to be more user friendly than triadic elicitation. However, we speculate that the triadic elicitation approach asks the expert to reflect more, with the potential to unearth more in-depth tacit AK than the sentence completion approach. We consider both approaches are valid and useful, as they have complementing qualities.

Regarding validity threats, Edwards et al. [3] offer criteria for evaluating a study that uses RGT, such as supplied vs. elicited elements or concerns. Our study used full individual repertory grids, as both elements (alternatives) and constructs (concerns) were elicited from each participant. This is especially well-suited for exploratory situations, like capturing tacit knowledge [3]. However, we decided to use a less known approach for construct elicitation, although more user-friendly: sentence completion [5]. To address the risks of using a less-established approach, we piloted it before the study, to make sure that sentence completion provides useful outputs.

Regarding the external validity of our study, we consider the study participants as representative for inexperienced software architects. Half of the graduated students had around 2 years of working experience. However, we cannot generalize our results to experienced architects. Moreover, we do not know if the decision descriptions in the architectural documents from students are representative for the industry. Concerning internal validity, the main issue is the history of the decision descriptions from the architectural documents. Students thought about the decisions, and created their descriptions as team work, while grids as individual work. We partly addressed this risk, by dividing the grids based on the number of students in the same group, who worked on the same decision topic. Additionally, the structure of the basic decision template influences the resulting decision descriptions. Different templates may result in different metrics and different information. Therefore we cannot claim that RGT provides better results when compared to any type of decision documentation. However, students repeatedly refined their decisions' descriptions in the architectural documents, as part of the course. In contrast, the students did not have time to refine their grids.

6 Conclusions and Future Work

The goal of our study was reducing AK vaporization. To achieve it, we used RGT on some architectural decisions. We also investigated how RGT compares to a basic approach to documenting architectural decisions. Specifically, we analyzed metrics on important parts of a decision: alternatives, concerns, and rankings. Also, content analysis of architectural documentation provided measurements for descriptions of decisions. We learned that grids seem to capture more alternatives and more rankings, compared to architectural documents.

We consider that although tool support for RGT exists, it is not geared towards capturing AK, leaving room for future improvements. Moreover, we need to refine metrics for AK vaporization, to help evaluating approaches for reducing it. Additionally, we plan to investigate the reuse of captured AK through RGT in order to obtain a good return on investment for the spent effort.

Acknowledgments. This research has been partially sponsored by NWO SaS-LeG, contract no. 638.000.000.07N07. We thank study participants, David Ameller, Uwe van Heesch and Pavel Bulanov for their help.

References

1. de Boer, R.C., Farenhorst, R.: In Search of 'Architectural Knowledge. In: Proc. Third Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent, pp. 71–78 (2008)
2. Bosch, J.: Software architecture: The next step. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 194–199. Springer, Heidelberg (1985)
3. Edwards, H.M., McDonald, S., Young, S.M.: The repertory grid technique: Its place in empirical software engineering research. Information and Software Technology 51, 785–798 (2009)
4. Fransella, F., Bell, R., Bannister, D.: A Manual for Repertory Grid Technique, 2nd edn. Wiley, London (2004)
5. Grice, J., Burkley, E., Burkley, M., Wright, S.: J: A sentence completion task for eliciting personal constructs in specific domains. Personal Construct Theory & Practice 1, 60–75 (2004)
6. Grice, J.W.: Idiogrid: software for the management and analysis of repertory grids. Behavior Research Methods 34, 338–341 (2002)
7. Jansen, A.: Architectural design decisions, PhD thesis, University of Groningen, Netherlands (2008)
8. Kruchten, P., Capilla, R., Dueñas, J.C.: The decision view's role in software architecture practice. IEEE Software 26, 36–42 (2009)
9. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. Quality of Software Architectures, 43–58 (2006)
10. Tofan, D., Galster, M., Avgeriou, P.: Capturing Tacit Architectural Knowledge Using the Repertory Grid Technique (NIER Track). In: Proceedings of the 33rd International Conference on Software Engineering, Honolulu, USA, pp. 916–919 (2011)

Guiding Architects in Selecting Architectural Evolution Alternatives

Selim Ciraci[1], Hasan Sözer[2], and Mehmet Aksit[3]

[1] Pacific Northwest National Lab. Richland, WA, USA
selim.ciraci@pnl.gov
[2] Özyeğin University, İstanbul, Turkey
hasan.sozer@ozyegin.edu.tr
[3] University of Twente, Enschede, The Netherlands
m.aksit@ewi.utwente.nl

Abstract. Although there exist methods and tools to support architecture evolution, the derivation and evaluation of alternative evolution paths are realized manually. In this paper, we introduce an approach, where architecture specification is converted to a graph representation. Based on this representation, we automatically generate possible evolution paths, evaluate quality attributes for different architectural configurations, and optimize the selection of a particular path accordingly. We illustrate our approach by modeling the software architecture evolution of a crisis management system.

1 Introduction

To add new features, to adopt new technologies, or to improve quality factors, software systems evolve [1]. Usually there exist multiple evolution alternatives, which should be evaluated rigorously at a high abstraction level [2]. In architecture evaluation methods like ATAM [3], trade-off analysis is a manual process. Tools and techniques are developed to guide the architects in planning and carrying out architecture evolution [2,4,5]. These, however, either do not support trade-off analysis or require the manual generation of evolution alternatives.

In this paper, we propose a method and the accompanying tool set to aid the user in selecting the "best" evolution alternative. Hereby, the architectural changes (anticipated/common evolutions) are modeled in xADL [6] in ArchStudio [7], with proposed extensions that specify how the architectural elements are changed (e.g., added, removed) and their impact on quality attributes. Our tools convert these specifications to graph transformation rules and store them in a repository. The user specifies the desired types of evolutions and inputs the initial architecture (also specified in xADL). Our tools, first, convert the initial architecture to a graph and, then, fetch architectural changes (from the repository) whose categories match the desired evolutions. Next, the evolution alternatives are generated with a graph transformation tool. Quality attributes are evaluated for each evolution alternative based on the impact of different

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 252–260, 2011.

architectural changes. Finally, the user can query and score the evolution alternatives according to the desired structural and path properties. An optimizer uses these scores and quality attributes to select the best evolution alternative. We illustrate our approach by modeling the software architecture evolution of a crisis management system.

This paper is organized as follows: next section introduces a motivating example, which will be used throughout the paper for illustration purposes. Section 3 explains the modeling of architectural changes. Section 4 explains the generation of the architectural alternatives and the selection of the optimal evolution alternative. Section 5 discusses the case study. Finally, conclusions are provided in Section 6.

2 Motivating Example

A crisis management system (CMS) [8] is used for managing the resources (e.g., ambulances) to aid in crisis situations. To support coordination of crisis resolution processes and efficient allocation of resources, the CMS architecture design incorporates programmable crisis managers called *scenarios* and resource allocation strategies. Figure 1 shows the architecture design of the CMS with only one scenario, the car crash scenario, and one allocation strategy, i.e., first come first serve (FCFS). In principle, there can be many scenarios and allocation strategies implemented as separate components at the corresponding layers. The users or the mobile devices communicate with the CMS through the *Crisis Manager* component. The *ScenarioCtrl* interface is used for relaying requests and events to the connector *Scenario Control Interface*. This connector decodes and forwards the received events to corresponding scenario(s) through interface dedicated for different types of outside events. Scenarios allocate resources through the connector *ResourceManager*. This connector sends resourse allocation messages to components representing the resource allocation strategy. The resource allocation strategy finds the best candidate resource from the list of resources and grants the allocation.

Anticipated Type of Evolutions: Anticipated evolutions for CMS include the addition of new scenarios and resource allocation strategies. These evolutions can be incorporated to the architecture in different ways. For example, assume that we want to add a new scenario called *presidential emergency*. This scenario has

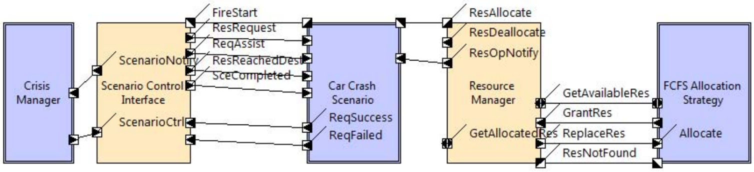


Fig. 1. A (partial) CMS architecture design.

a higher priority; so when a report about a presidential emergency comes and there are no resources available, an ongoing car crash scenario needs deallocate resources. We can add this scenario in two ways: *i)* We can extend connector *ResourceManager* to receive an outside event that causes other scenarios to release resources, *ii)* we can add a new resource allocation strategy. In the former case some of the resource allocation tasks are handled by the scenarios which might not be desirable. The later might be more costly as a new allocation strategy needs to be implemented. Such alternatives together with their pros and cons must be studied not to implement a suboptimal or undesired change.

3 Preparation Steps

As preparations steps, architectural changes are modeled. These are anticipated/repeating changes that are specific to the architecture of the software system at hand. We use the *Arch-Studio* tool [7] to describe such changes in *xADL* [6]. Hereby, changes are specified as partial structures annotated with operations, i.e., add/remove elements. Our tool set converts these specifications to the corresponding graph transformation rules. In this way, the graph system and the underlying theory are hidden from the users.

An architectural change that is represented as a graph transformation rule has a left-hand side (LHS), a right-hand side (RHS), a name, and a type. Therefore, each architectural change is specified in 4 steps. First, a name for the change is specified. Second, the architectural elements required by the change is modeled; this is analogous to the LHS of a transformation rule, e.g., if we want to add a new interface to the connector *Scenario Control Interface*, this connector is required and as such should be modeled in the architecture change model with the same name to be able to apply the change. Third, the transformation executed by the architectural change is modeled using operators that add/remove elements. An architectural element whose name starts with *new:* is added, *delete:* is removed, *not:* should not be in the architecture for the change to be applied. The elements with operators, and the elements from the second step forms the RHS of the transformation rule. For example, the architectural change *AddEventFirePreEmpt* adds a new interface called *FirePreEmpt*, which is modeled by the interface *new:FirePreEmpt* in Figure 2. We do not need to apply this change, if the connector *Scenario Control Interface* has already an interface named *FirePreEmpt*. We model this with the interface *not:FirePreEmpt*.

In the forth (last) step of specifying an architectural change, type of the architectural change is defined. This type is specified as a set of keywords describing what the architectural change does. It is possible to have more than one change for a type. Once the architectural change is modeled, it is stored in the repository managed by the CDE tool [9]. CDE tool first converts the change modeled in xADL to graph-transformation rules

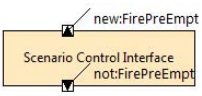


Fig. 2. An architectural change

and, then, indexes it according to the specified type (the tools described in this paper can be located at http://sourceforge.net/projects/caae/).

Quality Attributes: In order to capture these quality attributes in architectural changes and models, we have extended the existing xADL schema with new types; namely reliability interface and cost component. The concept of an interface with analytical parameters has been borrowed from [10]. The reliability interface is added to components/connectors and it consists of a reliability value R, where $0 < R < 1$. The cost component is a special component to which interfaces can not be added and which is named **Cost**.

Reuse of Architectural Changes: To promote reuse of architectural changes, our approach supports parameterization of names/descriptions. The idea behind parameterization is to keep the structure of the change fixed, but make the names variable. Therefore, the need of modeling new architectural changes for repeated evolutions, evolutions using the same change structure with different names, is eliminated. For example, adding an interface to the connector *Scenario Control Interface* is a common evolution. However, we need to model a new architectural change for each new interface, similar to the one illustrated in Figure 2, as the names of the new interfaces would differ. With parameterization, we can make the description of the added interface a parameter and store the architectural change in the repository as a **template**.

A parameterized name/description starts with "@" symbol. Thus, we can make the architectural change of Figure 2 a template by changing the name of the added interface from *FirePreEmpt* to *@outsideControlEvent*.

4 Automated Steps

In the automated steps the evolution alternatives are generated, which starts with the designer specifying the type of the desired evolutions. These are input to the CDE which, then, fetches all architectural changes whose type matches to the specified type and prepares the graph system.

Binding Changes to Architecture: After locating the architectural changes matching the desired evolutions, CDE makes a list of parameters these take. Each parameter is presented to the designer in the following format: *@<parameter Name>* _ *<archtecturalChangeName>* = . At the RHS of the assignment, the designer fills the actual descriptions/names of the architectural elements. In case an architectural change is used multiple times, a unique instance name for each usage should also be specified at the right-hand side of the assignment.

Once the list is filled, *CDE* uses it to **bind** the architectural changes to the current architecture by substituting the parameters with the values specified at the RHS of the assignment. For example, with the specified value *@outsideControlEvent_addNewOutsideEvent = FirePreEmpt*, CDE replaces the parameter *@outsideControlEvent* with the value *FirePreEmpt* in the architectural change *addNewOutsideEvent*.

Generation of the Alternatives: After the binding, CDE forms the graph system containing the architectural changes as graph transformation rules and the architectural model as a graph. These are loaded to GROOVE which automatically generates the state-space, which has an initial state and final states. The initial state in our case is the input architectural model. The final states are states where none of the transformation rules match and, as such, they do not have any outgoing edges. These states contain the evolved architectural models; each final state is an **evolution alternative**. The path from the initial state to a final state, shows the applied architectural changes to reach that final state. Hence, we term such paths **evolution paths**.

Figure 3 shows the state-space generated from the architectural changes *AddFirePreEmpt* (Figure 2) and *addInterfacePreEmptHandler*. The later architectural change adds the interface *AddFireEmpt* and a new component *FireEventHandler* that only receives messages from this interface. This state-space contains two evolution alternatives, states *s2* and *s3*. We can see that state *s2* results from the application of the architectural change *addInterfacePreEmptHandler*. To reach the state *s3*, on the hand, two architectural changes are applied: the change *AddEventFirePreEmpt* and the ender change *AddReceiverFirePreEmpt*.

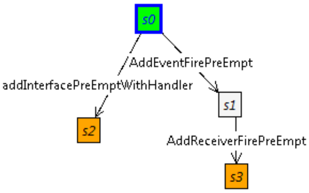


Fig. 3. The evolution paths for adding an interface

Querying/Scoring Alternatives: When the alternatives are generated, it is important to find the ones that have the desired structure and/or generated through a desired evolution path. For this, we have developed a querying mechanism, where the designers can query and give scores to the desired structure and/or architectural changes. Hence, the alternatives with the higher scores have a higher chance of being selected by the optimizer.

Two types of queries are possible with our system. The first type allows the designers to express a structure that is searched in the generated evolution alternatives. These are expressed as Prolog queries; we have extended GROOVE with a Prolog interpreter (using GNU-Prolog Java [11]) and implemented 8 rules that allows the user express queries based of the architectural models. When expressing the queries only the name parameter should be filled and a query should start with the rule *evolutionAlternative*. For example, we want the evolution alternatives that has a single component handling the events *FirePreEmpt* to have a high score. With, the following query we can locate such alternatives:

```
evolutionAlternative(S),connector("Scenario Control Interface",S,T)
component("PreEmpt Handler",S,C),outInterface("FirePreEmpt",T,I),
inInterface("FirePreEmpt",C,I2),link(I,I2,S),score(S,1)
```

The query above finds and gives the score 1 to all the evolution alternatives, which have a connector named *Scenario Control Interface* and a component named *PreEmpt Handler* linked to each other through the *FirePreEmpt* interface.

The second type of queries allows the designer to express desired/contraints over the evolution paths. These are expressed using Computational Tree Logic (CTL) [12]. Informally, a CTL formula consists of atomic propositions that are ordered with logical and temporal operators. In GROOVE, these automatic propositions consists of the labels of the transitions in the state-space; that is, the names of the applied architectural changes. We extended GROOVE's CTL evaluator to score the evolution alternative that occur after finding states that follow the formula. For example, the evolution alternatives generated from the evolution path where first the architectural change *addInterfacePreEmptHandler* followed by the architectural change *AddEventFirePreEmpt*, should get a low score because these contain the same interface added twice. We can query the evolution alternatives generated from such a path with the following formula:

$\text{(addInterfacePreEmpt} \wedge \text{(EF(AddEventFirePreEmpt)))}$
Score(-1)

Here, E means there exists at least one path and F means finally. The proposition *Score* is only used for expressing the given score and has no effect on the CTL formula. The path score is treated differently than the structural scores and by default both scores for all evolution alternatives are set to 0.

Calculation of Quality Attributes and Optimization: The cost of evolution for an alternative is calculated by summing up the estimated cost values of the architectural changes in its evolution path. Initially the cost is set to 0 as an attribute of a special node in the graph representation. Each architectural change is associated with a cost value. During the generation of the evolution path, the corresponding cost value is added by transformation rules for each applied change.

To estimate the reliability of a particular design alternative, each component is annotated with a reliability value R, where $0 < R < 1$. Based on these annotations and the connections among the components, reliability values are propagated through the interfaces. The reliability of a component is derived by multiplying its reliability value with the reliability values of other components on which it depends. The reliability measure for the overall system can be considered as a combined reliability of the components that interact with the user.

The selection of an evolution path considering multiple quality attributes and criteria (in this case, reliability, cost, structural query score and path query score) requires us to solve a multi-criteria optimization problem. The optimizer tool currently employs single-objective optimization techniques. As such, the objective function is specified as: maximize reliability such that the cost, structural query score and path query score are below/above certain thresholds.

5 Application of the Approach

In this section, we will consider two common evolutions to CMS and show how our approach helps the designers in selecting an evolution alternative. The first evolution is the addition of the scenario *Presidential Emergency Scenario* which requires an extra pre-emption event with a new crisis manager component. We designed 5 architectural changes that can be used to add a new crisis manager with an event. These changes are not specific to the evolution of the presidential emergency scenario but are rather alternative ways of adding a new crisis manager and an event. The second evolution is the addition a resource allocation strategy based on the location of the resources which requires a new resource allocation component and a new resource location update event. We designed 6 architectural changes that can be used to add a new event and a new resource allocation strategy. We have checked in all these to the repository, then followed the automated steps to generate the evolution alternatives.

Generating the evolution alternatives: We checked out 11 architectural changes from the repository. We used the "full" CMS architectural model, which includes 20 scenarios and 5 resource allocation strategies; the graph model generated from this model contains 647 elements. With the 11 architectural changes, 23 evolution alternatives that can be reached by 41 evolution paths are generated by GROOVE (in 21 seconds with a dual core 2.3Ghz laptop). The number of evolution paths are greater than the number of evolution alternatives because some paths lead to the same alternative; GROOVE can detect and merge isomorphic states.

Pruning the alternatives: We want both the pre-emption and the resource location update events to be sent from the *outside* to the CMS. In addition to this, we prefer alternatives that use new components to propagate/handle these events as these alternatives are more modifiable (pre-emption is separated from scenarios). As the alternatives that have new components to handle these events are more modifiable, we give such alternatives the score 2 with the following query:

evolutionAlternative(S), connector("Scenario Control Interface", S, T), outInterface("FirePreEmpt", T, I1), outInterface("ResLocUpdate", T, I2), component("Resource Event Handler", S, C), inInterface("FirePreEmpt", C, IIn1), inInterface("ResLocUpdate", C, IIn2), link(I, IIn1, S), link(I2, IIn2, S), score(S, 2)

The architectural change that adds a new component to propagate the resource location update should be followed by the change that links this new component with the the connector *Resource Manager*. We can ensure that paths do not follow this constraint to get a low score with the following CTL formula:

(AddAllocationStrategyHandlerComponent ∧ !(EF(AddAllocationStrategyHandlerConnection)))
Score(-1)

Optimization and the selected alternative: Figure 4 presents an excerpt from the alternative selected by the optimizer, whose goal was to find the alternative with the lowest cost and the highest reliability, structure and path scores. Here, the resource location update and pre-emption events are send from the interfaces *ResLocUpdate* and *FirePreEmpt*.

This alternative has the highest reliability because crisis handling and resource updates are separated.It has the second highest cost; however, it has the highest structural (because, score 2 is given to the alternatives that use a differ-

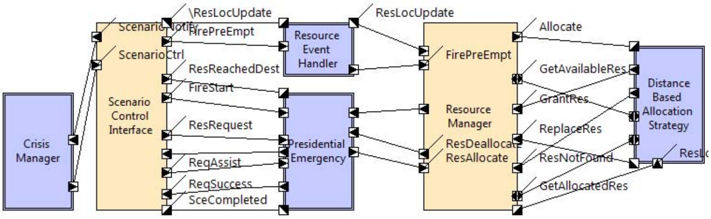


Fig. 4. The evolution alternative that uses the component *Resource Manager* to propagate the events about resources

ent component to propagate the events) and path (because, it does not violate the specified constraint) scores.

Discussion: In this application, 2 criteria were considered to evaluate 23 alternatives and 41 paths. The number of criteria and alternatives can be much higher for other systems. As such, manual generation and evaluation of these alternatives can be overwhelming. Moreover, it can be hard to differentiate among the isomorphic alternatives and reduce the design space manually as the number of alternatives increase.

6 Conclusion

We have introduced an approach for fostering reuse and automation in software architecture evolution. We have formalized architectural changes as graph transformation rules, which can be automatically applied on a graph representation of a software architecture. Thereby, our tools can generate possible evolution paths. Our toolset converts architecture descriptions and architectural changes specified in xADL to the corresponding graph representations and graph transformation rules, respectively. As such, the underlying theory is completely transparent to the designer. In addition to this, the evolution alternatives and paths can be queried and the alternatives that follow these queries can be scored. These scores and the quality attributes, such as cost and reliability are used by an optimizer to select the best evolution alternative.

References

1. Lehman, M., et al.: Metrics and laws of software evolution. In: METRICS, pp. 20–32 (1997)
2. Garlan, D., et al.: Evolution styles: Foundations and tool support for software architecture evolution. In: WICSA, pp. 131–140 (2009)
3. Kazman, R., et al.: The architecture tradeoff analysis method. In: ICECCS (1998)
4. Grunske, L.: Formalizing architectural refactorings as graph transformation systems. In: SNPD, pp. 324–329 (2005)
5. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. 44, 133–155 (2002)
6. Dashofy, E., van der Hoek, A., Taylor, R.: A highly-extensible, xml-based architecture description language. In: WICSA, p. 103 (2001)

7. Dashofy, E., et al.: Archstudio 4: An architecture-based meta-modeling environment. In: ICSE, pp. 67–68 (2007)
8. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis management systems: A case study for aspect-oriented modeling. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on Aspect-Oriented Software Development VII. LNCS, vol. 6210, pp. 1–22. Springer, Heidelberg (2010)
9. Ciraci, S., van den Broek, P., Aksit, M.: Framework for computer-aided evolution of object-oriented designs. In: COMPSAC, pp. 757–764 (2008)
10. Grassi, V., Mirandola, R., Sabetta, A.: An XML-based language to support performance and reliability modeling and analysis in software architectures. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, pp. 71–87. Springer, Heidelberg (2005)
11. Gnu prolog java, `http://www.gnu.org/software/gnuprologjava/`
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)

Architecture-Based Run-Time Fault Diagnosis

Paulo Casanova[1], Bradley Schmerl[1], David Garlan[1], and Rui Abreu[2]

[1] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{paulo.casanova,schmerl,garlan}@cs.cmu.edu
[2] Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

Abstract. An important step in achieving robustness to run-time faults is the ability to detect and repair problems when they arise in a running system. Effective fault detection and repair could be greatly enhanced by run-time fault diagnosis and localization, since it would allow the repair mechanisms to focus adaptation effort on the parts most in need of attention. In this paper we describe an approach to run-time fault diagnosis that combines architectural models with spectrum-based reasoning for multiple fault localization. Spectrum-based reasoning is a lightweight technique that takes a form of trace abstraction and produces a list (ordered by probability) of likely fault candidates. We show how this technique can be combined with architectural models to support run-time diagnosis that can (a) scale to modern distributed software systems; (b) accommodate the use of black-box components and proprietary infrastructure for which one has neither a specification nor source code; and (c) handle inherent uncertainty about the probable cause of a problem even in the face of transient faults and faults that arise only when certain combinations of system components interact.

Keywords: Autonomic computing, diagnosis, software architecture, run-time.

1 Introduction

With increasing reliance on software-based systems to support virtually all aspects of our daily lives, an important new requirement for these systems is the ability to detect and resolve problems at run time. This requirement has spawned an active area of research in autonomic computing [19,6,12].

Autonomic computing is based on the idea of turning ordinary software systems into closed-loop control systems. That is, systems are monitored to provide observations of their run-time behavior. Those observations are then analyzed at run-time in reference to models of desired or expected behavior. If significant deviations are observed, repair actions are executed to correct the problems.

When designing an autonomic system, a key issue is the kinds of models that are used in the control layer at run time to capture observed system behavior and detect problems. Research carried out over the past decade has demonstrated that software

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 261–277, 2011.

architectures can be particularly effective in this capacity [14,16,27].Architectures provide a high-level view of the system, reducing the complexity of understanding what the system is doing, and supporting scalability to complex distributed systems. Suitably annotated architectures also permit the autonomic decision-making apparatus to detect the presence of systemic problems and trends, such as degraded performance. Finally, architectures allow one to capture common patterns of repair that are tuned to the specific style of system and its implementation.

While such "architecture-based self-adaptation" shows great potential, one outstanding problem is *diagnosis:* determining the likely causes of a detected problem. By narrowing the scope of concern for repair to candidates that are the probable cause of an observed problem, the ability to effectively adapt to a problem can be greatly increased.

Run-time diagnosis for architecture-based self-repair, however, is particularly challenging. First, the presence of concurrency makes it difficult to identify which of many possible computations might have caused a problem. Second, reliance on middleware for distributed communication, and more generally the use of components and infrastructure produced by many organizations, means that in many cases neither specifications nor code is available for all parts of the system. Third, in many systems, problems may be intermittent, caused by transient faults or variability in loads. Moreover, while some faults may be directly traceable to a single component (such as a crashed server), in general the source of a problem may be a result of certain combinations of elements (e.g., a specific server interacting with a specific database).

In this paper we describe a systematic approach that adapts a reasoning technique called *spectrum-based multiple fault localization* (SMFL) to architecture-based self-repair. SMFL is a lightweight technique that takes as its input a form of trace abstraction and produces a list of likely fault candidates, ordered by probability of being the true fault explanation [1,4]. Impressive diagnostic results for *design-time* testing and debugging of both hardware and software systems have been achieved using SMFL [1]. However, there has been little work in applying these results in the context of *run-time* detection and repair, especially in the context of architecture-based adaptation. As we describe in the remainder of this paper, key features of our approach include: (a) the ability to define at a high level what kinds of behavior to use as the basis for fault localization; (b) the ability to associate such behaviors with families of systems (or architectural styles), allowing reuse of the specifications for all instances of the family; and (c) the ability to take into consideration quality attributes, such as performance and availability, in determining both the presence and cause of a problem.

2 Related Work

One of three approaches to fault diagnosis has been typically adopted by autonomic systems. One is to use simple heuristics. For example, software rejuvenation [21,30] is a technique where components are selectively restarted, using the heuristic of choosing the longest running (i.e., oldest) components to reboot next. This technique can generally improve the robustness of a system, since in many cases faults occur because parts of a system may degrade over time (due, for example, to memory leaks). However, clearly not all faults in a system are a result of aging. Thus heuristics have the advantage of being easy to calculate and often widely applicable, but they lack precision, resulting in inefficiencies and poor coverage.

A second approach is to develop special-purpose diagnostic mechanisms for a particular class of system and particular classes of faults. For example, recovery-oriented computing [5] uses a form of local rebooting that takes advantage of the particular characteristics of JEE-based systems, where built-in persistence mechanisms allow computations to be terminated and restarted without loss of data. Diagnosis in these systems uses statistical machine learning techniques to identify a specific component to restart – again taking advantage of the specific features of JEE systems. Similarly, the Google File System [15] and Hadoop [8] use fast, local recovery and replication to achieve high availability for scalable distributed file systems for data-intensive applications. These systems use custom-built monitoring and diagnosis to determine failures of individual servers. While such hand-crafted techniques are typically very effective for the specific kind of system they address, (1) they do not generalize to other systems, where the same architectural assumptions do not hold, and (2) they assume single-fault scenarios.

A third approach allocates the task of diagnosis to individual repair handlers. For example, the Rainbow system incorporates a set of repair strategies that are triggered when certain architectural invariants are violated in a running system [7,14]. Each strategy is responsible for determining whether to correct the problem at hand, and if so, how. In order to do this a strategy has to carry out its own fault diagnosis and localization. For example, a strategy triggered by high latencies might attempt to reboot faulty servers. But before it can do that it needs to figure out which servers (if any) might be failing. Associating diagnostics with the repair mechanism has the advantage that diagnosis can be specialized to the needs of the particular kind of repair. But it has the disadvantage that each repair handler must do its own diagnosis, possibly adding to run-time overhead (if multiple strategies are used), greatly increasing the effort required to produce repair handlers, and relying on the strategy writer to get the diagnosis right. Similarly, in the three-layer architecture model proposed in [23] higher level planning mechanisms are responsible for diagnosis *once* a problem has been detected.

None of these techniques provides a general, systematic basis for *run-time* fault diagnosis. In contrast, there has been considerable research on automatic fault diagnosis used at *development time*. Traditionally, automatic approaches to software fault localization are based on using a set of observations collected during the testing phase of system development to yield a list of likely fault locations, which are subsequently used by the developer to focus the debugging process [28]. Existing approaches can be generally classified as either statistics-based or model-based. The former uses an abstraction of program traces, collected for each execution of the system, to produce a list of fault candidates [24,18,25]. The latter combines a *model* of the expected behavior with a set of observations to compute a diagnostic report [11,26].

Model-based approaches are more accurate than statistical ones, but are much more computationally demanding (in both time and space), and they require detailed models of the correct behavior of the system under test. Recently a novel reasoning technique over abstractions of program traces, combining the best characteristics of both worlds, has been proposed [4]. It has low time/space complexity (like statistics-based techniques), yet with high diagnostic accuracy (like reasoning techniques). As we will see, such properties make the technique especially amenable to (continuous) run-time analysis. In this paper, we refer to this kind of reasoning technique as spectrum-based multiple fault localization (SMFL). Previous research efforts into SMFL have focused primarily on helping developers fix bugs at development time, where one can easily identify the start and end of a given test case, as well as which elements were involved in the execution of the test case. To our knowledge, none of these have been combined with architecture models to support run-time diagnosis, where, as opposed to development time, detecting a given execution is difficult (e.g., due to concurrency).

3 Approach

Applying SMFL at run time to diagnose problems relative to architectural models raises a number of challenges. First, we need to be able to identify the beginning and end of computations in the system that we are interested in. This is challenging because interactions may be interleaved and concurrent. Second, we need to be able to relate these run-time interactions, which are in terms of system level events, with their corresponding elements in an architecture model. In this section, we give a brief overview of our approach; in later sections we elaborate on the details.

To illustrate the ideas, consider a family of systems, whose architecture is illustrated in Figure 1, in which a variable number of clients can interact with a pool of servers that have access to a common data store. Client HTTP requests are mediated by one or more dispatchers, selected randomly by a client, which forward requests to a specific server in the pool. Although relatively simple in structure, such systems are representative of a large class of applications, and illustrate some of the challenges for run-time diagnosis. First, such a system could have hundreds of clients and servers (for example, running on a cloud computing platform), handling thousands of simultaneous requests. This makes it challenging to determine which elements are involved with a particular request. Second, when problems occur, it is important to pinpoint the causes quickly, since a problem with a dispatcher (for example), could drastically impact the overall ability of the system to deliver its services in a timely manner. Third, there are many sources of uncertainty inherent in this system. For example, high latency in handling customer requests could be caused by faults, or combinations of faults, in any number of components. Fourth, although certain kinds of problems may be easily detected and fixed (such as a server crash), softer intermittent failures causing high latencies are equally as important to detect and repair.

Our approach to adapting SMFL to support architecture-based fault detection and localization uses the following three steps:

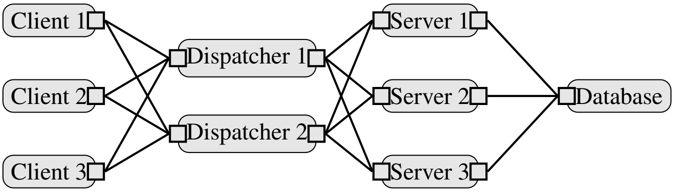


Fig. 1. Simplified Web Server Example

1. First, we define a collection of *transaction types* for the style of system under analysis using parametric architecture behavior descriptions. Each transaction type specifies (a) a set of finite computational paths through the architecture, and (b) the criteria for determining whether a given computation of that type has succeeded or failed. For the class of system shown in Figure 1, one possible transaction type would represent the normal client-server response sequence: a client initiates a request, which is handled by a dispatcher, dispatched to a particular server, and then returned back to the client. A success criterion might be that the client should receive a reply within a certain number of seconds.
2. Next we provide a way to monitor the running system from an architectural perspective, adapting prior work on architecture-based monitoring and fault detection [29]. This involves using probes and event monitoring mechanisms in the running system to determine (a) when a complete transaction has occurred, (b) the architectural elements involved, and (c) whether the transaction succeeded. For the example we would record the specific client, dispatcher, and server involved in the transaction, and whether the latency threshold was exceeded.
3. The results of spectrum monitoring are then accumulated in a fault localization phase. Adapting SMFL algorithms for the run-time setting, probabilistic rankings of likely fault causes (if any) are calculated. These can then be used to trigger repair mechanisms. In the example, the fault localization algorithms might determine, for example, that with probability 0.8 the cause of an intermittent latency problem is the combination of dispatcher 2 interacting with server 5.

To elaborate on this approach, in the following sections we first summarize the key ideas behind SMFL. (For a detailed description see [4].) Then we explain in more detail how we carry out these three parts of our approach.

4 Spectrum-Based Reasoning for Fault Localization

Fault localization based on reasoning over program spectra is characterized by the use of (a) *program spectra*, abstracting from actual observation variables, structure, and component behavior; (b) a low-cost, heuristic reasoning algorithm, STACCATO [4] to extract the significant set of *multiple-fault candidates*; and (c) abstract, intermittent models, that take into account that a faulty component may behave correctly with a specific probability, to compute the *candidate probability* of being the true fault.

4.1 Program Spectra

Assume that a software system is comprised of a set of M components c_j where $j \in \{1,\ldots,M\}$, and can have multiple faults, the number being denoted C (fault cardinality). A *diagnostic report* $D = < \ldots, d_k, \ldots >$ is an ordered set of diagnostic (possibly multiple-fault) candidates, d_k, ordered in terms of likelihood to be the true diagnosis.

A program spectrum is a collection of flags indicating which components have been involved in a particular dynamic behavior of a system. Our behavioral model is represented simply by a set of components *involved* in a computation, and does not have to indicate at a detailed behavioral level exactly what that involvement was. Thus, recording program spectra is light-weight, compared to other run-time methods for analyzing dynamic behavior (e.g., dynamic slicing [22]). Although we work with these so-called component-hit spectra, the approach outlined in this section easily generalizes to other types of program spectra [17].

Program spectra are collected for N (pass/fail) executions of the system. Both spectra and program pass/fail information are input to spectrum-based fault localization. The program spectra are expressed in terms of a $N \times M$ *activity matrix* A, for example in Table 1. An element a_{ij} has the value 1 if component j was observed to be involved in the execution of run i, and 0 otherwise. The pass/fail information is stored in a vector e, the *error vector*, where e_i signifies whether run i has *passed* ($e_i = 0$) or *failed* ($e_i = 1$). Note that the pair (A, e) is the only input to the spectrum-based diagnosis approach.

4.2 Candidate Generation

As in any model-based diagnosis (MBD) approach, the basis for fault diagnosis is a model of the program. Unlike many MBD approaches, however, no detailed modeling is used, but rather a generic component model. Each component (c_j) is modeled in terms of the logical proposition

$$h_j \Rightarrow (ok_{inp_j} \Rightarrow ok_{out_j}) \tag{1}$$

where the booleans h_j, ok_{inp_j}, and ok_{out_j} model component health, and the (value) correctness of the component's input and output variables, respectively. The above *weak* model[1] specifies nominal (required) behavior: when the component is correct ($h_j = \mathsf{true}$) and its inputs are correct ($ok_{inp_j} = \mathsf{true}$), then the outputs must be correct ($ok_{out_j} = \mathsf{true}$). As Eq. (1) only specifies nominal behavior, even when the component is faulty and/or the input values are incorrect it is still possible that the component delivers a correct output. Hence, a program pass does not imply correctness of the components involved.

Table 1. Program Spectra Example

c_1	c_2	c_3	e	
1	1	0	1	obs_1
0	1	1	1	obs_2
1	0	0	1	obs_3
1	0	1	0	obs_4

By instantiating the above equation for each component involved in a particular run (row in A) a set of logical propositions is formed. Since the input variables of each test can be assumed to be correct, and since the output correctness of the final component in

[1] Within the model-based diagnosis community, two broad categories of model types have been specified: (1) weak-fault models, which describe a system only in terms of its normal, non-faulty behavior, and (2) strong-fault models, which also include a definition of some aspects of abnormal behavior.

the invocation chain is given by e (pass implies correct, fail implies incorrect), we can logically infer component health information from each row in (A, e). To illustrate how candidate generation works, for the program spectra in Table 1 we obtain the following health propositions for h_j:

$$\begin{array}{ll} \neg h_1 \vee \neg h_2 & (c_1 \text{ and/or } c_2 \text{ faulty}) \\ \neg h_2 \vee \neg h_3 & (c_2 \text{ and/or } c_3 \text{ faulty}) \\ \neg h_1 & (c_1 \text{ faulty}) \end{array}$$

These health propositions have a direct correspondence with the original matrix structure. Note that only failing runs lead to corresponding health propositions, since (because of the conservative, weak component model) from a passing run no additional health information can be inferred.

As in most MBD approaches, the health propositions are subsequently combined to yield a diagnosis by computing the so-called minimal hitting sets (MHS, aka minimal set cover), i.e., the minimal health propositions that cover the above propositions. In our example, candidate generation yields two double-fault candidates $d_1 = \{1,2\}$, and $d_2 = \{1,3\}$. The step of transforming health propositions into diagnosis is generally responsible for the prohibitive cost of reasoning approaches. However, we use an ultra-low-cost heuristic MHS algorithm called STACCATO [1] to extract only the significant set of multiple-fault candidates d_k, avoiding needless generation of a possibly exponential number of diagnostic candidates. This allows a spectrum-based reasoning approach to scale to real-world programs [4].

4.3 Candidate Ranking

The previous phase returns diagnosis candidates d_k that are logically consistent with the observations. However, despite the reduction of the candidate space, the number of remaining candidates d_k is typically large, not all of them equally probable. Hence, the computation of diagnosis candidate probabilities $\Pr(d_k)$ to establish a *ranking* is critical to the diagnostic performance of reasoning approaches. The probability that a diagnosis candidate is the actual diagnosis is computed using Bayes' rule, that updates the probability of a particular candidate d_k given new observational evidence (from a new observed spectrum).

The Bayesian probability update, in fact, can be seen as the foundation for the derivation of diagnostic candidates in any reasoning approach: i.e., (1) deducing whether a candidate diagnosis d_k is consistent with the observations, and (2) computing the posterior probability $\Pr(d_k)$ of that candidate being the actual diagnosis. Rather than computing $\Pr(d_k)$ for *all* possible candidates, just to find that most of them have $\Pr(d_k) = 0$, candidate generation algorithms are used as shown before, but the Bayesian framework remains the formal basis.

For each diagnosis candidate d_k the probability that it describes the actual system fault state depends on the extent to which d_k explains all observations. To compute the posterior probability that d_k is the true diagnosis given observation obs_i (obs_i refers to the coverage and error information for computation i) Bayes' rule is used:

$$\Pr(d_k|obs_i) = \frac{\Pr(obs_i|d_k)}{\Pr(obs_i)} \cdot \Pr(d_k|obs_{i-1}) \tag{2}$$

The denominator $\Pr(obs_i)$ is a normalizing term that is identical for all d_k and thus need not be computed directly. $\Pr(d_k|obs_{i-1})$ is the prior probability of d_k. In the absence of any observation, $\Pr(d_k|obs_{i-1})$ defaults to $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$, where p denotes the *a priori* probability that component c_j is at fault, which in practice we set to $p_j = p$. $\Pr(obs_i|d_k)$ is defined as

$$\Pr(obs_i|d_k) = \begin{cases} 0 \text{ if } obs_i \wedge d_k \text{ are inconsistent;} \\ 1 \text{ if } obs_i \text{ is unique to } d_k\text{;} \\ \varepsilon \text{ otherwise.} \end{cases} \tag{3}$$

As mentioned earlier, only candidates derived from the candidate generation algorithm are updated, meaning that the 0-clause need not be considered in practice.

In model-based reasoning, many policies exist for defining ε [9]. Amongst the best ε policies is one that uses an *intermittent* component failure model, extending h_j's permanent, binary definition to $h_j \in [0,1]$, where h_j expresses the probability that faulty component j produces correct output. ($h_j = 0$ means persistently failing, and $h_j = 1$ means healthy, i.e., never inducing failures).

Given the intermittency model, for an observation $obs_i = (A_{i*}, e_i)$, the ε policy in Eq. (3) becomes

$$\varepsilon = \begin{cases} \prod\limits_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 0 \\ 1 - \prod\limits_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 1 \end{cases} \tag{4}$$

Eq. (4) follows from the fact that the probability that a run passes is the product of the probability that each involved, faulty component exhibits correct behavior. (Here we adopt an *or*-model; we assume components fail independently, a standard assumption in fault diagnosis for tractability reasons.)

Before computing $\Pr(d_k)$ the h_j must be estimated from (A, e). There are several approaches that approximate h_j by computing the probability that the *combination* of components involved in a particular d_k produce a failure, instead of computing the *individual* component intermittency rate values [3,10]. Although such approaches already give significant improvement over the classical model-based reasoning (see [4] for results), more accurate results can be achieved if the individual h_j can be determined by an exact estimator. To compute such an estimator, h_j is determined per component based on their effect on the ε policy (Eq. (4)) to compute $\Pr(d_k)$. The key idea is to compute the h_js for the candidate's d_k faulty components that *maximizes the probability* $\Pr(obs|d_k)$ *of a set of observations obs occurring, conditioned on that candidate* d_k (maximum likelihood estimation for naïve Bayes classifier d_k). Hence, h_j is solved by maximizing $\Pr(obs|d_k)$ under the above epsilon policy, according to $\underset{\{h_j \mid j \in d_k\}}{\arg\max} \Pr(obs|d_k)$.

To illustrate how candidates are ranked, consider the computation of $\Pr(d_1)$. As the four observations are independent, from Eq. (3) and Eq. (4) it follows

$$\Pr(obs|d_1) = (1 - h_1 \cdot h_2) \cdot (1 - h_2) \cdot (1 - h_1) \cdot h_1 \tag{5}$$

Assuming candidate d_1 is the actual diagnosis, the corresponding h_j are determined by maximum likelihood estimation, i.e., maximizing Eq. (5). For d_1 it follows that $h_1 = 0.47$ and $h_2 = 0.19$ yielding $\Pr(obs|d_1) = 0.185$ (note, that c_2 has much lower health than c_1 as c_2 is not exonerated in the last matrix row, in contrast to c_1). Applying the same procedure for d_2 yields $\Pr(obs|d_2) = 0.036$ (with corresponding $h_1 = 0.41$, $h_3 = 0.50$). Assuming both candidates have equal prior probability p^2 (both are double-fault candidates) and applying Eq. (2) it follows $\Pr(d_1|obs) = 0.185 \cdot p^2/\Pr(obs)$ and $\Pr(d_2|obs) = 0.036 \cdot p^2/\Pr(obs)$. After normalization it follows that $\Pr(d_1|obs) = 0.84$ and $\Pr(d_2|obs) = 0.16$. Consequently, the ranked diagnosis is given by $D =< \{1,2\},\{1,3\} >$.

5 Adapting SMFL to Architecture-Based Run-Time Diagnosis

On the surface of it, combining SMFL with architecture-based adaptation would appear to be a natural synthesis. Architecture models, on the one hand, provide an abstract component-oriented view that can form the basis for a scalable representation of the elements that might contribute to faulty behavior. Further, architecture-based monitoring and fault detection (but not diagnosis) are reasonably well established [29]. SMFL, on the other hand, provides a light-weight, efficient, statistical approach that supports diagnosis in the face of uncertainty, coordinated faults, and transient errors. Further, SMFL is agnostic about the nature of a fault, allowing systemic properties based on quality attributes (such as performance) to guide the ranking procedure.

However, there are a number of obstacles that must be overcome to synthesize these two disciplines. First, there needs to be some way to *define the traces of interest*: one must be able to describe what kinds of computations should be monitored, as well as the criteria for determining whether a computation has succeeded or failed. Moreover, while SMFL expects *finite* traces, in general the behavior of a running system is not finite (or so one hopes). Second, there must be a way to *detect the occurrence of traces in the running system.* As noted earlier, concurrency makes this difficult, since many simultaneously executing traces may be present in a system. (Recall that in the development time context for which SMFL was originally created, each trace can be observed as a separate run of the system under test.) Third, the algorithm for performing SMFL must be adapted to handle concurrently executing traces, and provide an appropriate window of observation (as described later).

5.1 Defining Transactions

Recall that SMFL expects as input a series of spectra, where each spectrum is a finite set of components that participated in a given computation (a finite program trace), together with an indication of its status (pass/fail). How can we define such computations – which will in turn serve as the basis for monitoring, diagnosis, and fault localization?

The problem is non-trivial for two reasons. First, a trace defines a *finite* execution. However, we are interested in systems that operate continuously, so that at a system level the behavior of the system is infinite. Second, different kinds of systems embody very different kinds of computational models. For example, complex computations in a service-oriented architecture (SOA) are often defined by an orchestration script, which indicates how the various components are coordinated, and how data passes from one to another. In contrast, a system based on sensor networks may involve processing streams of sensor readings.

Our approach is based on two key ideas. The first is the idea of a *transaction family*. A transaction family defines a parameterized pattern of behaviors as finite computations, expressed in terms of the architectural elements (components and connectors) that are involved in that computation, and the flow of information/control between them (in a way similar to [20]). An instantiation of that pattern (in terms of specific architectural elements) is an individual transaction. Additionally we associate a set of properties with the components and flows. These properties indicate things like the time that a flow event happened or the load on a server. Finally, a transaction family includes a boolean function that determines whether a given transaction has succeeded.

The second idea is to *associate these transaction families with architectural styles*. An architectural style describes the types of elements and their possible legal associations in a system, which allows architectural patterns referring to those types to be defined. Several transaction families can be defined for each architectural style, each representing a different pattern of computation. Note that the transaction families need not cover *all* of the behaviors of systems in the family – only the ones that are of interest to diagnosis. However, by defining transaction families at the architectural style level, we can immediately reuse diagnosis systems for different systems. And although a different technology may be required to place probes (techniques for probing C programs are different from those for Java programs), both the diagnosis system and its configuration are fully reusable.

There are many possible ways that one might define transaction families, including state machines, process algebras, and so on. In our work we adopt a form of message sequence charts [13]. For example one transaction family for a web-server family that might be used to model the system in Figure 1 would be represented by the sequence diagram in Figure 2. The pattern of communication shown there defines the client-server round-trip execution flow discussed earlier. It involves an arbitrary client, dispatcher, and server, as well as the database. The first three represent parameters of the family which (as we describe below) will be instantiated with specific components during system execution. Properties associated with the family include the time taken to serve a client's request (i.e., the request latency). An associated boolean function returns true if the latency (difference in request and reply times) is under the appropriate threshold.

Transaction families have several important benefits. First, they involve relatively minimal specification: rather than requiring a full formal account of the architectural behavior, we focus only on finite abstract "slices" of it, reducing the overhead of defining relevant behavior and making the approach generally accessible. Second, definition of transaction families for a given system or architectural style can be incremental: it is possible to add new templates or to add detail to an existing template (for example, by including a finer-grained account of the set of elements involved in the transaction). This allows users of the technique to get increased benefits for increased effort. Third, by associating transaction families with architectural families, we amortize the effort of

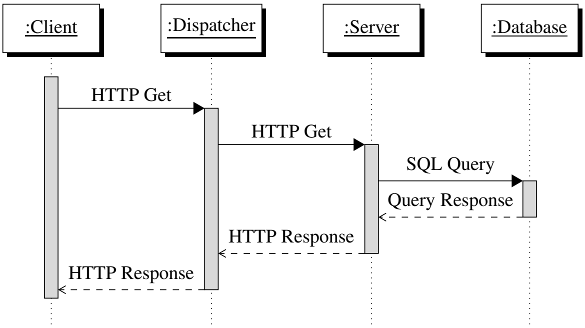


Fig. 2. Example Message Sequence Chart for a HTTP Request Transaction

defining behaviors, and allow reuse of prepackaged collections of transaction families for commonly-used architectural styles.

5.2 Detecting Traces

Given a way to specify transactions, we now need a way to observe them in a running system and then use those observations to carry out fault diagnosis. Figure 3 shows the process that we use do to this.

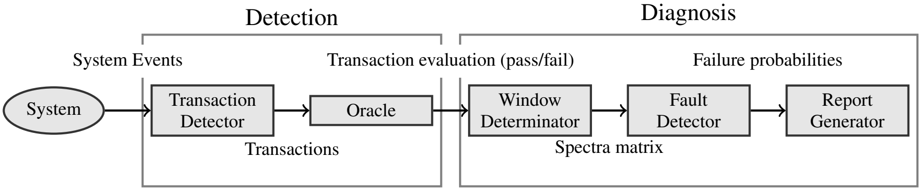


Fig. 3. The Architecture of our Experimental Framework

To detect transactions, we adapt earlier work in architecture-based monitoring [29]. First, a system is instrumented so that it can be monitored at runtime. Monitored events are placed on an “event bus” where they can be consumed by the detection phase of our diagnostic infrastructure. In the client-server example above, monitored events include activities like initiation of HTTP requests over the client-dispatcher connectors.

To monitor a system, there are numerous mechanisms that can be used that vary in terms of the kind of behavior they detect and the kind of system they are appropriate for. For distributed systems, standard middleware and network communication infrastructure provide mechanisms to monitor communication events and their properties. For systems working on a single host, code-oriented monitoring can be used. For example, aspect-oriented techniques can weave monitoring code into an existing code base (see, for example, [29]). In this research the choice of monitoring mechanism and the placement of relevant probes has not yet been a major focus of our efforts. However, as we discuss later, we view this as an area for future research.

During the detection phase, events are first filtered to extract those relevant to the transaction families that are being observed. Next, events are passed to a detection machine generated from the transaction families. Specifically, adapting earlier work on

DiscoTect, we monitor events as a set of concurrent state machines, modeled as Petri Nets [29]. The key idea is that behavior is tracked by moving tokens through a state machine in response to low-level events. When tokens reach certain terminal states the machine emits the set of architectural elements that were involved in the trace and an indication of the transaction type. This information is then fed to an oracle that evaluates the boolean function associated with that spectrum type on the detected spectrum.

5.3 Diagnosis

The second phase of processing is *diagnosis*. This is broken down into three parts. The first part is window determination. This step is responsible for aggregating a sequence of transactions to define the matrices – the (A,e) of Section 4 – that can be analyzed by the SFML algorithms. In determining these matrices it is important to define an appropriate window. If the window is too small, there may be too few transactions for the results to be statistically significant. If the window is too large, it may contain out-of-date transactions that may skew the diagnosis towards past behavior.

There are a number of criteria that might be used to determine this window. In our current experiments we have found that a time-based window works well. That is, we aggregate all spectra within a temporal window. The value for that time bound needs to be determined by experimentation as it is dependent on the rate of system usage: with high transaction rates, a smaller time window can aggregate enough traces, but if the transaction rate is low then we need a larger time window.

Once a window of spectra has been determined, the associated matrices are given to the SMFL algorithm, which calculates a list of candidate fault explanations (if any) ordered by probability of being the likely cause. This is simply a straightforward application of the SFML algorithms described earlier.

Finally the results are passed to a Report Generator, which outputs the results of the SFML analysis: a list with sets of failed components and their associated probabilities. Automated repair mechanisms (or human operators) can then interpret the results in architectural terms.

6 Evaluation

To evaluate the approach we conducted experiments on a system similar to the example in this paper. In particular, we investigated the hypothesis that the technique could accurately pinpoint problems in a system exhibiting (a) variability in the number of components; (b) distributed system structures that involve realistic, off-the-shelf communication infrastructure and componentry; (c) the presence of transient faults, where failure is based on systemic attributes like end-to-end performance; and (e) faults that might involve more than one component. The combination of these properties yields a system that would be challenging to diagnose given current technology.

One class of problems that fits this criteria are intermittent multi-component faults with additional noise. These problems arise with faulty network connections or application errors that occur only with specific combinations of input data. With these kinds of problems, a fault occurs only sometimes and is generally associated with a specific path on the system. However, other intermittent faults may occur less often due to other

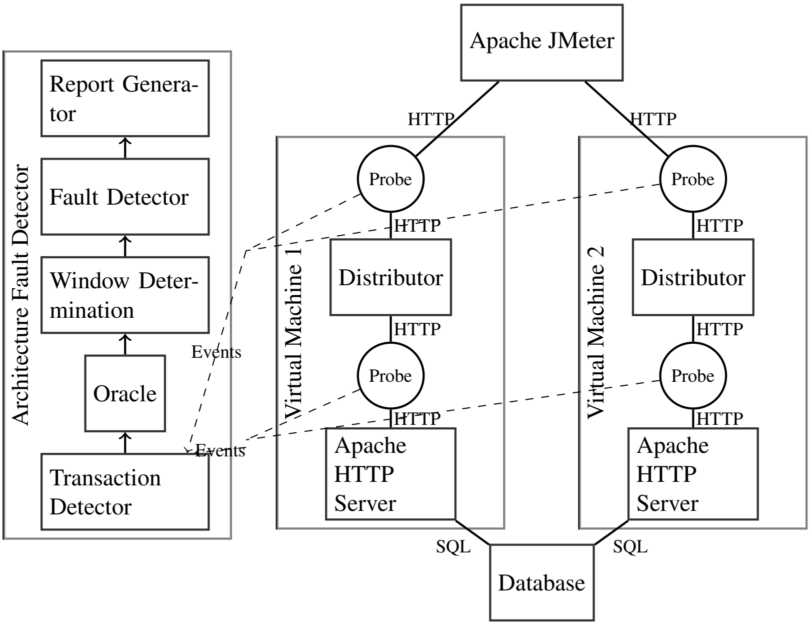


Fig. 4. Evaluation Experiment Setup

reasons, generating additional noise in the spectra. We want to be able to separate out the real errors from the noise.

To create this experiment, we recreated an environment similar to the one Figure 1. In this system, two virtual machines (simulating two servers) run two web servers and dispatchers. The dispatchers choose which web server to send requests to using a round-robin algorithm. An external multi-threaded load test program, *Apache JMeter*, generates requests on both virtual machines simulating clients accessing the system.

A trace family is defined for this system: a *standard* request in which the client performs a request to a dispatcher which forwards it to a web server.

Two interception points, or *probes*, were placed in each machine, one before the request arrives at each dispatcher and one between the dispatchers and the web servers. These interception points (custom-developed based on the *pygmy HTTP server*) add a specific header to the HTTP request to allow tracking the transaction and report to an event bus all events with the component name.

The fault detector receives events from the event bus and uses a Petri Net (PN) to determine to what family the transaction belongs, as previously discussed. The PN used to identify the transaction family is the one in Figure 5. Transitions on the PN are enabled when the corresponding events arrive. A transaction in the PN is initialized with a token in the `START` place and ends when a token arrives at the `DONE:Standard` place. The oracle considers a transaction to be a success if the time elapsed between the request and response is less than 2.5 seconds. This is representative of systems in which response time is a measure of success – systems that do not exhibit easier-to-detect fail-stop failures.

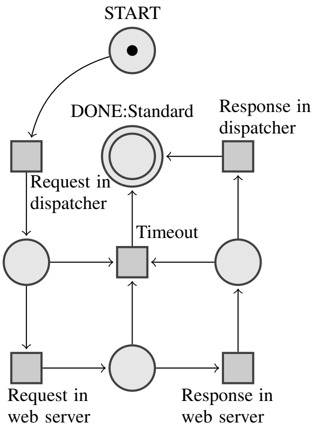


Fig. 5. Petri Net used to identify the transaction family

To simulate network delay (or server processing delay), we added a random delay in both *IP1* and *IP2* of the first virtual machine. This means that 25% of all requests receive an added time delay that ensures some of the time they will fail. This generates a hard-to-find problem, namely an intermittent failure on one of the paths: the one containing the *dispatcher 1* (D1) and *web server 1* (WS1). Simultaneously it adds a small (but non-zero) failure probability on both the D1-WS2 and D2-WS1 paths. The experimental results show that the fault detection algorithm is able to statistically separate these results and produce the correct output.

The total number of traces obtained during a run and their distribution between the various components is shown in Table 2(a). As the results show, the D1-WS1 path fails 26% of the time. The other two paths which include D1 and WS1 fail slightly less than 5% of time time and the D2-WS2 path has no failures.

Because SMFL's only input are the spectra, enough data need to be collected before the problem can be detected. In fact, as shown in Table 2(b), the failure probabilities change over time. For example, a 20s window would have determined that WS1 was the only component responsible for the observed failures. Only after 30s is the algorithm able to indict WS1 *and* D1 as the components responsible for the observed failures. This means that window size needs to be carefully chosen so that the SMFL algorithm has enough information to yield accurate diagnosis [2].

Table 2. Results

(a) Number of success/fail spectra for each combination of dispatcher and web server.

	WS1	**WS2**	**Total**
D1	85/31	129/5	**214/36**
D2	117/5	122/0	**239/5**
Total	202/36	251/5	**451/41**

(b) Time evolution of results of failure diagnosis.

Time Window	**Succ./Fail.**	**Diagnosis**
0-10s	30/2	$D1 : 84\%, WS1 : 16\%$
0-20s	119/8	$WS1 : 100\%$
0-30s	201/16	$D1, WS1 : 99\%, D1, D2 : 1\%$

7 Conclusions and Future Work

In this paper we described an approach that combines architecture models for monitoring system behavior and spectrum-based fault localization for diagnosing problems. Such a combination provides a systematic, efficient and scalable technique to deal with run-time failures independent of the system domain. Important features of the approach are that it is lightweight, generally applicable to any kind of system, tolerant of uncertainty, and capable of detecting soft anomalies and problems that involve combinations of components.

This line of research raises a number of research questions requiring further investigation. In our current system, probes are manually placed to monitor the activity of the running system. We plan to investigate methods for efficient automatic probe placement, including analysis to identify the minimum set of probes required to accurately monitor the spectrum types defined for the system, as well as techniques for dynamic probe placement (e.g., to enable/disable probes at run-time). Our current oracle is determined at design-time, but machine learning-based approaches could provide designs that perform better in adaptive systems. Moreover, as we observed in the experiment, SMFL window size is an important parameter that can affect the accuracy of the diagnosis. We plan to study a systematic, generic method to automatically determine this parameter. We believe that this can be done in a parametric way, based on the family of system and the kind of implementation base on which it is deployed. Furthermore, our architecture-based fault localization approach allows the definition of multiple spectrum types. It is not yet clear whether each type should have its own SMFL diagnosis instance, or whether they should be combined into a single detection component. A key issue will be to determine whether there is a need for multiple SMFL windows depending on spectrum type, as this will require the use of multiple SMFL instances. While the approach scales well in terms of its algorithmic complexity, we plan to conduct experiments on large-scale systems to evaluate the scalability of the method in practice. Finally, we plan to integrate the diagnosis mechanism into detection-diagnosis-repair cycle, to determine how it impacts round-trip self-repair efficiency.

Acknowledgements. This research was supported by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and W911NF-09-1-0273 from the Army Research Office, and by a grant from the Foundation for Science and Technology via project CMU-PT/ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN (FCOMP-01-0124-FEDER-012983).

References

1. Abreu, R., van Gemund, A.J.C.: Diagnosing multiple intermittent failures using maximum likelihood estimation. Artif. Intell. 174(18), 1481–1497 (2010)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proc. of TAIC PART 2007. IEEE Computer Society, Los Alamitos (2007)
3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: An observation-based model for fault localization. In: Proc. of WODA 2008. ACM Press, New York (2008)

4. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: Taentzer, G., Heimdahl, M. (eds.) Proc. of ASE 2009. IEEE Computer Society, Los Alamitos (2009)
5. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot: A technique for cheap recovery. In: Proc. OSDI 2004, San Francisco, CA (2004)
6. Cheng, B.H.C., de Lemos, R., Garlan, D., Giese, H., Litoiu, M., Magee, J., Müller, H.A., Pezzè, M., Taylor, R. (eds.): Proc. of SEAMS 2010. ACM Press, New York (2010)
7. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proc. of SEAMS 2006, May 21-22 (2006)
8. Cutting, D.: The hadoop framework (2010)
9. de Kleer, J.: Diagnosing intermittent faults. In: Biswas, G., Koutsoukos, X., Abdelwahed, S. (eds.) Proceedings of the 18th International Workshop on Principles of Diagnosis (DX 2007), Nashville, Tennessee, USA, May 29-31, pp. 45–51 (2007)
10. de Kleer, J.: Diagnosing multiple persistent and intermittent faults. In: Proc. of IJCAI 2009. AAAI Press, Menlo Park (2009)
11. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence 32(1), 97–130 (1987)
12. Dobson, S.A., Strassner, J., Parashar, M., Shehory, O. (eds.): Proc. of ICAC 2009. ACM Press, New York (2009)
13. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
14. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self adaptation with reusable infrastructure. IEEE Computer 37(10) (October 2004)
15. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of the Symposium on Operating Systems Principles. ACM, New York (2003)
16. Ghosh, D., Sharman, R., Raghav Rao, H., Upadhyaya, S.: Self-healing systems - survey and synthesis. Decis. Support Syst. 42, 2164–2185 (2007)
17. Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification and Reliability 10(3), 171–194 (2000)
18. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization of test information to assist fault localization. In: Proc. of ICSE 2002. ACM Press, New York (2002)
19. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1) (2003)
20. Kiviluoma, K., Koskinen, J., Mikkonen, T.: Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In: Proc. of ISSTA 2006. ACM Press, New York (2006)
21. Kolettis, N., Fulton, N.D.: Software rejuvenation: Analysis, module and applications. In: Proc. of FTCS 1995. IEEE Computer Society, Washington, DC, USA (1995)
22. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters 29, 155–163 (1988)
23. Kramer, J., Magee, J.: A rigorous architectural approach to adaptive software engineering. J. Comput. Sci. Technol. 24, 183–188 (2009)
24. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proc. of PLDI 2005, Chicago, Illinois, USA (2005)
25. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P.: Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering (TSE) 32(10), 831–848 (2006)
26. Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: Proc. of ASE 2008 (2008)

27. Mikic-Rakic, M., Mehta, N., Medvidovic, N.: Architectural style requirements for self-healing systems. In: Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, pp. 49–54. ACM, New York (2002)
28. Palviainen, M., Evesti, A., Ovaska, E.: The reliability estimation, prediction and measuring of component-based software. Journal of Systems and Software 84(6), 1054–1070 (2011)
29. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering Architectures from Running Systems. IEEE Transactions on Software Engineering 32(7), 454–466 (2006)
30. Trivedi, K.S., Vaidyanathan, K.: Software aging and rejuvenation. In: Wiley Encyclopedia of Computer Science and Engineering. John Wiley & Sons, Inc., Chichester (2008)

A Self-adaptive Monitoring Framework for Component-Based Software Systems

Jens Ehlers and Wilhelm Hasselbring

Software Engineering Group
Christian-Albrechts-University Kiel
24098 Kiel, Germany
{jeh,wha}@informatik.uni-kiel.de

Abstract. To allow architectural self-adaptation at runtime, software systems require continuous monitoring capabilities to observe and to reflect on their innate runtime behavior. For software systems in productive operation, the monitoring overhead has to be kept deliberately small. By consequence, a trade-off between the monitoring coverage and the resulting effort for data collection and analysis is necessary. In this paper, we present a framework that allows for autonomic on-demand adaptation of the monitoring coverage at runtime. We employ our self-adaptive monitoring approach to investigate performance anomalies in component-based software systems. The approach is based on goal-oriented monitoring rules specified with the OCL. The continuous evaluation of the monitoring rules enables to zoom into the internal realization of a component, if it behaves anomalous. Our tool support is based on the Eclipse Modeling Project and the Kieker monitoring framework. We provide evaluations of the monitoring overhead and the anomaly rating procedure using the JPetStore reference application as a Java EE-based test system.

Keywords: Adaptive monitoring, failure diagnosis, anomaly detection.

1 Introduction

Performance is a critical characteristic for software systems. Even though monitoring the operation of systems is often neglected in practice. A recent survey among Java practitioners and experts [10] indicates this antagonism: Adequate application-level monitoring tools that allow to analyze the causes of performance problems are seldom known and employed in software engineering projects.

It is difficult to decide in advance where to place the monitoring probes and which data should be collected. Thus, probes are typically instrumented only in reaction to prior performance degradations or system failures. In contrast to construction-time profiling, continuous monitoring at operation time has to regard a deliberately small monitoring overhead. Consequently, a main issue is the limited amount of information that can be collected and processed. More detailed monitoring data allows for more detailed analyses of the underlying software system's behavior. We evaluated and quantified the impact of how monitoring data is collected, processed, and persisted for subsequent analyses. A finding is that it is feasible to instrument probes at a variety of

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 278–286, 2011.

possibly relevant measuring points, as long as not all of them are active at the same time during operation.

In this paper, we present a self-adaptive, rule-based monitoring approach that allows on-demand changes of the monitoring coverage at runtime. The monitoring rules follow the goals for which monitoring data is required, e.g. the evidence of SLA compliance, dynamic adaptation of resource capacities, or usage pattern recognition for interface design. We will concentrate on the monitoring goal to localize performance anomalies that change the valid behavior of software system as perceived by its users. For the specification of the monitoring rules, we employ the Object Constraint Language (OCL) [8]. The rules refer to performance attributes of a previously extracted system runtime model. As the model values (particularly anomaly scores rating the timing behavior of system-inherent operations) change during operation, a continuous evaluation of the monitoring rules is required. Our implementation is based on EMF (Eclipse Modeling Framework)[1] meta-models which allow for evaluation of OCL query expressions on object-oriented instance models at runtime.

The remainder of this paper is structured as follows: In Section 2, we describe our approach for self-adaptive performance monitoring and the underlying anomaly rating procedure. Its evaluation in lab experiments and industrial systems is summarized in Section 3. Related work is discussed in Section 4. Finally, a conclusion and an outlook to future work are given in Section 5.

2 Self-adaptivity for Continuous Software System Monitoring

In this section, we present our approach for self-adaptive software system monitoring, which is embedded into our Kieker monitoring framework[2] [5]. Kieker facilitates to monitor and to analyze the runtime behavior of component-based software systems. The underlying monitoring and analysis process is structured into the following activities: probe injection, probe activation, data collection, data provision, data processing, visualization, and (self-)adaptation. Different plugins can be integrated into this analysis process via the pipes-and-filters pattern. In the following, we present a Monitoring Adaptation Plugin addressing rule-based adaptation of the current monitoring coverage.

To enable fine-grained monitoring of component-internal behavior, probes have to be instrumented at various measuring points in the components' control flow. At operation time with extensive system workload, it is not possible to process each probe actuation. Only a selection of the measuring points can be activated. An adequate initial coverage is to activate the measuring points that intercept the execution of system interface operations. Our proposed Monitoring Adaptation Plugin allows the specification of monitoring rules which are evaluated continuously and may effect changes of the current monitoring coverage. We will specify a rule that the coverage of a component's interior control flow should be increased if it does not behave as expected. In this way, our approach affords automatic on-demand adaptation of the effective software system monitoring.

[1] `http://www.eclipse.org/modeling/emf/`

[2] `http://kieker.sourceforge.net/`

If the monitoring adaptation is conducted manually, the human decision to change the set of active measuring points is usually caused by (critical) incidents that imply anomalous runtime behavior. A performance engineer who observes such an incident is interested in the root cause and tries to activate more measuring points in the affected components. Subsequently, it takes a while until enough relevant records have been collected via the newly activated measuring points. It is well-known that a major part of the failure recovery time is required to locate the root cause of a failure. An estimation of 75% of the recovery time being spent just for fault localization is referred to in [6]. Our self-adaptive monitoring approach will reduce this potentially business-critical wait time that delays a failure or anomaly diagnosis.

2.1 Runtime Evaluation of OCL-Based Monitoring Rules

We employ the OCL to specify the monitoring rule premises. OCL is well known for its purposes to specify invariants on classes, pre- and postconditions on operations, or guards in UML diagrams. Nevertheless, the first objective listed in the OCL specification suggests OCL to be a query language [8]. In our case, the monitoring rule premises can actually be regarded as queries that select a set of measuring points to be activated or deactivated. The Monitoring Adaptation Plugin provides an editor with syntax highlighting and code completion for performance analysts to specify the required OCL expressions. The context in which an OCL expression will be evaluated is determined by a selectable context element. In the expression, the context element can be referenced by the OCL identifier self. Appropriate context elements are the analysis models of other Kieker plugins which provide the input of the Monitoring Adaptation Plugin. As all plugins are based on EMF meta-models, we are able to utilize the EMF Model Query sub-project, which allows constructing and running queries on EMF models by means of OCL. Goal-oriented self-adaptation is based on the possibility to refer to attributes in the OCL expressions that change their values during runtime, e.g. responsiveness metrics and derived anomaly scores.

P_1: **context** CallingContextTree:
self.callingContexts→*select*(level = 1 **or**
(parent.op.monitored **and** parent.anomalyScore > *t*))
→*collect*(op)

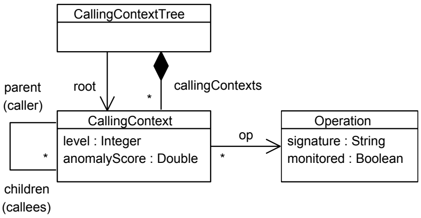


Fig. 1. Monitoring premise and corresponding simplified CCT meta-model

The Monitoring Adaptation Plugin runs a thread that evaluates the specified monitoring rules. The time interval between succeeding evaluations of the monitoring rules has to be configured manually. Enough time is required to collect reliable new monitoring data in each iteration. On the other side, the delay time must be short enough to react promptly to observed anomalies. An interval value in the scale of a couple of minutes is appropriate. Probe (de)activation instructions are delegated to the monitoring agent instances via their remote adaptation interface.

In the following, we discuss an example monitoring rule: “If an operation is selected by the rule premise P_1 as described in Figure 1, then activate the probe measuring point required to intercept and monitor calls to this operation.” The context element of P_1 is a calling context tree (CCT) model. The simplified meta-model of a CCT is depicted in Figure 1. P_1 selects all operations that are called from a caller operation that is already monitored (parent.op.monitored) and behaves anomalous in a particular calling context (with an unique call stack), i.e. the context's anomaly score exceeds a specified threshold t (parent.anomalyScore $> t$). Additionally, all operations are added to the result set that are at the topmost level of the CCT (level = 1), i.e. system-level interface operations for incoming client requests.

2.2 Software Performance Anomaly Rating

The above monitoring rule references an operation-level anomaly score metric. As it strongly depends on the context if an observation has to be considered as anomalous or not [2], we capture and separate different contexts as far as possible (e.g. separation by calling context), but we assume that it is not possible to separate all context-determinant impact factors (e.g. operation input parametrization, component state, system workload). Even from a fine-grained contextual viewpoint, response times can be arbitrarily distributed and do not necessarily converge to a parametric distribution model. Thus, we suggest an anomaly rating procedure based on time series analysis that disregards any technical or economical influences. The characteristic features of the underlying stochastic process are recovered from the present time series of response times. Our anomaly rating procedure consists of four steps, which are summarized in the following and explicated in detail in [3]:

(1) Forecast expected response times for each software service in dependence of the stack context based on historical observations. We provide different forecast models such as single exponential smoothing (SES), Holt-Winters smoothing, and ARIMA models.

(2) Test if a sample of newly measured service response times is to be rated as normal or anomalous related to the expected forecast value from (1). A Student's t-test is conducted based on the measured sample variance.

(3) Based on the sequent rating of response times samples from (2), calculate an anomaly score expressing the recent degree of a software service to exhibit anomalous timing behavior. Here, we construct an anomaly scoring function that reflects the frequency and the trend of anomalous samples over time.

(4) Aggregate and correlate anomaly scores from (3) to higher levels of abstraction, e.g. component-level anomaly scores.

3 Evaluation

We employed the Kieker Monitoring component in the productive systems of a telecommunication company and a digital photo service provider [5]. These previous case studies confirmed the practicability and the robustness of our approach. Regarding the monitoring cost, our industrial partners were not able to perceive any monitoring overhead due to the instrumentation of our probes. Thus, we set up lab experiments to quantify the monitoring overhead and to evaluate the self-adaptive anomaly detection. The results of these evaluations are recapped in the following.

Monitoring Overhead: The goal of our monitoring cost evaluation is to quantify and to decompose the monitoring overhead. In Figure 2, we apportion monitoring costs for instrumentation (Δ_I), data collection (Δ_C), and data logging (Δ_L). In the experiment, we monitored an operation that takes 500 μs to be processed on a specific test system (Sun Blade X6270 with 2x Intel Xeon E5540, total 8 cores at 2.53 GHz, 24 GB RAM, ZFS RAID, SunOS 5.1, Java HotSpot x86 Server VM 1.6). The response time deviation is minimal as we carried out an extensive warm-up phase to saturate the JVM behavior, particularly the just-in-time compilation. In the experiment, the monitored operation was continuously executed by 15 concurrent threads. The boxplots show that (1) instrumentation, i.e. processing previously woven, but inactive dummy probes, causes negligible overhead (Δ_I is less than 1 μs) compared to (2) data collection and (3) logging, i.e. creating and persisting the monitoring records to a Monitoring Log (Δ_C and Δ_L are each about 4 μs). In case (3) of the experiment where logging is enabled, the records were written asynchronously into the local file system by a dedicated writer thread. This avoids a direct delay of the response time perceived by the system users. The remaining logging overhead is effected by the thread concurrency. Our evaluation results suggest the conclusion that injecting probes at a multitude of measuring points is not critical as long as data collection and logging can be (de)activated systemically. This finding underpins the adaptive activation of measuring points proposed in Section 2.1.

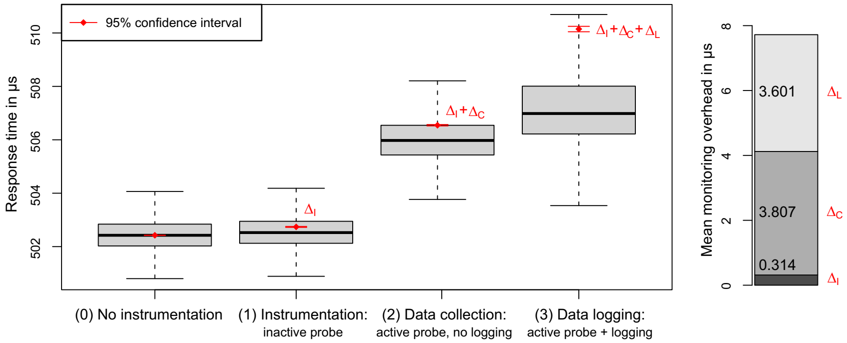
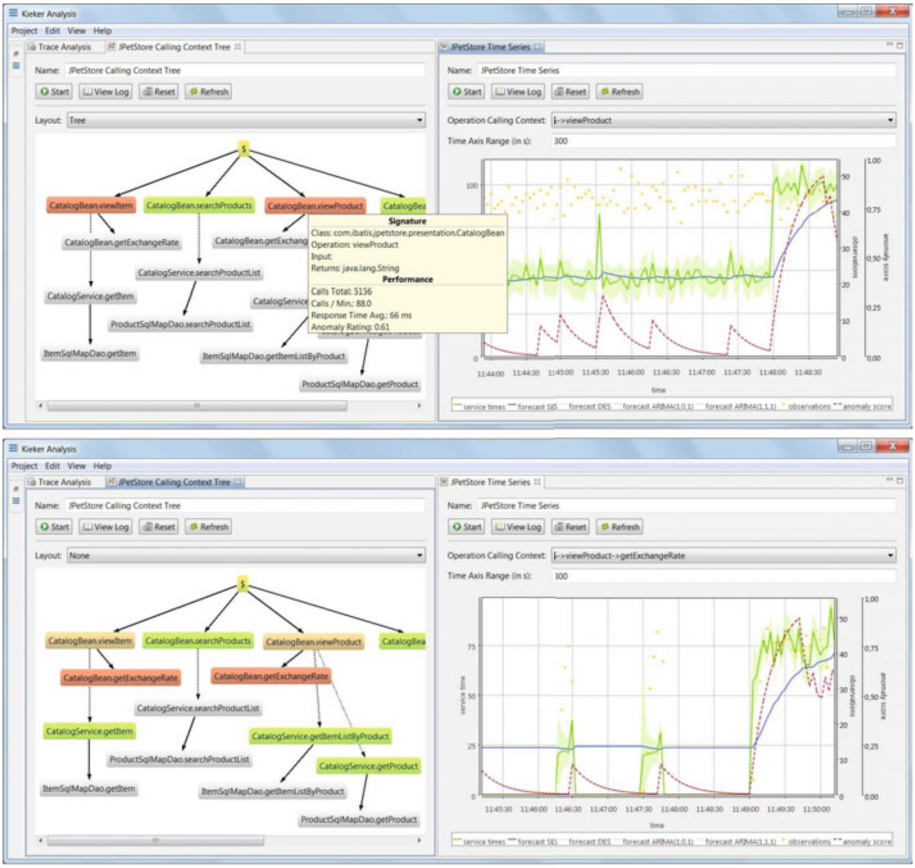


Fig. 2. Evaluation of the monitoring cost

Anomaly Detection: In [3], we presented our evaluation results for different forecast models. Here, we show that consecutive divergences of measurements and forecasts indicate anomalous timing behavior. For our evaluation, we use the JPetStore[3] reference application as a test system. Initially, we monitor only the system's interface operations.

[3] http://sourceforge.net/projects/ibatisjpetstore/



Kieker Analysis

Name: JPetStore Calling Context Tree

Layout: Tree

Signature
Class: com.ibatis.jpetstore.presentation.CatalogBean
Operation: viewProduct
Input:
Returns: java.lang.String

Performance
Calls Total: 5156
Calls / Min.: 88.0
Response Time Avg.: 66 ms
Anomaly Rating: 0.61

Name: JPetStore Time Series

Operation Calling Context: $->viewProduct

Time Axis Range (in s): 300

Kieker Analysis

Name: JPetStore Calling Context Tree

Layout: None

Name: JPetStore Time Series

Operation Calling Context: $->viewProduct->getExchangeRate

Time Axis Range (in s): 300

Fig. 3. Kieker screenshots with calling context tree (left) and responsiveness time series (right)

In our experiment, we stress the system under test with workload that causes a desirable level of resource utilization, i.e. the CPUs are continuously utilized in a range between 30% and 50%. Given this load, the system was run and observed for 2 days, using SES for forecasting with a fixed smoothing factor of 0.15. The anomaly scores of the monitored services did not exceed a threshold value of 0.5. That is, there have never been 7 of 10 subsequent samples that were rated anomalous. For example, the mean response time of the viewProduct service was 48.7 ms with a standard deviation of 10.3 ms. A response time curve of this operation is depicted in the top right view part of Figure 3. The green line indicates the observed mean response time surrounded by a light green confidence interval whose range depends on the observed variance and the specified significance level. The blue line indicates the expected response time determined by the used forecast model. During failure-free operation, the forecast value is mostly within the confidence interval. By consequence, the anomaly score indicated by the red line does not rise considerably. In the very right part of the depicted time series, a fault is injected that disrupts the failure-free operation. Suddenly, response times nearly double. Without this pattern being expected in advance, the forecast model

adapts itself only moderately to the new timing behavior. In this period, measurement and forecast diverge so that the anomaly score increases rapidly and exceeds the defined alarm threshold.
This situation is captured in the top left view part of Figure 3, where an extract of the system's calling context tree is shown. The color of the operation nodes from green to red indicate their current anomaly score. Operations that are hitherto not monitored are colored in light gray. As a consequence of the anomalous behavior of the viewProduct and viewItem operations, the evaluation and appliance of the monitoring rule P_1 explicated above leads to the monitoring activation for their callees. As shown in the bottom left view part of Figure 3, the monitoring coverage is adapted to localize the root cause of the anomalous behavior. In the scenario depicted in the screenshots, both anomalous interface operations depend on a common operation called getExchangeRate effecting the anomaly. The time series in the bottom right view part of Figure 3 demonstrates that the getExchangeRate operation has not been monitored continuously until the anomaly occurred. Only sparsely distributed sample observations have been made to check over the learned expected behavior. In the discussed evaluation scenario, we systematically changed the responsiveness of the external web service which is invoked from inside of the getExchangeRate operation. Though we injected this fault on purpose, a similar incident can easily occur in a productive system that depends on third-party services.

Furthermore, we studied two more fault injection scenarios, which are not described in detail due to space restrictions: In the first scenario, we dropped and recreated a database index causing significant changes in the response time of several operations called from lower levels of the system's CCT. We kept track of how our self-adaptive monitoring approach successively adapts the monitoring coverage to zoom in and out the CCT. In a second scenario, we increased the overall system load abruptly to simulate a situation where a load-balanced system replica drops out and the remaining replicas have to absorb the capacity reduction. As in our experiment setup particularly the CPU resources have not been heavily underutilized, almost all expensive operations react anomalous. It is obvious that a manual exploration of such cause-and-effect chains as constructed in our experiments is much more time-consuming and error-prone than an automated processing. A major contribution of the self-adaptive monitoring approach is to save this time and effort.

4 Related Work

An integrated software system monitoring framework such as Kieker is concerned with two aspects: (1) monitoring, i.e. instrumentation and data acquisition, and (2) subsequent analysis. Related work comprises the COMPAS JEEM project [9] which facilitates the injection of probes as a component-level proxy layer in Java EE systems. In the context of COMPAS, adaptation of the monitoring coverage at runtime is studied in [7]. However, monitoring is restrained to the interface level of Java EE components such as EJBs or Servlets. By the observation of component-internal operation responsiveness, Kieker allows a finer-grained insight.

Concerning application-level fault determination without addressing self-adaptation, related work is provided by the Pinpoint approach [6]. In contrast to Kieker, Pinpoint does not focus on performance time series, but applies pattern-oriented data mining techniques to detect anomalies in the request traces. A further related approach addressing monitoring of resource utilization and component interactions in distributed systems is Magpie [1]. While the implementation of Kieker concentrates on Java-based systems, Magpie is realized to monitor systems based on Microsoft technology. The Rainbow project [4] employs monitoring for architecture-based adaptation of software systems. To our knowledge, Magpie, Pinpoint, and Rainbow so far do not contribute means for rule-based self-adaptation to control the monitoring coverage. The same applies to related commercial products like CA Wily Introscope, DynaTrace, or JXInsight. The popular open-source tool Nagios is intended for infrastructure monitoring, not for application-level monitoring.

5 Conclusions and Future Work

Responsiveness and scalability of a software system components have to be monitored and analyzed continuously. In case a system component responds anomalous, our self-adaptive monitoring approach enables zooming into the component-internal behavior on demand. Zooming means to activate more (or less) measuring points in the application-level control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. A set of OCL-based monitoring rules is proposed to control the monitoring coverage automatically. In Section 2, we presented an extension to our Kieker monitoring framework supporting self-adaptive software system monitoring based on the continuous evaluation of OCL-based monitoring rules at runtime. Further, we briefly described our underlying anomaly rating procedure for the timing behavior of software systems. In Section 3, we quantified the monitoring overhead and evaluated the anomaly detection procedure in lab experiments.

In our future work, we plan to study the adaptive monitoring approach in case studies with industrial partners. Further, we intend to implement unsettled practical issues concerning our Kieker monitoring framework such as model-driven instrumentation, IDE integration, and support for other programming languages in addition to Java.

References

1. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation, pp. 259–272. USENIX (2004)
2. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Computing Surveys 41(3), 1–58 (2009)
3. Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proc. of the 8th IEEE/ACM Intl. Conf. on Autonomic Computing (ICAC 2011), pp. 197–200. ACM, New York (2011)

4. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
5. van Hoorn, A., Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., Kieselhorst, D.: Continuous monitoring of software services: Design and application of the Kieker framework. Tech. Rep. TR-0921, Dept. of Computer Science, University of Kiel (2009)
6. Kiciman, E., Fox, A.: Detecting application-level failures in component-based internet services. IEEE Trans. on Neural Networks 16(5), 1027–1041 (2005)
7. Mos, A., Murphy, J.: COMPAS: Adaptive performance monitoring of component-based systems. In: 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems, 26th Intl. Conf. on Software Engineering (ICSE 2004), pp. 35–40 (2004)
8. OMG: Object Constraint Language, Version 2.2. http://www.omg.org/spec/OCL/2.2/ (2010)
9. Parsons, T., Mos, A., Murphy, J.: Non-intrusive end-to-end runtime path tracing for J2EE systems. IEE Proc. – Software 153(4), 149–161 (2006)
10. Snatzke, R.G.: Performance survey 2008 – survey by codecentric GmbH (2009), http://www.codecentric.de/de/m/kompetenzen/publikationen/studien/

Towards Real-Time Monitoring and Controlling of Enterprise Architectures Using Business Software Control Centers

Tobias Brückmann, Volker Gruhn, and Max Pfeiffer

paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, 45127 Essen, Germany
{tobias.brueckmann,volker.gruhn,
max.pfeiffer}@paluno.uni-due.de

Abstract. Enterprise Architecture Management (EAM) plays an important supporting role in IT management of organizations to align their IT infrastructure to actual business needs. This emerging research paper presents an approach to enable real-time monitoring and controlling of enterprise architectures. Therefore, we adapted the "control center" concept as applied in power plants or railway control plants. The contribution of this paper presents an architecture for real-time monitoring and controlling facilities for complex business application landscapes. The business software control center is designed to give a real-time view of instances of IT-supported business processes together with the currently involved software systems and services. Moreover, IT operators are supported by controlling centers to actively control the load of software services at the business function level and to control the flow of business process instances through the organization's IT infrastructure.

Keywords: Enterprise Architecture Management, Real Time, Software Control Center.

1 Introduction

Evolving technologies and continuously changing business processes are an enormous challenge for IT managers. By supporting the IT management to align the IT infrastructure to actual business needs, several models and processes were developed in so-called Enterprise Architecture Management (EAM) [3, 8, and 10] frameworks. In this context, EAM is a "management function" [10] and Enterprise Architectures (EA) are architectural models that contain information about the organization's business processes as well as information about the organization's IT infrastructure. Enterprise Architecture models target connections and dependencies of the business and IT architecture. They should reflect the degree of the actual alignment of the IT to business needs. However, creating and maintaining complex EAs of large organizations is a longsome and extensive task, whose results are often outdated due to the continuous changing nature of business processes and IT application landscapes.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 287–294, 2011.

For this reason, our paper introduces an approach for real-time monitoring and controlling of Enterprise Architectures. Inspired by control rooms in power plants or railway control plants, which provide an integrated and always up-to-date view of the involved infrastructure and the processed material, we propose the use of so-called business software control centers to provide an integrated and always up-to-date view of software services and applications together with the state of instances of executed business processes. Although metering of CPU load, network bandwidth and memory usage is commonly used to monitor and report operating figures at the hardware and operating system level, a link between business functions and software services or application calls has not been considered yet. Furthermore, the IT operators are not supported yet to actively control the load of software services at business function level or to actively manage the flow of process instances through the organization's IT infrastructure. Our approach is based on connecting real-time models of different views.

The remainder of this paper is structured as follows: Section 2 gives a brief overview of related work. Likewise, we discuss foundations of our approach before we state the research challenges derived from our vision of business software control centers under consideration of the existing and related work in Section 3. In Section 4, we introduce our approach at the architecture level together with an illustrating scenario. Finally, in Section 5, we conclude the paper and sketch our future work.

2 Related Work

As related work, we considered EAM frameworks, which integrate business and IT views into organizations, existing software systems monitoring approaches, and business process monitoring approaches.

EAM frameworks such as Zachman-framework [10], DoDAF [3], and TOGAF [8] describe the IT infrastructure of an organization and relate it to the implemented business processes. Zachman [10] first came up with the idea to structure complex information systems in logical architectures. His framework helps to classify and model information systems and their relation to the business model and business processes in different views and level of abstraction. The main goal of the Zachman framework is to provide an inventory model of all components of the IT infrastructure (incl. hardware, networks, storage and applications). The DoD Architecture Framework (DoDAF) [3] was designed to support architecture decisions for stable and interoperable systems as part of IT landscapes. It also defines different views (operational view, systems and service view, and technical standards view) to inspect the internal architecture of an organization on different perspectives and scopes.

The main goal of TOGAF [8] is an optimal alignment of business and information technology. The methods and techniques of the framework support Enterprise Architecture Management with a defined process comprising eight main phases. TOGAF surveys the current architecture of the organization and transfers it to new developed target architecture for applying sophisticated IT governance.

The EAM frameworks described above aim to provide processes and models to support the alignment of the business architecture and the IT architecture of organizations. Hence, we developed our work on top of these frameworks and developed an approach for real-time monitoring and controlling of EAs.

Real-time system monitoring is a commonly used technique in industry. There are commercial solutions for monitoring hardware loads such as networks, memory, and CPU, as well as for monitoring software components such as versions and configurations. IBM Tivoli [5] is optimizing performance and availability of IT infrastructures. It can be applied to managing operating systems, databases and servers. With Nagios [6] infrastructures like systems, applications and services can be monitored, and infrastructure problems can be identified. Zenoss [11] is a monitoring and event management tool for servers, networks, and virtualized infrastructures. These tools and tool sets support the planning and organization of large IT landscapes. They give a real-time overview of the current state and help to respond quickly to changes and failures. However, the focus of system monitoring solutions is a pure IT view of an organization. They do not include the business models and processes. Hence, they do not support an integrated real-time view of business functions and software functions.

In the context of Business Intelligence [4], different architectures are already developed for business process monitoring. These architectures provide reports, dashboards and alerts reflecting only information form the IT infrastructure level to the business function level.

In addition to system monitoring tools, business process monitoring tools are used for real-time monitoring of the internal business activities of organizations. As a difference to system monitoring approaches, business process modeling only focuses on executed instances of defined business processes. As commercial solutions, Oracle BAM [7] and NetWeaver SAP [8] support real-time process control and workforce management. Oracle BAM [7] monitors business services and processes from an enterprise down to the actual business process level. It uses a so-called dashboard to display the real-time data, which can be used by process managers to identify needs for modifications of the implemented business processes. SAP NetWeaver [8] is a tool for creating large business process models on different abstraction levels. Comparable to the system monitoring approaches, business process monitoring solutions focus only on the monitoring of current business activities and do not take the underlying IT infrastructure into account.

Summing up the existing approaches in the field of Enterprise Architecture Management and real-time monitoring and controlling, there are established processes and models to create EA models. Moreover, there are tools supporting system monitoring from only a technical viewpoint, and there are tools supporting business process monitoring from only a business viewpoint. However, an integrated real-time view for monitoring and controlling of both is not supported yet.

3 Problem Statement

As shown above in Section 2, a lot of modeling concepts and views to relate business functions and IT functions are available. However, they are only used at design-time and for hand-made models. To enable real-time monitoring and controlling of Enterprise

Architectures, we have to establish an automated link between implemented business processes and implemented software applications and services. Therefore, we need

- A conceptual model that provides relations between implemented business functions and software system functions as a foundation of the visualization of the real-time EA.
- A real-time visualization of instances of actual processed business functions and involved software systems and services.
- A bi-directional communication infrastructure between software systems and the control center.
- An automated update mechanism, which recognizes changes of business functions and changes in application landscape and displays these changes.
- An infrastructure that is able to provide a real-time visualization of executed business processes, operated software systems and their relations.
- A process model for development of applications and services that can be monitored and controlled using a business software control center.

4 Business Software Control Center

The underlying vision of a business software control center is to provide a real-time view of the current business process instances together with the actual involved software systems and services. Therefore, the real-time states of all connected applications and services need to be captured and communicated to the business software control center. To present an integrated view of both – business functions and IT systems – the control center contains a conceptual and visual model of the EA. This model provides a foundation for the control center to map the incoming information from the IT systems to the graphical user interface. The GUI of the control center contains a set of different views in different levels of detail. The user can for example select whether only IT systems are displayed, or an integrated view of business functions and IT functions should be presented. Moreover, if the business software control center supports active controlling functions, additional services can be added or the whole IT support of business processes can be rearranged. The reminder of this section introduces an architectural description of such a business software control center as an emerging result of our work.

Figure 1 shows a schematic GUI sketch of the business software control center based on the ideas of traditional control centers. The business process and IT infrastructure are visualized in real-time. The connection between both views shows which process step is supported by which software functionality. Depending on the use of the business software control center, necessary information can be added and irrelevant information can be hidden. The business process shows a view of a simplified billing process in BPMN [9] which is extended by process tokens. These process tokens represent the actual instance (i.e. user) during this process in real-time. The connection shows which process step is executed on which software component of the IT infrastructure (here, for example, the billing step A is related to a specific component of the banking server).

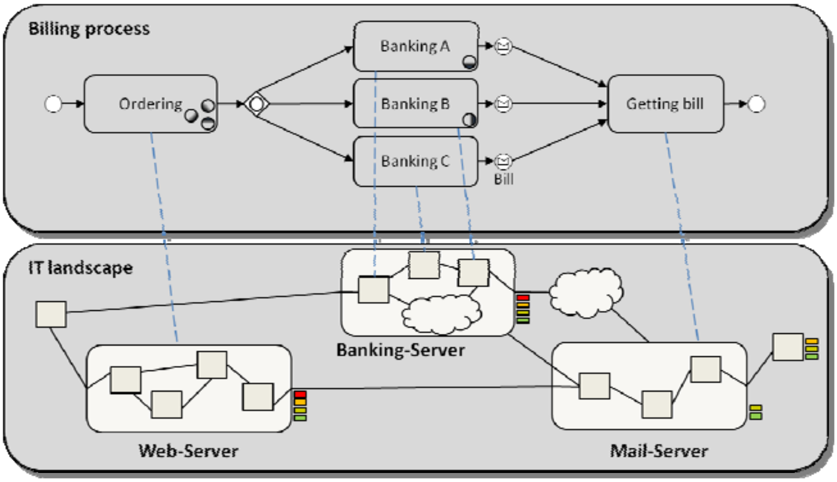


Fig. 1. Conceptual Sketch of the control center GUI

The technical infrastructure that is required to operate a business software control center comprises the control center unit, the actual Enterprise Architecture (EA) model, the monitored and controlled IT infrastructure, and a control center bus. An overview is given in Figure 2. The control center unit is a set of connected components as described in detail in Section 4.1. The visualization and the internal model of the control center unit depend on the underlying EA model, which specifies the enterprise as seen by the control center (see Section 4.2). The monitored applications and services (IT infrastructure) are connected to the business software control center via a control center bus (see Section 4.3).

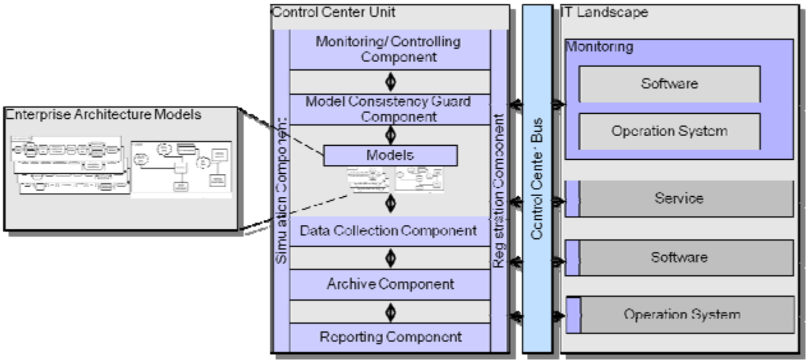


Fig. 2. Architecture Overview of the Business Software Control Center

4.1 Control Center Unit

The control center unit is the central unit and contains several components to display, update and store the relevant information as described in this subsection.

Monitoring Component. The monitoring module is an interactive monitoring panel for observing the real-time states of the business processes and the IT infrastructure. It provides a customizable GUI and supports different views in different levels of detail. Compared to Figure 1, the business process view and the IT landscape view are shown together. For example, if the banking service A fails in the IT landscape, it will change both views and a notification will occur in the GUI in real-time.

Controlling Component. The controlling component is an interactive control panel based upon the monitoring component. The controlling component extends the control center with monitoring capabilities for control functions. Related to Figure 1, if the business contract of the banking service A is canceled in the business process view, it will be disabled in the software component as well.

Model Consistency Guard Component. This component evaluates continuously if the modeled EA fits to the actual registered and monitored software systems and services. Moreover, this component is responsible for detecting inconsistencies of the models with the real world. If the inconsistency, for example an unregistered banking service D, was not triggered by the control center, a GUI notification will occur.

Data Collection Component. The data collector component is responsible for a continuously updated real-time model, which provides the basis for the visualization in the monitoring component. Therefore, it processes the incoming information from the control center bus and updates the run-time model. For example, the billing process (Figure 1) passes into the next step after ordering when the banking server is entered by a user and sends a request.

Registration Component. The registration component is a register that connects the information of the EA model with the existing IT infrastructure. This component is responsible for establishing and maintaining the logical connection of the business software control center and the monitored software systems. If a new banking service D is added to the banking server, it must be registered with the main control unit. Afterwards, the process model will be updated in real-time.

Archive Component. The archive component stores the complete history of all incoming and processed information together with the EA model. The stored information is needed for statistic evaluations and report generation (see Report Component) as well as for simulation purposes (see Simulation Component).

Reporting Component. The reporting component generates reports based on the data stored in the archive component. Moreover, the reporting component is also used to run statistical analysis. Results of such an analysis can be used to create load forecasts and to support IT management decisions – for example, which banking service is used more often than others, or in which step most users leave the process.

Simulation Component. The simulation component of the control center uses the EA model together with collected data of the archive component to enable simulation runs of the business processes and software systems. With the help of simulation, the IT department can do runtime experiments without affecting real business process instances. For example, a failure of the banking service A (Figure 1) during a peak load can be simulated.

4.2 Enterprise Architecture Models

In the context of our proposed business software control center, two types of models play an important role: The EA model, which is a design-time model and which contains a structural view onto defined business processes, existing software systems, and their relations. The EA model has defined visual representations, which are used by the monitoring component of the control center. In contrast to business process models (such as BPMN [8]) and system models (such as UML [1]), the EA model needs to integrate both points of view into one consistent EA model.

In addition to the design-time EA model, the control center uses an internal run-time model. The run-time model is based on the design-time EA model. It carries information about concrete instances of business function calls and software system function calls.

4.3 Control Center Bus

Besides the control center unit, a proper communication infrastructure between the control center and controlled software systems is needed. The communication bus has to support a heterogeneous infrastructure, because each deployed system or service should be able to communicate with the control center. Moreover, we expect a lot of sensitive messages have to be handled by the communication infrastructure. Hence, it has to be assured that no messages gets lost, that security requirements can be met, and real-time critical message are delivered as fast as they are needed. Additionally, in case of a communication failure, the failure has to be detected and treated automatically without affecting the stability of the control center.

5 Conclusion

In this emerging research article, we presented an approach to enable real-time monitoring and controlling of Enterprise Architectures. On the foundation of existing EA models, we adopted the “control room” concept (as applied e.g. in power plants), and provided an architectural description of business software control centers. With such control centers, we aim to support real-time monitoring and control of facilities for complex business application landscapes. We proposed a technical infrastructure that comprises the control center unit, the actual EA model, the monitored and controlled software applications and services, and a control center bus. The heart of this architecture is the so-called control center unit, which contains a set of connected components to display, update and store the relevant information as delivered from software systems. Reporting and simulation functions are also considered.

In the next steps towards real-time monitoring and controlling of enterprise architectures, we plan to develop a detailed and usable visualization concept that integrates business functions and software system functions and implements a research prototype of the proposed business software control center. We plan to run the first experiments with the prototype in a simulated industrial environment together with an industrial partner.

References

[1] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading (1998)

[2] Business Process Monitoring with SAP, http://www.sdn.sap.com/irj/sdn/nw-process-monitoring (last access April 02, 2011)

[3] Department of Defense (DoD). DoD Architecture Framework Version 1.0: Volume I: Definitions and Guidelines (2004)

[4] Golfarelli, M., Rizzi, S., Cella, I.: Beyond data warehousing: what's next in business intelligence? In: Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP. ACM, New York (2004)

[5] IBM Tivoli Monitoring, http://www.ibm.com/software/tivoli/products/monitor/ (last access March 23, 2011)

[6] Nagios, http://www.nagios.org/ (last access March 23, 2011)

[7] Oracle Business Activity Monitoring (Oracle BAM), http://www.oracle.com/technetwork/middleware/bam/overview/index.html (last access April 02, 2011)

[8] The Open Group. TOGAF "Enterprise Edition" Version 8.1 (2005)

[9] White, S., Miers, D.: BPMN Modeling and Reference Guide. Future Strategies, Lighthouse Point (2008)

[10] Zachman, J.: A framework for information systems architecture. IBM Systems Journal 26, 276–292 (1987)

[11] Zenoss - Making the Cloud Work with Monitoring, Analytics & Insight, http://www.zenoss.com/ (last access March 23, 2011)

Towards a Model-Based Approach for Reconfigurable DRE Systems

Fatma Krichen[1,2], Brahim Hamid[1], Bechir Zalila[2], and Mohamed Jmaiel[2]

[1] IRIT, University of Toulouse, France
{krichen,hamid}@irit.fr
[2] ReDCAD, ENIS, University of Sfax, Tunisia
{bechir.zalila,mohamed.jmaiel}@enis.rnu.tn

Abstract. This paper defines a model-based approach, which treats the reconfiguration issues for Distributed Real time Embedded (DRE) systems at a high level of abstraction. We aim at specifying reconfigurable DRE systems using a characterization approach. To treat the reconfiguration requirements, we propose a meta-model and a UML profile as implementation of this meta-model. This leads to a simple way to model reconfigurable systems thanks to UML tools, our RCA4RTES meta-model and profile, and the MARTE profile and library.

1 Introduction

Embedded systems should respect the variation of execution environment and response to the evolution of user requirements during their execution. Most DRE systems are not fully autonomous and require human intervention to respond to events and to be reconfigured. However, human intervention are error prone and require more time and much efforts. It is sometimes impossible to stop a real-life time critical system for reconfiguration. In another side, the hardware resources of an embedded system are generally limited and their use has to be optimized. To develop a rich embedded system with several functionalities and low cost hardware resources, this system should execute, at a given instant, only the required software components. The hardware resources should be allocated only when required. It is very tedious and complex to develop such a system without providing a high-level of abstraction. Due to their complexity, DRE systems are usually more difficult to design than other types of applications, and in particular for reconfigurable ones. New modeling concepts are required to specify dynamic reconfigurations of these systems.

Our research work aims at proposing a model-based approach that allows to describe reconfigurable DRE systems at a high level of abstraction. The high level description can be used after some refinements to build the real system. It should allow the description of reconfigurable DRE systems with an undefined number of configurations. The dynamic reconfiguration allows modifying a system during its execution using architectural or behavioral reconfigurations.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 295–302, 2011.

The remainder of this paper is organized as follows. In Section 2, we briefly review some related technologies that address reconfigurability in embedded systems. In Section 3, we present in details our proposition to specify the dynamic reconfiguration in the context of DRE systems. As a proof of concept we present, in Section 4, a case study that has dynamic reconfiguration requirements: a GPS. Finally, Section 5 concludes this paper and presents future work.

2 Related Work

Several work have been carried out to specify embedded systems and particularly reconfigurable ones. A detailed state of the art has been presented in [1]. In the following, we only detail the two standards AADL and MARTE.

AADL (Architecture Analysis & Design Language) [2] is an architecture description language, which allows the specification of DRE systems as a component assembly. It allows to describe both the software and the hardware parts of a system. AADL allows also to specify reconfigurable systems using state machines describing modes and mode transitions. A mode presents a particular state (configuration) while a transition presents an event, which allows system reconfiguration. Compared to our approach, the modes in AADL are statically predefined. Thus, all possible system reconfigurations must be integrated in the model. This reduces considerably the modeling possibilities. AADL specifies embedded systems at a low level (thread, processor, etc), so that the modeling of reconfigurations is related to a specific application and platform.

MARTE (Modeling and Analysis of Real-Time Embedded systems) [3] is a UML profile for modeling and analysis of real time embedded systems inspired from the SPT profile [4]. It allows the separation in the specification of both the hardware and the software parts of platform resources, and the modeling of non-functional properties. It presents a set of packages which allows to specify a system at several levels of abstraction. Moreover, MARTE allows to specify the behavioral reconfigurations of real time embedded systems using state machines composed by a set of modes and transitions between them. Contrary to our approach, MARTE does not support distributed systems. It allows to specify only the behavioral reconfigurations of system with a predefined number of configurations.

3 The RCA4RTES Model Based Approach

We propose a model-based approach, called RCA4RTES, to specify reconfigurable DRE systems. We introduce the concept of *MetaMode*, which captures and characterizes a set of modes instead of defining each of them. A reconfigurable DRE system is specified with a non-predefined number of configurations. The *MetaMode* is described by structured component types, connectors as well as non-functional and structural constraints. The modes belonging to a *MetaMode*, are specified by the set of instances of structured component types and connectors defined by this *MetaMode*, which satisfy its constraints.

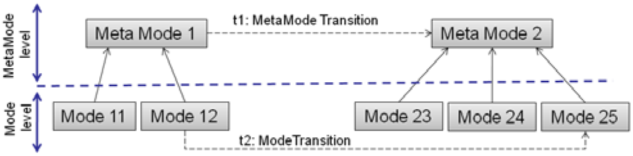


Fig. 1. MetaMode modeling

Our approach defines policy based reconfigurations. We specify dynamic reconfigurations using state machines, which define a set of *MetaModes* (states) and transitions between them. A *MetaMode* transition presents a set of reconfigurations between modes belonging to these *MetaModes* (as shown in Figure 1). When an event (presented as a *MetaMode* transition) is triggered, reconfigurations (i.e. presented as *mode* transition) are applied on the current mode to one of the modes belonging to the target MetaMode. Reconfiguration policies allow to automatically select the target mode. As examples of policies, we cite the selection of modes with low resource consumption, the selection of modes requiring limited reconfiguration actions, etc.

The switch from the source mode to the target mode is generated based on the corresponding MetaMode transition and the selected reconfiguration policies. It is made by a set of concrete reconfigurations such as adding and removing components and connectors and modifying the components properties. Figure 1 explains the previous concepts with a toy example. The designer specifies the dynamic reconfigurations of his DRE system using a state machine, which contains two *MetaModes*: $MetaMode1$ and $MetaMode2$. A transition $t1$ presents a reconfiguration between these two *MetaModes*. The mode transition *t2* is one of the possible transitions deduced from $t1$ thanks to reconfiguration policies. The current mode $Mode12$ is automatically replaced by $Mode25$.

Each MetaMode must be allocated on the hardware instance. As the hardware architecture is unchanged, the allocation is defined from software architecture models (*MetaModes*) to execution supports. The designer uses our approach to perform the following three-step process:

Step 1: Software part specification: the designer starts specifying the dynamic reconfiguration of his DRE system using a state machine, which is composed of a set of *MetaModes* and transitions between them. He specifies, for each defined *MetaMode*, its structural component types and connectors as well as its non-functional and structural constraints,

Step 2: Hardware part specification: the designer specifies the hardware architecture in terms of hardware components such as processor,

Step 3: Software part allocation on the hardware part: the designer allocates the specified *MetaModes* (defined in step 1) on the hardware architecture (defined in step 2). Some allocation constraints should be defined in order to specify the allocation policies defined the mapping from software models to hardware instances.

To describe software specifications of reconfigurable DRE systems, we define the RCA4RTES meta-model (section 3.1) and the RCA4RTES UML profile as an implementation of this meta-model (section 3.2).

3.1 The RCA4RTES Meta Model

To treat reconfiguration issues of DRE systems, we define the RCA4RTES meta-model. This meta-model is composed of four packages:

- **SWConstraintRTES:** specifies the allocation constraints as well as the structural and non-functional constraints assigned to *MetaModes*,
- **SWConfRTES:** describes configurations by specifying the system *MetaModes*. It imports the **SWConstraintRTES** to add constraints,
- **SWEventRTES:** specifies the events, which launch reconfigurations,
- **SWReconfRTES:** specifies the dynamic reconfigurations between *MetaModes*. It imports both **SWConfRTES** and **SWEventRTES** to define respectively *MetaModes* and events.

SWConfRTES package shown in Figure 2 introduces the meta-class *MetaMode*, which allows to present multi-state applications. A *MetaMode* is described by a set of structured components, connectors, as well as structural and non-functional constraints. We define the meta-class *StructuredComponent* composed of a set of interaction ports. The structured components can be periodic, sporadic or aperiodic threads. It can be also a composition of structured components. A connector which is presented by the meta-class *Connector* links two or more structured components or interaction ports. The constraints are defined in the **SWConstraintRTES** package. Each *MetaMode* has several instances (modes). For each mode, a configuration relates the mode to the deployment plan. A deployment plan describes a configuration by a set of structured components, the connections between them, their configuration, and their allocation to physical nodes. We introduce the meta-class *Allocation* to specify the allocation of *MetaModes* to execution supports, which will be specified using MARTE profile. This allocation implies non-functional and allocation constraints.

Our meta-model presents three kinds of constraints (figure 2), which are described in the **SWConstraintRTES** package:

- Structural constraints are related to the structure of architectures,
- Non-functional constraints specify conditions on the non-functional properties associated with *MetaMode* elements (i.e., components, connectors),
- Allocation constraints specify the policies used for the allocation of each *MetaMode* to a fixed hardware architecture (i.e., execution supports).

The *MetaModeChangeEventKind* enumeration in the **SWEventRTES** package presents two kinds of events that can be launched a *MetaMode* transition: an *application event* and an *infrastructure event*. An application event presents a configuration change in accordance with user requirements while an infrastructure event presents a variation of situation in the infrastructure.

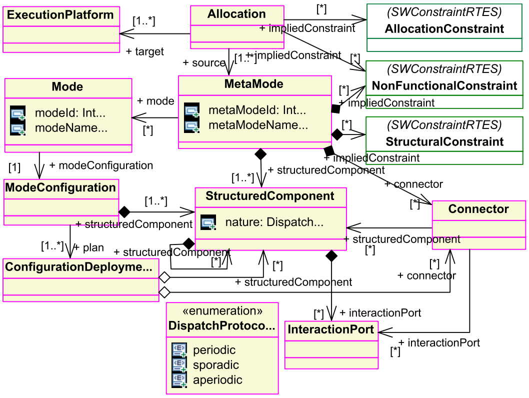


Fig. 2. SWConfRTES package

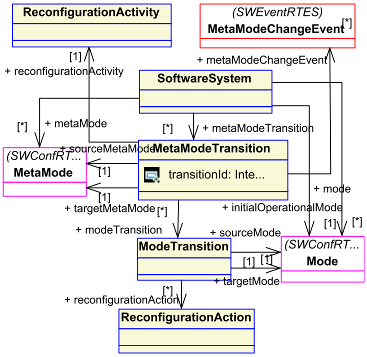


Fig. 3. SWReconfRTES package

The **SWReconfRTES** package presented in Figure 3 allows to describe the dynamic reconfigurations of DRE systems. The meta-class *SoftwareSystem* is composed of a set of *MetaModes* and *MetaMode* transitions. A transition allows switching the system from a *MetaMode* to another when an event is triggered. For each transition, an activity of reconfiguration is associated. It represents an algorithm for switching from the current configuration (Mode) to the target one. A *MetaMode* transition presents an abstraction of a set of mode transitions.

3.2 The RCA4RTES Profile

To handle reconfiguration requirements of DRE systems, we derive a profile from the RCA4RTES meta-model. This profile imports the NFP and the VSL profiles of MARTE profile [3] and the Basic NFP types of MARTE library [3]. The full profile description is given in Figure 4.

As we are interested in real time embedded systems, the structured components are considered as threads or a set of threads. In our profile, we define the following properties for defining and characterizing these threads:

- **Nature** defines the nature: periodic, sporadic or aperiodic thread,
- **Period** defines the period of a periodic thread or the minimal time between two activations of a sporadic thread,
- **Deadline** defines the deadline for periodic and sporadic threads,
- **StartTime** defines the startup time of an aperiodic thread,
- **EndTime** defines the end time of an aperiodic thread,
- **MemorySize** defines the storage size required by component for execution,
- **WCET1** defines the worst case execution time on a processor with 1 GHz of frequency. It presents the execution time of instructions on processor. It is defined by the ratio of the instruction number on the processor frequency. *WCET1* value varies according to the processor frequency,

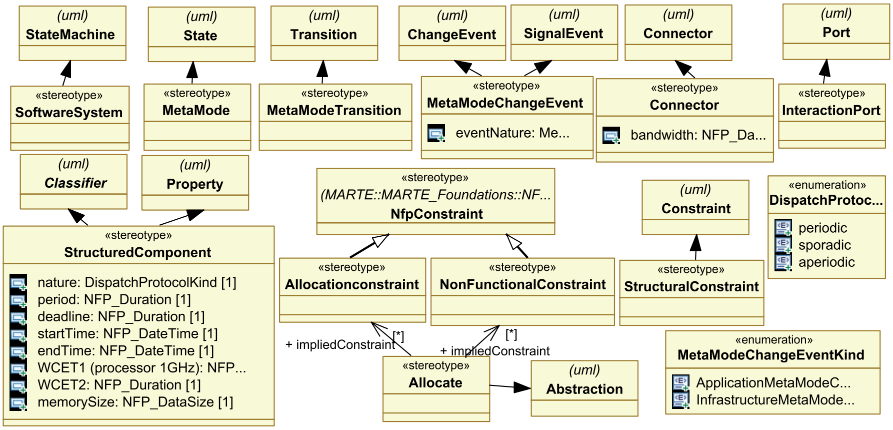


Fig. 4. RCA4RTES profile description

- **WCET2** presents a constant time such as a waiting time. *WCET2* value is fixed and the same for any processor frequency.

The stereotype *Connector*, which extends the meta-class *Connector* of UML is characterized by the property *bandwidth*. Figure 4 presents the properties assigned to both StructuredComponent and Connector stereotypes as tagged values and their types, which are imported from MARTE library. These properties are given in our profile in order to enable the verification of non-functional properties such as CPU usage, memory usage, etc.

The *StructuralConstraint* stereotype extends the meta-class *Constraint* of UML in order to specify architectural constraint using OCL (Object Constraint Language). Both *NonFunctionalConstraint* and *AllocationConstraint* stereotypes inherit from the stereotype *NfpConstraint* of NFP package of MARTE profile. This inheritance allows the designer to use VSL (Value Specification Language), which is an extension of OCL and allows to specify non functional properties and constraints as well as the complex expressions of time. To ensure the allocation of *MetaModes* to execution supports, we define the stereotype *Allocate* which extends the meta-class *Abstraction*. This stereotype is associated with non-functional and allocation constraints.

A reconfigurable DRE system is described by a state machine. For this reason, we introduce the stereotype *SoftwareSystem* which extends the meta-class *StateMachine* of UML. This state machine is described by a set of *MetaModes* and transitions between them. These transitions are activated by events. For that, we define both *MetaMode* and *MetaModeTransition* stereotypes, which extend respectively the meta-classes *State* and *Transition* of UML. The *MetaModeChangeEvent* stereotype extends both the meta-class *SignalEvent* and the meta-class *ChangeEvent*. An enumeration (*MetaModeChangeEventKind*) presents the two kinds of events is defined.

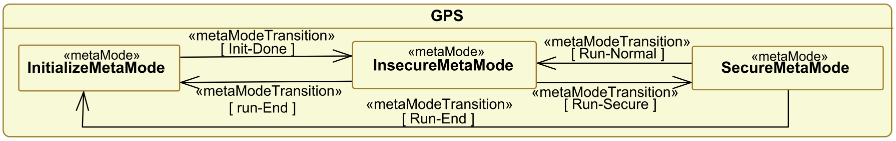


Fig. 5. The state machine of GPS case study

4 Case Study

In this section, we illustrate our proposed approach by a use case that has dynamic reconfiguration requirements: a GPS (Global Positioning System) [5]. GPS helps the user to determine the road to be followed from his current place to some specified destinations using information provided by an encrypted signal of satellite. This signal contains various information useful for localization and synchronization. The control base receives and sends information to satellites in order to synchronize their clocks. For simplicity sake, many functions of this case study have been omitted. We only detail the architecture of the terminal. Both satellite and control base are represented by basic components.

Following our approach, in the first step, we begin by defining a state machine specifying the dynamic reconfigurations. We use UML state machine diagram (Figure 5). We define three *MetaModes* of GPS: (1) `initialize MetaMode`, (2) `insecure MetaMode`, which consists of a traditional (or public) use of a GPS and (3) `secure MetaMode`, which represents a restricted use of a GPS with some safety requirements. The transition from one *MetaMode* to another is ensured by event triggering. For example, the switch from `insecure` to `secure` occurs when the monitor commands to drive in secure state. Then, we use the UML component diagram to describe each *MetaMode*, including structured components types, connectors, as well as non-functional and structural constraints. The top part of Figure 6 shows the `insecure` *MetaMode* of GPS. This MetaMode has seven structured component types (such as, GPS satellite, Receiver, etc) connected by a set of connectors. This *MetaMode* has a set of structural constraints defined by OCL language. We also specify the properties of each structured component and connector.

In the second step, we specify the hardware architecture of GPS (i.e. GPS terminal, GPS satellite and GPS control base nodes) in terms of hardware components (such as processor, memory, etc) using MARTE profile.

In the third step, we specify the allocation of *metaModes* to execution supports using UML component diagram. The allocation of insecure *MetaMode* to GPS terminal hardware and GPS satellite hardware is presented in Figure 6. The allocation constraints describe the policies of allocation of models to hardware instances. For example, the allocation of structured component `Encoder` instances is devised between the two processors `cpu1` and `cpu2` of GPS terminal.

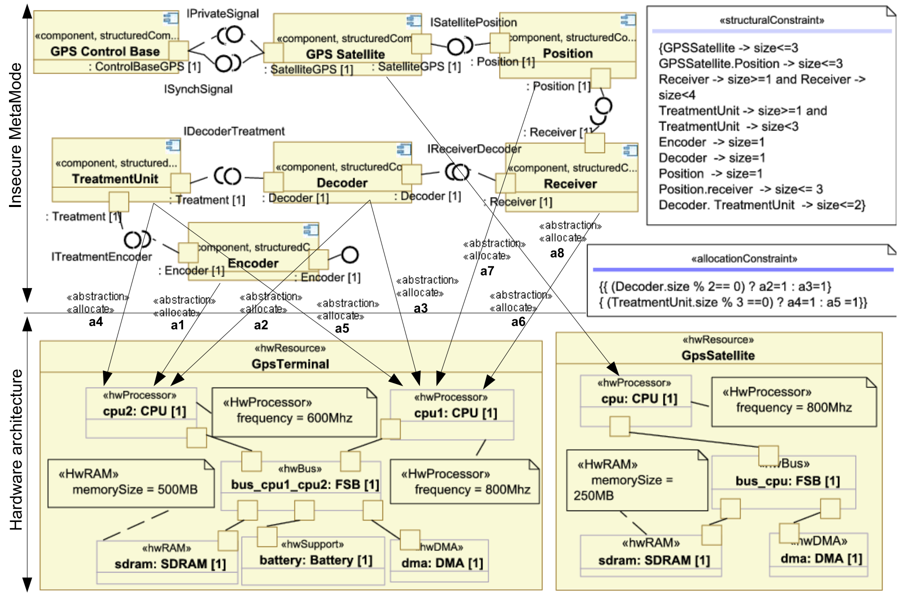


Fig. 6. Allocation of Insecure MetaMode to GPS terminal hardware and GPS satellite hardware

5 Conclusion and Future Work

In this paper, we presented a new model-based approach RCA4RTES to specify the dynamic reconfigurations of DRE systems at high level of abstraction. The dynamic reconfigurations are described by state machines composed of a set of *MetaModes* and transitions between them. We introduced the new concept MetaMode, which captures and characterizes configurations instead of defining each of them. Each *MetaMode* specifies a non-predefined number of configurations. To treat the reconfiguration issues, we introduced a new meta-model and an implementation as UML profile. As future work, we plan to investigate the verification of non-functional properties like CPU and memory usage.

References

1. Krichen, F.: Position paper: Advances in reconfigurable distributed real time embedded systems. In: International Workshop on Distributed Architecture Modeling for Novel Component Based Embedded Systems. IEEE, Los Alamitos (2010)
2. SAE: AADL (2009), http://www.sae.org/technical/standards/AS5506A
3. OMG: MARTE Profile (2009), http://www.omg.org/spec/MARTE/1.0/
4. OMG: SPT Profile (2005), http://www.omg.org/spec/SPTP/
5. Hamid, B., Krichen, F.: Model-based engineering for dynamic reconfiguration in DRTES. In: Workshop on Model-Driven Software Engineering. ACM, New York (2010)

An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle

Rafael Capilla[1], Olaf Zimmermann[2], Uwe Zdun[3],
Paris Avgeriou[4], and Jochen M. Küster[2]

[1] Universidad Rey Juan Carlos, Madrid, Spain
rafael.capilla@urjc.es
[2] IBM Research, Zurich, Switzerland
olz,jku@zurich.ibm.com
[3] Vienna University of technology, Vienna, Austria
uwe.zdun@univie.ac.at
[4] University of Groningen, Groningen, The Netherlands
paris@cs.rug.nl

Abstract. Software architects create and consume many interrelated artifacts during the architecting process. These artifacts may represent functional and nonfunctional requirements, architectural patterns, infrastructure topology units, code, and deployment descriptors as well as architecturally significant design decisions. Design decisions have to be linked to chunks of architecture description in order to achieve a fine-grained control when a design is modified. Moreover, it is imperative to identify quickly the key decisions affected by a runtime change that are critical for a system's mission. This paper extends previous work on architectural knowledge with a metamodel for architectural decision capturing and sharing to: (i) create and maintain fine-grained dependency links between the entities during decision identification, making, and enforcement, (ii) keep track of the evolution of the decisions, and (iii) support runtime decisions.

Keywords: architectural design decisions, architectural knowledge, metamodel, runtime decisions, traceability, evolution.

1 Introduction

Existing software architecture design processes [1] lack adequate mechanisms to explain the line of reasoning that architects follow in order to make design decisions. Reasoning about the architectural design is considered a tacit process that exists only in the architect's mind; the decisions that lead to a software architecture are often overlooked during architecture design and thus not systematically documented. In recent years, the software architecture community has established design decisions as first-class entities that should be captured alongside with other design elements. Therefore, the creation of software architectures is now also seen as the result of a set

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 303–318, 2011.

of design decisions rather than just as an assembly of components and connectors [2]. Making decisions explicit preserves architectural knowledge when staff is exchanged, e.g., when subject matter experts join the development team only temporarily or when transitioning from development to maintenance. As mentioned in [3], long-term benefits such as reduced maintenance effort should motivate users to capture the design rationale explicitly in the form of architectural decisions. This particularly holds true in successive iterations of the system as it evolves.

This paper extends previous work on architectural knowledge with a metamodel for architectural decisions to: (i) create and maintain fine-grained dependency links between the entities during decision identification, making, and enforcement, (ii) keep track of the evolution of the decisions, and (iii) support runtime decisions. Section 2 describes the background and the motivation of this research. In Section 3 we present a metamodel supporting traceability to keep track of the decisions made and their relations to design elements and artifacts. Section 4 then outlines the implementation of the metamodel in several prototype tools. Section 5 discusses a case study in the Service-Oriented Architecture (SOA) domain to demonstrate how the extensions of the metamodel are of practical use for SOA design. Section 6 describes the related work and section 7 summarizes the conclusions and future work.

2 Motivation and Problem Identification

A variety of research prototype tools have been developed to support design decisions in software architecture. From our experience developing and using various tools for architectural decision modeling, e.g., the Architectural Decision Knowledge Wiki [4], Architecture Design Decision Support System [5], and The Knowledge Architect [6], we observed three major shortcomings related to the creation and maintenance of the traceability links between the architectural knowledge and other artifacts:

1. The coarse *link granularity* in existing metamodels makes models easy to populate, but does not support a fine-grained tracing and tracking of decisions in relation to atomic design elements such as attributes in a class model or tasks in a business process model. Support for fine-grained trace links in current architectural decision modeling tools is weak or inexistent as some of the tools import UML design models externally and decisions can be only linked to coarse-grained artifacts.
2. Existing metamodels do not put special attention on *history and evolution of decisions*. Only a few of them treat evolution of decisions and architecture partially. One reason for this limitation is that most commercial and open source UML modeling tools do not offer explicit support for architecture evolution (e.g., Jude Community, Magicdraw).
3. The decision making process suggested by existing metamodels assumes that all decisions can be made at design time; *deferring decisions to runtime* is not supported. At present, the existing architecture decision modeling prototype tools do not offer support for runtime decisions that can be traced back to the architecture or to requirements when a piece of code or a system module change.

The first problem area addressed in this paper is *link granularity*. Links connecting key design decisions to architectural artifacts should include relationships to smaller parts of the design. Such an approach helps to achieve the precision required to estimate the impact of changes accurately. Small but important decisions should also be captured and linked properly. For instance, a decision to introduce a new UML package or class seemingly constitutes a more coarse-grained decision than the decision to add a new attribute to an existing class; however, the attribute may express a key architectural concern, e.g., it might flag an architecture component to be subject to financial and general IT controls audits or it might demarcate a system transaction boundary in a service composition. In many cases, fine-grained decisions are derived from coarse-grained ones made before; however, the lack of accuracy of existing traceability models do not offer a way to track the impact on the design or code. Thus, it is required to introduce trace links with narrower and more precise scope to achieve more precision in the traceability of architectural decisions during decision identification, making, and enforcement.

The second problem pertains to the maintenance of a system, as the design decisions made in the past might become obsolete, and the *history and evolution of decisions* should be recorded in the same way versioning repositories store the history and evolution of source code. This is useful for a number of reasons. In certain cases during system evolution, the architects have to revisit past decisions and revert to them if a new decision appears to be wrong. In other cases, architects may need to roll back the design, and start a new decision path from that point. Finally new stakeholders that become involved in a project can be educated much more efficiently by studying the evolution of decisions over time and the rationale that lead to the existing set of decisions and the present design.

As a third problem, we observed that today the dynamicity of certain systems may imply that certain decisions affect architectures that have already been deployed but have to be modified during runtime. For instance, a composite service which replaces an atomic service with another one due to new quality-of-service conditions during execution requires *deferring decisions to runtime*. Such deferred decisions have to be tracked back to the architecture and requirements so that conformance to them can be ensured. Supporting runtime decisions becomes increasingly relevant in modern operating environments and deployment infrastructures such as virtualized data centers: each instantiation of a virtual software image may decide for a slightly different set of quality properties. Examples include the heap and disk size of virtual UNIX machines (*infrastructure-as-a-service scenario*), Java and relational data source settings of Web application servers (*platform-as-a-service*), and login and encryption policies of hosted Web conferences (*software-as-a-service*). These decisions are based on user preferences and current resource consumption (system load); these two types of decision drivers only become known at runtime. Consequently, it makes sense to defer the detailed architectural decisions about these infrastructure settings to runtime (while at design time certain architectural templates that constrain the runtime configuration options can be predefined).

In our previous work [4, 7] we introduced a conceptual framework for decision modeling with reuse to extend recent research on design decisions. Our work focused on the following main contributions:

1. A **decision-making process** which comprises *decision identification* to delimit the scope, *decision making* to choose a feasible design alternative for each design issue, and *decision enforcement* to share the results of the decision making step with relevant stakeholders.
2. A **decision-capturing and sharing metamodel** supporting the decision making process. This metamodel is specified as a Unified Modeling Language (UML) class diagram and a formal definition based on elementary set and graph theory [4]. The metamodel, illustrated in Figure 1, relies on three main core domain entities: *ADIssue, ADAlternative*, and *ADOutcome* (*AD* stands for *Architectural Decision*). An *ADIssue* captures an architectural problem that requires a design solution whereas *ADAlternative* instances capture the pros and the cons of the design choices an architect has (i.e., the possible solutions available and the criteria for choosing or not choosing such option). Finally, *ADOutcome* instances capture project-specific knowledge including the justification and the consequences of decisions actually made. This metamodel is implemented in the Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool, which is a collaboration system and decision modeling tool [4]. Other existing tools are based on similar metamodels [5], [6].

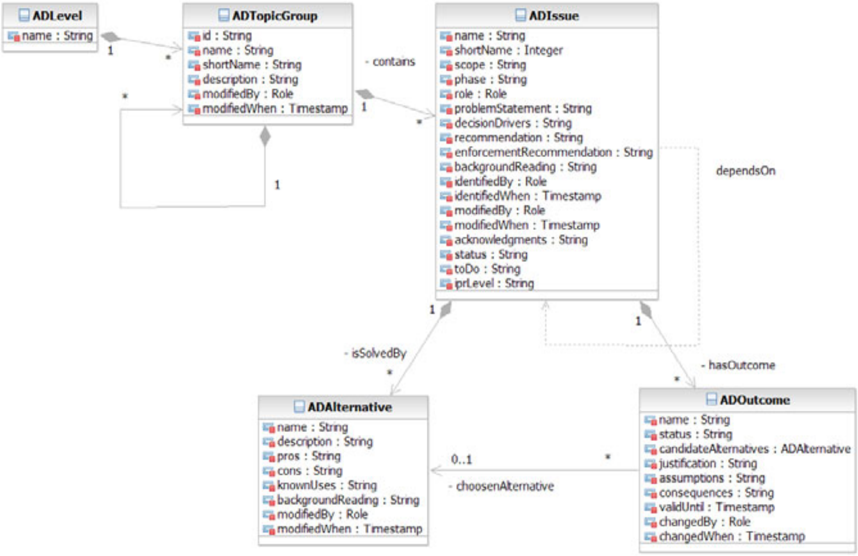


Fig. 1. Metamodel for architectural design decisions implemented in the Architectural Decision Knowledge Wiki tool

With regards to the problems of link granularity, history and evolution of decisions and deferring decisions, the existing metamodel does not offer support. We will later explain how it can be extended to support these concepts. We worked with more than one hundred practicing architects, who applied and appreciated the metamodel as well as the SOA guidance model instantiated from it [4], [7]. As part of our validation activities, we conducted a user survey. Among other things, users pointed out:

1. Decisions have to be visited multiple times and sometimes revised as the design evolves; any waterfall process or big design upfront is not adequate for most real-world projects. Decisions are hardly made in isolation.
2. The lifetime of decisions transcends their identification, making, and enforcement; they have to be evaluated once a system is implemented, at least in prototypical form. Only then it becomes evident whether made decisions have led to a design and implementation that allows the system to meet the quality attributes that have been stated for it.
3. There is a desire to model links from decisions to other model elements and artifacts represented more explicitly (e.g., types of requirements appear as decision driver text in the metamodel in Figure 1, but are not first class metamodel entities that can be linked to). The scope attribute of an issue (in the metamodel in Figure 1) can identify the type of design model element an issue pertains to, but at present this textual information does not link to any artifacts used in the design process.

The metamodel extensions specified in this paper are motivated in this user feedback. We base our proposed metamodel extensions on the metamodel that underlies in Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool because this tool is populated with a SOA guidance model comprising more than 500 issues and 2000 alternatives recurring in SOA design; architectural patterns described in the literature are among these alternatives (only a subset of these issues and alternative descriptions have been published so far). Hence, we count on a significant amount of knowledge to describe different types of design issues from a realistic point of view. However, our metamodel extensions are designed in such a way that they can be implemented in other tools as well (assuming that these tools support extensibility of their respective metamodels). To support this claim, we outline how we implemented the new concepts in an extensible commercial requirements engineering product later in this paper

3 Enhanced Trace Links and other Metamodel Extensions

To overcome the three problems mentioned before, we extended the conceptual metamodel of Figure 1. Our main rationale for adding new elements is to support explicit trace links to small architectural artifacts that help to check the integrity of the decision network, to evaluate the impact of changes, to keep track of the history and evolution of changes, and to record the root causes of changes. This new metamodel is shown in Figure 2. In the remainder of this section we describe the new classes and new elements highlighting them in italicized text.

Links to Design Artifacts: Two new classes, *ADDesignElement* and *ADDesignArtifact*, specify the parts of the architecture that result from one or more design decisions represented by outcome instances. *ADDesignElement* instances represent elements of modeling languages. For example, if we map to Unified Modeling Language (UML), it refers to a UMLNamedElement (i.e., any UML element that can be named). This includes coarse grained elements such as components and connectors, but also more fine grained elements such as class attributes. *ADDesignArtifact* aggregates and assembles such elements into project deliverables such as a platform-independent, technology-neutral functional component model. *ADDesignElement* instances are defined to have an *ADDesignElementType*, which also becomes the type of the scope attribute of the *ADIssue* class. In the architectural decisions viewpoint, the relationships between two newly introduced subclasses of *ADOutcome*, *ADDecidedOutcome* and *ADDeferredOutcome* (the existing metamodel introduced the *ADOutcome* class to record actual decisions made to solve a problem including its rationale), and *ADDesignElement* (with subclass *ADRuntimeElement*, introduced below) allows us to define trace links to individual parts of an architecture. *ADDecidedOutcome* and *ADDeferredOutcome* indicate that enforcing a decision at design time differs from enforcing a decision at runtime (with respect to the artifacts in which the decision materializes; e.g., UML class or conceptual application server node at design time vs. Java class or XML deployment descriptor at runtime). Such fine-grained linkage down to the level of individual architectural elements (e.g., UML components and connectors, physical topology units and hosting links, attributes of UML components or Java classes or XML elements) increases the precision and expressivity of the decision models. In summary, we have now introduced external links from decisions to structural and behavioral models, which were not supported previously.

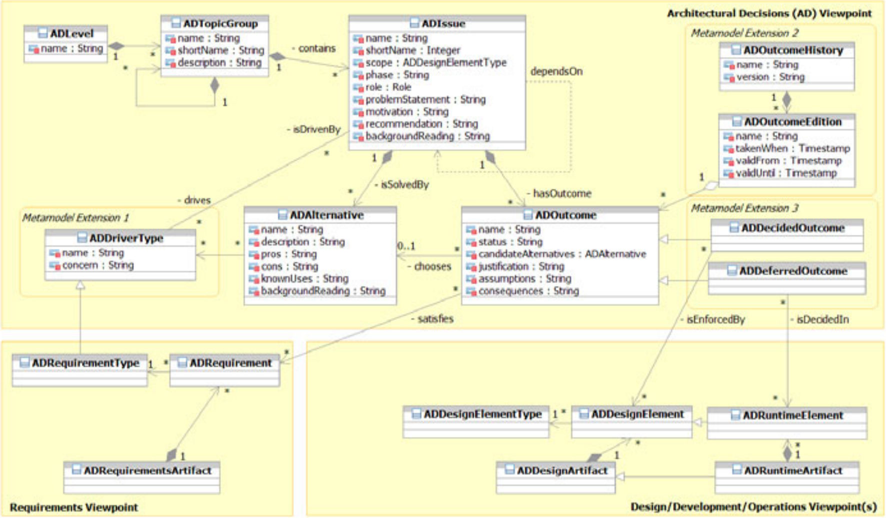


Fig. 2. UML metamodel for capturing design decisions with focus on maintenance, evolution, and runtime concerns

In the decision making process, several alternatives (*ADAlternative*) can be captured, considered, and evaluated before a decision is made. An external link, from requirements to decisions, can be established via the new class *ADDriverType*, which gathers the origins and influencers of decisions, such as types of functional and non-functional requirements. Because an issue is a reusable knowledge entity, the

ADDriverType class supports only types of requirements (e.g., quality attributes such as performance and modifiability), but not real instances of such requirements: the additional class *ADRequirement* serves this purpose. *ADRequirement* instances may represent analysis artifacts such as business process models, use cases, or user stories as well as non-functional requirements such as software quality attributes (e.g., sub-second response time performance, modifiability via multi-platform support, etc.). *ADRequirementsArtifact* instances compile a number of individual requirements. Each *ADRequirement* instance is classified by its kind, which is expressed by the *ADRequirementType* class. As a result of the improvement, we removed the decisionDrivers attribute initially defined in the *ADIssue* class (e.g., a problem that has to be solved). Thus, the new metamodel supports now full traceability from requirements to decisions and other design artifacts.

Decision History and Evolution: The evolution of decisions is described by means of the *ADOutcomeEdition* class, which establishes a chain of decisions that change over time. For instance, a corporate system may have to replace its middleware after several years of successful production use because new enterprise-level requirements demand a technological change in the organization. Hence, this decision made in the past for selecting the right middleware may have became obsolete and may have to be replaced by a new one. The *ADOutcomeHistory* class keeps track of the history of changes to a decision made years or months ago (i.e., collections of related *ADOutcomeEdition* instances, each of which referring to a single *ADOutcome instance*).

Support for Runtime Decisions: Some systems may change their status, operation mode (e.g., a system that updates its software version changes its operation mode from normal operation to maintenance mode until the reconfiguration process finishes and the system returns to the normal mode), or configuration during runtime due to external or internal conditions. Hence, the decisions that led to, for instance, a given product architecture might have to be modified, and in some cases lead to a different architecture. In such cases, certain decisions have to be replaced temporarily by new ones or they can also become obsolete for a given time period. Therefore, we introduce the *ADRuntimeElement* class (atomic) and the *ADRuntimeArtifact* class (composite) to reflect such situations and represent the code pieces that enforce the decisions represented by instances of the *ADDeferredOutcome* class. As decisions that change during runtime cause the architecture to be modified according to the depth of the change, adding support for runtime decisions improves traceability between artifacts; runtime artifacts can serve as link targets. These finer grained traceability links can determine the parts of architectures that have to be modified when changes happen. To our knowledge, this feature has not been implemented before in other tools and models capturing design rationale. Hence, we extend and enhance previous works for systems that require more surveillance or adaptability due to, for instance, new context conditions. Examples of issues that cannot always fully be resolved at design time are:

- Specifically to Service-Oriented Architecture (SOA), capturing runtime decisions and linking these to code assets is required. For example, our metamodel can describe the decision in a composite Web Service (a type of design element) to dynamically modify the Business Process Execution Language (BPEL) workflow that realizes the composite Web service, e.g., to engage a new subprocess to reflect a certain business rule or other runtime condition. Such late decision is often based on new quality-of-service conditions that modify the Service Level Agreement (SLA) for a given period (e.g., regarding guaranteed response times). Our metamodel uses the classes *ADRuntimeArtifact* and *ADDeferredOutcome* to express such situations.

- The decision how to route a service invocation request that represents an atomic activity in an executable business process model (i.e., *dynamic service composition*). Note that this decision can only be deferred to runtime if such flexibility does not violate regulatory constraints such data privacy and system and process assurance compliance (such concerns can be modeled as AD*DriverType* and linked to issues according to the metamodel presented in Figure 2).
- The decisions enable to customize certain software features when reusing a particular application package, middleware component, or product family (e.g., using variation points in software product lines [8], [9]). For instance, a database management system might support distributed two-phase commit (2PC) protocol at an extra performance and license cost; when the decision to use the system is made, it might not be known yet whether the 2PC support is required. This decision might even change over time, which can be expressed as a series of chained AD*OutcomeEdition* instances.
- The decision to delegate some of the responsibilities to end users that are performed by architects/developers in traditional software engineering (*situational application* development via Web-centric container architectures such as mashups). For instance, such design issues might deal with user interface patterns, data formats (e.g., MIME types), and information provider selections.

4 Implementation in Existing and Emerging Tools

This section outlines how the enhancements in the extended metamodel can be supported by three existing architectural knowledge management and modeling tools: ADDSS [5], The Knowledge Architect [6], and Architectural Decision Knowledge Wiki/Web Tool [4]. These tools share several goals and usage scenarios, but differ in their origins, use cases, and tool architecture. We discuss all three independently developed tools to illustrate the generality of our approach by explaining how the extended metamodel can be supported by them. In addition, we present an actual implementation of the extended metamodel on top of a commercial requirements engineering and management platform which supports metamodel extensions and Web-based artifact linking.

4.1 Architecture Design Decision Support System (ADDSS)

In this tool [5], the model underlying the tool supports explicit traces to requirements (*ADDriverType*) and architectures (*ADDesignElement*, *ADDesignArtifact*) as well as between design decisions, but links between decisions and smaller parts of the architecture can not be specified in a fine grained fashion. To overcome this, Figure 2 specifies a class *ADDesignElement* and establishes links from the *ADOutcome* to provide fine grained links to small design artifacts. Evolution in ADDSS is only supported by several attributes; there is no way to define a chain of decisions history as in the proposed metamodel of Figure 2 (using the ADOutcomeEdition and *ADOutcomeHistory* classes). Finally, ADDSS does not support runtime decisions like in our proposed solution. Hence, the *ADRuntimeElement*, *ADRuntimeArtifact* and *ADDeferrredOutcome* classes should be incorporated into ADDSS' metamodel to enable tracking runtime decisions.

4.2 The Knowledge Architect (KA)

This tool suite [6], [10] is comprised of a number of specialized tools for capturing, (re)using, translating, sharing, and managing software architectural knowledge. The Knowledge Architect entails specialized support for integrating the various architecting activities [11] and supporting collaboration between the stakeholders of these activities. The different tools support different activities (e.g. analysis, design, sharing) and therefore each tool has a specialized Architectural Knowledge (AK) metamodel to deal with the different types of knowledge produced and consumed during the architecting process. The different metamodels are integrated into the central knowledge repository of the tool suite. Traceability can be achieved in two ways: a) within each metamodel, traceability links are established between the AK concepts (e.g., between "decisions", "concerns", "decisions topics" and "alternatives" in the document knowledge client of the KA) b) across different metamodels traceability links can be established within the knowledge repository (e.g. "decisions" and "concerns" are common concepts of both the document knowledge client and the analysis model knowledge client of the KA). The KA can be extended in two ways to support the metamodel of Figure 2: a) all the tools have extensible metamodels (not hard-coded but completely customizable), thus the new concepts and relations can be added in a straightforward way; b) the central knowledge repository itself stores knowledge in Resource Description Framework (RDF) format and can directly accommodate the metamodel extensions of Figure 2. As an example the classes *ADDecideOutcome* and *ADDeferredOutcome* can simply inherit from the class *Decision*, while *ADDriverType* can inherit from the class *Concern* (both Decision and Concern belong to the document knowledge client metamodel). The extensions for history and evolution are not necessary to be implemented as the KA, as the tool suite uses the versioning system of Sesame to track the evolution of each knowledge entity.

4.3 Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool

Architectural Decision Knowledge Wiki is a Web 2.0 collaboration tool supporting the decision modeling capabilities and original UML metamodel first published in [7]. A version 1.0 was originally implemented in PHP and released in March 2009; in October 2009, a Java reimplementation of the tool was released under the name Architectural Decision Knowledge Web Tool [4]. The tool supports about 50 decision modeling and making use cases. It assembles *ADIssue* and their *ADAlternative* on a *decision identification* tab (these metamodel entity types are jointly referred to as *decisions required*). *ADOutcome* instances are created and updated on a second *decision outcome* tab (capturing *decisions made*), which exposes a simple decision state management workflow to the user (with open/decided/approved/rejected states). To support the extended metamodel introduced in the previous sections, the following additional features and components are required:

1. The *ADDriverType* class is a result of refactoring the decision driver attribute in *ADIssue*; hence, the new capability can be implemented by *refactoring* the user interface components displaying the decision identification tab as well as the underlying server-side business logic and database schema. Having performed these refactorings, the fine-grained traceability links can be added to the decision identification tab; advanced user interface features such as pop-ups can be added.
2. The *ADOutcomeHistory* and *ADOutcomeEdition* classes can be realized by implementing the *edition pattern*. The business logic and the database schema of the existing implementation already do so; on top of that, an additional *decision evolution* tab can be added to the user interface to display the decision making history.
3. Deferring decisions to runtime can be supported by introducing a *new state* "deferred" for outcome instances; this requires to update the user interface components supporting the decision making tab, as well as the state machine implemented in the business logic realizing *ADOutcome* instance creation and lifecycle management.

4.4 Implementation in IBM Rational Requirements Composer

To investigate and demonstrate the technical feasibility, practicality, and usability of these enhancements, we created a demonstrator in a requirements modeling and management platform prior to implementing them in the actual tools (following the well-established design principles such as user interface storyboarding and prototyping).

For our proof-of-concept we used a recently released requirements engineering and storyboarding tool, *IBM Rational Requirements Composer (RRC)*. Version 2.0 of this Jazz repository-based product became generally available on jazz.net in November 2009. The RRC metamodel by default supports artifacts such as business process models, use case diagrams, storyboards, but also supplemental rich text documents representing features and non-functional requirements. All artifacts as well as external resources can be linked to each other via Web URLs. Via attribute groups, the default metamodel can be extended.

We first created custom attribute groups to represent the original metamodel and then added new attribute groups representing *ADDriverType* and *ADDeferredOutcome*. *ADOutcomeHistory* does not require product configuration; it is supported by the server component of the RRC product (via the snapshotting capabilities which stores model versions in the Jazz repository). Next, we instantiated SOA model elements (instances) via templates we created from sample rich text artifacts which use the newly defined attribute groups. The sample model elements were populated from the existing SOA guidance model available in Architectural Decision Knowledge Web Tool (via copy-paste). Finally, fine grained traceability links were added to demonstrate requirements to decisions linkage.

The sample links from requirements to issues and back (introduced in the previous section and shown in the extended UML diagram in Figure 2) demonstrate the technical feasibility of our concepts; the links reside on the individual requirement/issue/outcome instance level, not on document-to-document level. This paves the way for requirements to decisions integration as suggested by our metamodel extensions. Concerns expressed as *ADDriverType* become first class citizens in the user interface (tagged as architecturally significant requirements) and the architecture of the tool (unlike in the original implementations). In conclusion, this implementation demonstrated that the extended metamodel is generic and expressive enough to be supported in multiple tools.

5 Instantiation for SOA Enterprise Applications

We applied our extended metamodel to an industrial case study from the telecommunications industry. This industrial case study concerns the modernization of an existing, business-to-business *order management system* (OM) in a major telecommunications company employing a wholesaler-retailer business model [12]. In this business process-centric scenario, a key business requirement (concern) was to ensure enterprise resource integrity over multiple channel interactions and time. User channels included the Internet (providing end user self services) and call centers. Two of the order management processes consisted of up to 19 steps and could run for up to 24 hours. Market deregulation and increasing competition caused the concrete problem of having to coordinate competing requests for the same physical resources in the shared telephony network. This coordination was seen to improve customer satisfaction (measured as number of successful order requests).

This business environment led to many architectural design challenges. Key technical requirements in this order management context were multi-channel request coordination and process instance and timeout management. A business transaction started via the Internet-based self-service channel had to be able to continue via call center (back office) interaction. Different VSP retailers reserved resources in a single network owned by the wholesaler, so incomplete requests had to be undone after a certain amount of time. The system context and resource integrity management requirement suggested introducing a process layer as a governing architecture element. This process layers serves one user channel per user type. These channels reside in the presentation layer of the order management system. The required long-running process instance tracking and timeout management could be implemented in a macroflow engine [13] dedicated for this task (called). Short-running, transactional flows could be handled by dedicated microflow engines [13].

All these concerns are addressed in the logical architecture of the production solution which is outlined in Figure 3 and explained in detail in [12]. While such UML class diagram can give an architectural overview, many detailed concerns cannot be covered on this level of refinement. For instance, many technology- and product-specific design issues and the rationale of the decision outcomes should be explained in detail elsewhere. More specifically (in the context of this paper and the proposed metamodel extensions), the architecture elements should be traced back to the outlined requirements, the evolution of the system from a plain Java Web application to a process-based SOA should be captured, and the necessity to defer certain decisions to runtime should be captured.

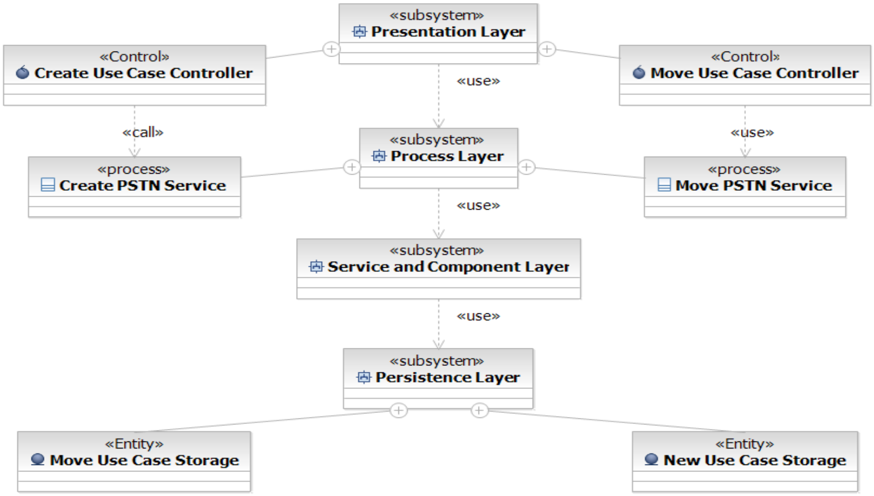


Fig. 3. Functional components of the order management system

Let us map the model elements in Figure 3 back to the metamodel from Figure 2. All UML classes representing functional components are instances of ADDesignElement (irrespective of their stereotypes); the class diagram itself is an instance of ADDesignArtifact. The ADDesignArtifactType of this class diagram artifact is “functional component model”; the ADDesignElementType of the ADDesignElement instances is “(functional) UML component” (we can view component stereotypes such as “subsystem”, “control component”, and “process component” as subtypes; however, this subtyping is not expressed by our metamodel). Example of traceability links will be given in the next subsection and Figure 5. We uses the extended metamodel of Figure 2 to illustrate how these design/modeling problems in the Order Management (OM) case study can be modeled.

Early in the project, a decision was required to decide for the main architectural concepts. In particular, a process-based SOA and the related architectural patterns were chosen because the solution was supposed to be flexible and adaptable. One of the important conceptual decisions in this context was to decide whether a service composition layer should be introduced into the architecture (the outcome of this decision led to the inclusion of the Process Layer component in Figure 5).

Figure 4 shows a (heavily simplified) instance of the metamodel for this decision, working with a subset of the design elements from Figure 3. Both instances of the core classes of the existing metamodel (*ADIssue, ADAlternative, ADOutcome*) and our metamodel extensions are illustrated (*ADRequirement*, *ADDesignElement*, *ADOutcomeHistory*, etc.). A sample decision <<ADReqType>> Portability and a concrete <<ADRequirement>> Runs on 2 Platforms (i.e., solution can on at least two platforms) were identified for one required and made decision (<<ADIssue>> Workflow Language with selected <<ADAlternative>> BPEL).

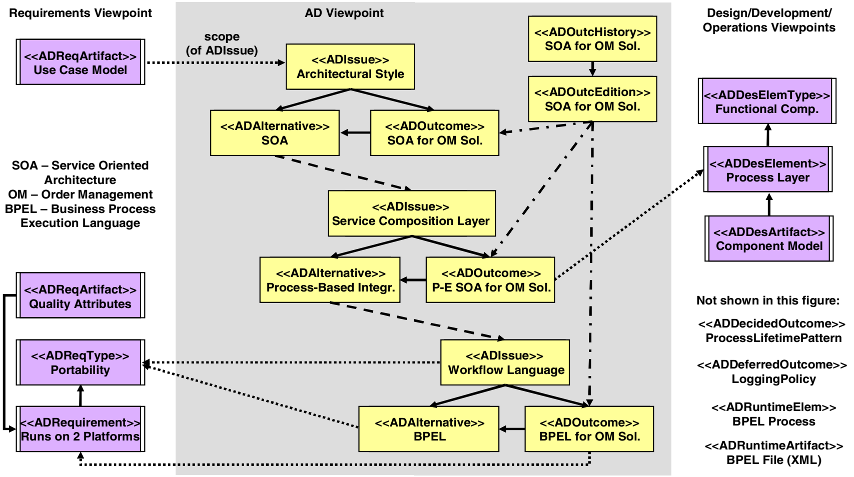

Requirements Viewpoint
AD Viewpoint
Design/Development/ Operations Viewpoints
<<ADReqArtifact>> Use Case Model
scope (of ADIssue)
<<ADIssue>> Architectural Style
<<ADOutcHistory>> SOA for OM Sol.
<<ADOutcEdition>> SOA for OM Sol.
<<ADAlternative>> SOA
<<ADOutcome>> SOA for OM Sol.
<<ADDesElemType>> Functional Comp.
<<ADDesElement>> Process Layer
<<ADDesArtifact>> Component Model
SOA – Service Oriented Architecture
OM – Order Management
BPEL – Business Process Execution Language
<<ADIssue>> Service Composition Layer
<<ADAlternative>> Process-Based Integr.
<<ADOutcome>> P-E SOA for OM Sol.
<<ADReqArtifact>> Quality Attributes
<<ADReqType>> Portability
<<ADRequirement>> Runs on 2 Platforms
<<ADIssue>> Workflow Language
<<ADAlternative>> BPEL
<<ADOutcome>> BPEL for OM Sol.
Not shown in this figure:
<<ADDecidedOutcome>> ProcessLifetimePattern
<<ADDeferredOutcome>> LoggingPolicy
<<ADRuntimeElem>> BPEL Process
<<ADRuntimeArtifact>> BPEL File (XML)

Fig. 4. Architectural decisions made in case study with links to design model context a.k.a. exemplary application (instantiation) of the AD metamodel for the case study

At this stage, we couldn't test the evolution of the decisions as we only produced the first version of the architecture of the OM system using, but we captured the evolution of the system from a plain Java Web application to a process-based SOA.

Furthermore, decisions that might change at runtime can be tracked using the proposed metamodel extension (i.e.: AD*RuntimeArtifacts*, AD*RuntimeElements*) and the class that enforces the decisions (AD*DeferredOutcome*). In the order management SOA, the system transaction boundary and the logging settings might differ for certain components in the process layer and for components in the service layer shown in Figure 3. This metamodel extension is not illustrated in Figure 4.

6 Related Work

Several research prototype tools [11], [14] for capturing, using, and documenting architectural design decisions have recently appeared; many of these use templates and metamodels for capturing knowledge attributes and managing decision dependencies [15], [16]. Tools such as PAKME, ADDSS, Archium, The Knowledge Architect, and AREL offer traceability mechanisms between decisions and other software artifacts at different levels. Some of these tools support the evolution of trace links between decisions and forward and backward traces. The traceability supported by the tools can be used to estimate those artifacts that are impacted by the change in a decision, as the majority of the mentioned tools lack fine grained links between decisions and small architectural artifacts (e.g., a UML class or component instead of an entire subsystem). In addition, the approach presented in [17] highlights the role of traceability in software architecture evolution and describe a method to manage such traceability for design decisions using a model-driven development approach.

Software product lines (SPL) need to model also the dependencies of feature models (i.e.: in practice they constitute a decision model) for different phases of the software life-cycle. Modeling dependencies and dealing with traceability problems in SPL is discussed in [18], where a wide list of dependency types between features are defined as constraints a software product must satisfy, while in [19] the authors explain how metamodels from PAKME and ADDSS tools can be merged to support product lines concepts and model dependency links between architectural design decisions and the variability rules associated to a feature model. Other works refer to Dynamic Software Product Lines (DSPLs) [20] to provide the necessary binding for runtime variation points to adapt the software to changes in the environment. The authors state that it is impossible to foresee al the variability a SPL requires, and use dynamic architectures and support for runtime decisions to be able to support system configuration and binding at runtime (for automatic decision-making). Designing and managing runtime variation points in architecture is also described in [21], where patterns are used to provide such facility in SPL and add the necessary flexibility for domain-specific applications (e.g.. custom Web servers that cannot be stopped when deploying or configuring components).

Lago et al. [22] discuss three different traceability issues during SPL derivation, and they focus on those traceability links between feature models and structural models (i.e.: architecture-level decisions). In [23], a Dependency Structure Matrix (DSM) is used to represent and manage dependencies in complex software architecture and to reveal underlying architectural patterns. Acceptable and unacceptable dependencies are expressed using design rules to describe the semantics of such dependencies.

All the aforementioned approaches lack explicit support for runtime decisions that can be deferred and tracked back from code to the architecture and to the design decision. Furthermore, in most cases they support coarse grained links between decisions and other software artifacts. Evolution is only partially supported in two existing tool prototypes. Hence, our approach improves these features and enriches previous metamodels and tools with runtime decisions. Other approaches that consider fine grained traceability paths between different artifacts do not consider the inclusion of design decisions as we do.

Traceability between decisions and from decisions to artifacts is related to traceability between requirements and model elements in general. This general problem of establishing and maintaining traceability has been studied in the literature and different approaches exist. Maeder et al. [24] present an approach for automating traceability maintenance under changes by classifying changes and automating updates of the traceability graph. Such an approach could in principle also be applied to traceability management for architectural decisions. Cleland-Huang and Chang [25] propose a traceability method that is based on the publish-subscribe architecture in order to keep traceability links up to date. It remains for future work to investigate the best approach to maintain traceability links between architectural decisions and requirements.

7 Conclusion and Future Work

Our approach revisits and enhances previous models and tools as we provide full traceability between individual decisions and other software artifacts using fine grained links, even if the decision networks becomes more complex to manage and to maintain. We are aware that capturing fine grain trace links introduces additional costs to maintain the links over time and this cost should not be higher than the expected benefits, but the architect must decide when to define such links to smaller parts of the architecture that must be traced (e.g., a critical software component in a system composed by a few number of classes is replaced at runtime by a new component with extended functionality and defined by a new UML for which a new trace link must be create for its corresponding design decision). With such links we achieve a better control of individual decisions and we are able to find out in detail which parts of the architecture are affected by a change in the requirements or code. Because certain software systems may vary their context conditions during runtime, they require adequate models to support runtime decisions that can be deferred. Hence, we extend previous works to track runtime decisions and make software architects aware of changes that may affect the design. Other extensions would include supporting the full context of decisions that evolve and store not only the decisions but also the issues, drivers, and requirements that accomplish a particular solution. Finally, other non-SOA domains like self-adaptive systems can benefit from tracking runtime decisions as a way to monitor better those changes that happen during system execution.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)
2. Bosch, J.: Software Architecture: The Next Step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004)
3. Kruchten, P., Capilla, R., Dueñas, J.C.: The Decision's View Role in Software Architecture Practice. IEEE Software 26(2), 36–42 (2009)
4. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Journal of Systems and Software 82(8), 1249–1267 (2009)
5. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: Web-based Tool for Managing Architectural Design Decisions (SHARK'066). ACM SIGDOFT Software Engineering Notes 31(5) (2006)
6. Jansen, A., de Vries, T., Avgeriou, P., van Veelen, M.: Sharing the Architectural Knowledge of Quantitative Analysis. In: Proceedings of the Quality of Software-Architectures (QoSA), pp. 220–234 (2008)

7. Zimmermann, O., Gschwind, T., Küster, J.M., Leymann, F., Schuster, N.: Reusable Architectural Decision Models for Enterprise Application Development. In: Overhage, S., Ren, X.-M., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 15–32. Springer, Heidelberg (2008)
8. Bosch, J.: Design and use of Software Architecture: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Reading (2000)
9. Pohl, K., Böckle, G., Linden, F.v.d.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
10. Liang, P., Jansen, A., Avgeriou, P.: Collaborative Software Architecting through Architectural Knowledge Sharing. In: Finkelstein, A., Grundy, J., van der Hoek, A., Mistrík, I., Whitehead, J. (eds.) Collaborative Software Engineering (CoSE), pp. 343–368. Springer, Heidelberg (2010)
11. Liang, P., Avgeriou, P.: Tools and Technologies for Architecture Knowledge Management. In: Software Architecture Knowledge Management: Theory and Practice, pp. 91–111. Springer, Heidelberg (2009)
12. Zimmermann, O., Doubrovski, V., Grundler, J., Hogg, K.: Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). ACM Press, New York (2005)
13. Hentrich, C., Zdun, U.: Patterns for Process-Oriented Integration in Service-Oriented Architectures. In: Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, pp. 1–45 (July 2006)
14. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Babar, M.A.: A Comparative Study of Architecture Knowledge Management Tools. Journal of Systems and Software 83(3), 352–370 (2010)
15. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22 (2005)
16. Kruchten, P., Lago, P., van Vliet, H.: Building Up and Reasoning About Architectural Knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
17. Navarro, E., Cuesta, C.E.: Automating the trace of architectural design decisions and rationales using a MDD approach. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 114–130. Springer, Heidelberg (2008)
18. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: Dannenberg, R.B., Krueger, C. (eds.) ICOIN 2004 and ICSR 2004. LNCS, vol. 3107, pp. 69–85. Springer, Heidelberg (2004)
19. Capilla, R., Ali Babar, M.: On the role of architectural design decisions in software product line engineering. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 241–255. Springer, Heidelberg (2008)
20. Hallsteeinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. IEEE Computer 41(4), 93–95 (2008)
21. Goedicke, M., Köllmann, C., Zdun, U.: Designing Runtime Variation Points in Product Line Architectures: three cases. Science of Computer Programming 53(3), 353–380 (2004)
22. Lago, P., Muccini, H., van Vliet, H.: A scoped approach to traceability management. Journal of Systems and Software 82(1), 168–182 (2009)
23. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: OOPSLA 2005, pp. 167–176 (2005)
24. Mäder, P., Gotel, O., Philippow, I.: Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 174–189. Springer, Heidelberg (2009)
25. Cleland-Huang, J., Chang, C.: Event-Based Traceability for Managing Evolutionary Change. IEEE Transactions on Software Engineering 29(9) (September 2003)

A Model for Specifying Rationale Using an Architecture Description Language

Lakshitha de Silva[1] and Dharini Balasubramaniam[2]

School of Computer Science, University of St Andrews, St Andrews, KY16 9SX, UK
[1]lakshitha.desilva@acm.org
[2]dharini@cs.st-andrews.ac.uk

Abstract. Besides structural and behavioural properties, rationale plays a crucial role in defining the architecture of a software system. However, unlike other architectural features, rationale often remains unspecified and inaccessible to tools. Existing approaches for recording rationale are not widely adopted. This paper proposes a simple model for capturing rationales as part of an architecture specification and attaching them to elements in the architecture. The bi-directional links between rationales and elements enable forward and backward traceability. We describe a textual architecture description language named Grasp that implements this model, and illustrate its capabilities using an example.

1 Introduction

Software architecture [9,11] establishes a crucial foundation for the systematic development and evolution of software. It provides a high level abstraction of the structure and behaviour of a system in terms of its constituent elements and their interactions. An architecture also reflects *rationale*, the reasoning behind design decisions that guided its creation. Rationale is an intrinsic component of software architecture [9], representing alternatives, trade-offs, assumptions, constraints and others factors considered during its design.

A number of architecture description languages (ADLs) have been developed over the years to formally specify architectures. Most ADLs are conceptually based on the primitives of components, connectors, interfaces and configurations, though wide variations exist in the treatment of these primitives [7]. However, most current ADLs are unable to effectively describe rationale. While a number of techniques have been proposed to capture and represent rationale (e.g. [13,15,10,6]), these have not been extended to ADLs.

The importance of explicitly recording architecture rationale has been widely discussed [9,2,12]. The general consensus is that the tendency to modify software without due consideration to rationale often causes *architecture erosion*. Furthermore, the complex nature of rationale, which is usually a blend of design trade-offs, technical limitations and other constraints, is not completely reflected in the implementation. Therefore, the availability of an effective mechanism to capture rationale from the outset of the design process is imperative to retaining the engineering quality, performance and maintainability of a system.

This paper proposes an approach for specifying rationale as part of an architecture model using the *Grasp* ADL. Grasp allows associating rationale descriptors to elements in the architecture. Both formal expressions and natural

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 319–327, 2011.

language can be used for specifying rationale. An expression in a rationale descriptor may refer to external requirements, quality attributes or elements in the architecture. We describe these concepts with the aid of an example. The paper concludes by outlining the current status and planned future work.

Our work on rationale is part of a larger research agenda for controlling architecture erosion in software systems. We hypothesise that, by maintaining consistency between an architectural specification and its implementation, it is possible to minimise erosion. In order to verify this hypothesis, a simple but expressive model of rationale together with an ADL that can support the specification and evaluation of this model are required.

2 Related Work

Tyree and Akerman [15] use document templates to formally record design decisions. However such techniques face the difficulty of linking documents to other forms of architectural models, which in turn hampers the ability to trace rationale to elements in the architecture using tools. Practitioners may also find it hard to keep documentation up to date with large evolving architectures.

The Archium model [6] describes software architectures as compositions of design decisions, giving due prominence to rationale. However, the effort required to model a conceptual design-decision-oriented architecture and then transform this into a basis for implementation may inhibit industrial adoption.

The Architecture Rationale Element Linkage (AREL) model attempts to capture architecture rationale with traceability [13]. AREL promotes architecture rationale to a first-class entity and establishes relationships between rationale and elements in the architecture. These relationships form a causality chain, providing the basis for backward and forward traceability. The rationale model presented in this paper was largely influenced by AREL. Our work both simplifies and extends the AREL model. It is simpler because alternative rationales are excluded and no distinction is made between qualitative and quantitative rationale. We extend AREL by treating rationale as a statement of "reasons", where a reason can optionally be bound to a system requirement, a quality attribute or an element within the architecture itself.

Zhu and Gorton [16] use UML profiles to model design decisions in an architecture specification. This approach also models non-functional requirements and associates them with architectural elements. Capturing rationale as well as design decisions could extend its usefulness. The relationships between quality attributes and rationales in our model were influenced by this work.

Another UML-based approach proposed by Carignano et al. [4] focuses on the architectural design process and its environment rather than the design outcome. While process and environmental factors may significantly influence architectural design, we believe that rationale should be an inherent part of the design outcome in order to be useful for architecture analysis.

Rationale management systems (e.g. SAURAT [3]) provide tool support for recording, managing and associating rationale with various software artefacts. Although these tools are useful for externally retaining rationale of evolving systems, our work makes rationale an intrinsic part of an architecture specification.

3 Conceptual Model

The proposed rationale model consists of three primary entities. They are *architecture element* (AE), *rationale* and *reason*. An AE has zero or more associated rationales justifying its purpose. A rationale is a conjunction of one or more reasons. A reason is a logical expression that evaluates to a Boolean outcome. If a reason in a rationale evaluates to false, then every dependent AE is considered to have failed its rationale. Any reason expressed in natural language alone is treated as a logically true statement. Figure 1 formalises this model in UML.

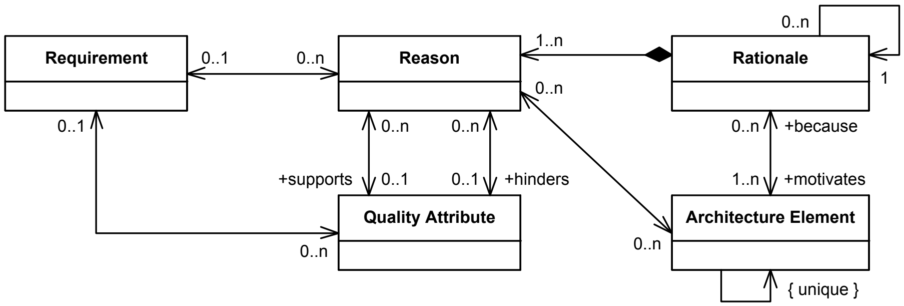


Fig. 1. Model for associating rationale with elements of an architecture

A rationale and an AE have a bi-directional association with each other. A rationale *motivates* the existence or behaviour of one or more AEs. Conversely, an AE can be attached to zero or more rationale entities. Since an AE is justified by a rationale associated with it, the term *because* is used to brand this association. It is noted that a rationale can cross-cut many AEs. Our model supports this possibility as a single rationale can be associated with many AEs.

An AE X can also act as a reason for the rationale of another AE Y since X can cause or require the existence of Y in the same architecture. However, direct or indirect cyclic references are not permitted. Thus, relationships between rationales and AEs form an acyclic graph similar to that of AREL. The $\{unique\}$ constraint applied to the AE entity ensures that each AE instance is unique and therefore does not become a reason for its own rationale.

A rationale may extend another, effectively inheriting its reasons. However, multiple-inheritance is not permitted in the interest of simplicity and clarity.

This model also enables associating *quality attributes* (QAs) as motivating reasons for rationales. A QA is a constraint that should hold when a system implements and delivers its services [14]. Typical drivers for QAs are non-functional requirements. QAs that can be described quantitatively are specified as a collection of properties in this model. Since design choices that support certain QAs may negatively impact others, the association between reasons and QAs are twofold. A reason may *support* one or more QAs. A reason may also *hinder* one or more QAs while in the process of justifying some design decision.

Lastly, reasons and QAs may be attributed to *requirements*, which in this context are references to an external requirement specification. We consider detailed specification of requirements to be outwith the scope of our rationale model.

We note that, in contrast to the rationale model in ISO 402010 [5], our approach does not model architecture decisions. It instead treats an AE as a realised design decision (i.e. a design outcome) justified by some rationale(s).

4 Modelling Rationale with Grasp

Grasp is a textual ADL capable of specifying rationales and associating them with elements of an architecture. It implements the conceptual model presented in Sect. 3. In order to demonstrate architecture specification in Grasp, we reuse the case study of an electronic fund transfer system (EFT) used to illustrate the AREL approach [13]. The result of applying Grasp to a portion of this example, namely the asynchronous messaging subsystem, is shown in Listing 1.

The EFT system is designed to execute high-value online fund transfers between local banks and the central bank in China. A key requirement for the messaging subsystem is performance and hence, the designers chose to use an asynchronous messaging strategy to achieve this critical quality attribute. Subsequent design decisions and outcomes were directly or indirectly influenced by the decision to build an architecture that supports asynchronous messaging.

The AREL approach uses UML profiles to model rationale, AEs and their associations. These UML elements in the AREL example were manually translated into their equivalent Grasp constructs. The next few subsections discuss different aspects of a Grasp model with the help of this example.

4.1 Modelling Rationale

The Grasp specification of the messaging subsystem begins with explicit declarations of QAs and references to external requirements. In the EFT example, Rq_AckProcessing points to an existing requirement which states that all messages should be acknowledged. The two QAs, CommPerformance and CommReliability, define performance and reliability qualities that should be exhibited by the messaging subsystem in terms of properties. Once declared, the two QAs and the requirement become motivating reasons for describing rationale.

The Grasp example declares four rationales named AR10, AR13, AR14 and AR15, each with its own set of reasons. Each rationale is also tagged with a descriptive name using the annotation feature in Grasp. Annotations are name-value pairs useful for providing additional information without altering the semantics of a Grasp construct. Rationale AR10 has two reasons behind it: to achieve CommPerformance and to satisfy Rq_AckProcessing. Grasp uses the keyword *supports* to relate reasons to QAs and requirements. The reason for rationale AR13, on the other hand, is an expression that relates to an AE referenced by parameter M. This expression checks whether the set of properties of M includes property AsyncComm using the *subsetof* operator. Note that the AE passed as parameter M should exist in the namespace of the context in which AR13 is evaluated. The remaining two rationales are declared in a similar manner.

4.2 Modelling System Structure

For specifying the static runtime structure of an architecture, Grasp follows the popular components and connectors paradigm [11,14]. Along with these two primitives, Grasp supports layer, interface, link and check elements as its primary building blocks. An abstract reusable construct called *template* helps to define composite structures from which components and connectors are instantiated.

Listing 1. Grasp specification of a partial EFT system architecture [13]

```
architecture Example
{
    requirement Rq_AckProcessing;

    quality_attribute CommPerformance {
        property HandleMultipleBankConnections = true;
        property MaxVolumePercentageFromOneBank = 50;
        property MinTransactionsPerDay = 8000;
    }

    quality_attribute CommReliability {
        property NoLossPaymentProcessing = true;
        property NoDuplicateProcessing = true;
    }

    @(Desc="OptimalMsgProcPerformance")
    rationale AR10() {
        reason supports CommPerformance;
        reason supports Rq_AckProcessing;
    }

    @(Desc="ProcessingSequence")
    rationale AR13(M) {
        reason [AsyncComm] subsetof M.properties();
    }

    @(Desc="NoLossTransaction")
    rationale AR14(M) {
        reason supports CommReliability;
        reason [AsyncComm] subsetof M.properties();
    }

    @(Desc="TimeOutMechanism")
    rationale AR15(M) {
        reason [AsyncComm] subsetof M.properties();
    }

    template AsyncComponent() {
        property AsyncComm;
    }
    template AsyncMsgProc() extends AsyncComponent {}
    template AsyncMCPDrv() extends AsyncComponent {}
    template AsyncErrDet() extends AsyncComponent {}
    template AsyncErrRec() extends AsyncComponent {}
    template AlarmSvcs() {}

    system PaymentGateway {
        component MsgProc = AsyncMsgProc() because AR10;
        component MCPDrv = AsyncMCPDrv() because AR13(MsgProc);
        component ErrDet = AsyncErrDet() because AR14(MsgProc);
        component ErrRec = AsyncErrRec() because AR14(MsgProc);
        component Alarm = AlarmSvcs() because AR15(ErrDet);
    }
}
```

The chosen example consists of only components. A component roughly maps to the ≪AE≫ stereotyped class in AREL's UML-based model. A base template AsyncComponent is extended by every other template except AlarmServices. The AsyncComm property, defined in AsyncComponent and inherited by all extending templates, identifies those components that implement the asynchronous design.

The *system* block in the Grasp specification contains a number of *component*s that are instantiated from templates. However, the interconnections (i.e. wirings) between these components are not included as this information is missing in the original case study. Grasp provides a *link* primitive to specify the wiring among components and connectors through their interfaces.

4.3 Binding Rationale to Architecture Elements

The next step in building a Grasp specification is to associate rationale to various elements in the architecture. As shown in Listing. 1, each *component* instance has a *because* clause that attaches a previously declared rationale to that component. Some rationales in this example accept arguments that refer to other components within the same namespace. A rationale can be attached in a similar manner to any type of AE including templates. A rationale associated with a template is inherited by a component or connector instantiating that template.

This example also illustrates the dependency graph that forms with AEs and rationales in a Grasp model. For instance, the Alarm component is bound to rationale AR15, which in turn is tied to ErrDetect through its reason. ErrDetect is associated with AR14, which depends on MsgPro, which depends on AR10.

4.4 Evaluating Rationale

A rationale is evaluated in the context of the AE to which it is attached. The application context of a rationale is an important aspect in this model. A rationale cannot be evaluated as a free-standing entity and therefore must be associated with at least one AE for this purpose. At the same time, not all AEs associated with a given rationale may satisfy that rationale. Hence, a rationale is evaluated within the context of each AE it motivates, independent of other associated AEs.

In our example, rationale AR10 is tested within the context of MsgProc, AR13 within the context of MCPDriver, AR14 within both ErrDetect and ErrRecovery, and AR15 within Alarm. In the case of rationale AR14, which attaches to two components, it may possibly pass with one component and fail with the other.

4.5 Traceability

The usefulness of a rationale model largely depends on its ability to trace dependencies between rationale and AEs. Such a model should be able to provide *forward tracing* and *backward tracing* [13]. Forward tracing allows a change to a given rationale or an AE to be traced downwards through the graph to every other rationale and AE affected by that change. Thus it facilitates impact analysis during design modification, offering the means to understand the effects of a change prior to its implementation. Backward tracing assists the discovery of

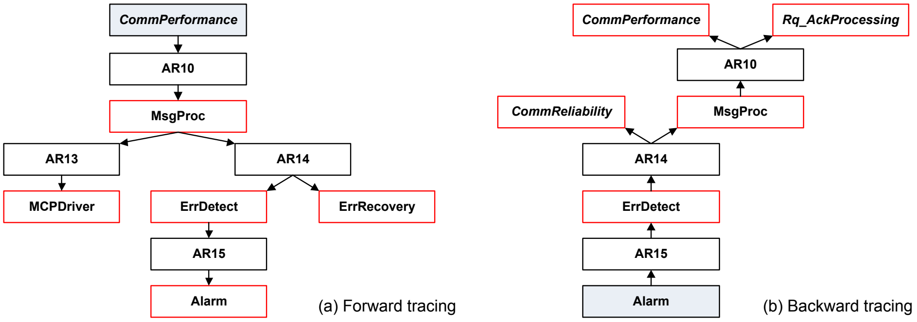


Fig. 2. Traceability in the Grasp model

factors that affect a given AE by tracking upwards through the graph. It helps developers understand the justifications for design outcomes along with sensitive points in the architecture. An AE that traces back to a large number of rationales can be treated as highly sensitive to changes in the architecture.

Figure 2(a) illustrates a forward trace that starts from CommPerformance. This trace will essentially expose the impact on the architecture if any of the performance criteria given in CommPerformance were to change. As shown, rationale AR10 is directly affected, which in turn affects MsgProc. The impact continues to propagate down the dependency graph affecting AR13, AR14 and beyond. As it stands, all five components in the example architecture are affected by a change to the performance quality attribute.

In the complementary backward tracing example in Fig. 2(b), the trace starts from Alarm and moves up towards the root of the graph looking for nodes that have the potential to affect Alarm. This diagram shows that component Alarm is affected by a change to components ErrDetect or MsgProc, QAs CommReliability or CommPerformance, or requirement Rq_AckProcessing. Backward tracing also provides a clear view of the justification behind a design outcome.

5 Implementation Status

The rationale model described in this paper has been defined and incorporated into the Grasp language. A Grasp compiler, developed using the ANTLR parser generator framework [8], is currently available. Work on tools to evaluate rationale, check consistency properties and visualise the architecture is in progress.

6 Conclusions and Future Work

The importance of capturing rationale as part of an architecture has been widely discussed. However, the adoption of rationale techniques has been weak. The Grasp ADL attempts to address this problem with a simple rationale specification mechanism coupled with strong tool support for architectural design.

An extension to the Grasp model to capture alternative architectural decisions and their rationales is also being investigated. An important consideration here is the tradeoff between the usefulness of recording design alternatives and the simplicity of the rationale model essential for encouraging adoption. Furthermore, we are currently extending Grasp to support dynamic architectural properties, enabling the evaluation of architecture rationale during system execution. This work is the first step of a larger agenda to control architecture erosion by maintaining the correspondence between architecture and implementation in all relevant aspects including structure, behaviour and rationale.

Another vital concern is the evaluation of the effectiveness of the rationale model in addressing the above-mentioned issues. Building architecture models of existing software and comparing system evolution with and without the Grasp framework will enable us to carry out this evaluation. Grasp is currently being used to capture the software architectures of constraint solvers as part of an EPSRC-funded project [1]. The architecture specifications from this project will provide some of the required case studies.

It is possible to consider alternative mechanisms, such as separate views of structure, behaviour and rationale of an architecture that are somehow linked to enable traceability among them. Given the need to link architecture and implementation in order to minimise erosion, our approach aims to simplify the process and combine these aspects in a single representation with visualisation tools providing suitable abstractions for users.

Acknowledgment. This work is supported through a PhD studentship awarded by Scottish Informatics and Computer Science Alliance (SICSA) and University of St Andrews.

References

1. Balasubramaniam, D., Silva, L.d., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Dominion: An architecture-driven approach to generating efficient constraint solvers. In: Proc. of the 9th Working IEEE/IFIP Conference on Software Architecture, p. 4 (2011)
2. Bosch, J.: Software architecture: The next step. In: Proc. of the 1st European Workshop on Software Architecture, pp. 194–199 (2003)
3. Burge, J.E., Brown, D.C.: SEURAT: integrated rationale management. In: Proc. of the 30th International Conference on Software Engineering, pp. 835–838 (2008)
4. Carignano, M.C., Gonnet, S., Leone, H.P.: A model to represent architectural design rationale. In: Proc. of WICSA/ECSA 2009, pp. 301–304 (2009)
5. ISO/IEC/IEEE: ISO/IEC 42010: Systems and Software Engineering – Architecture Description. ISO/IEEE (2009), (Draft: ISO/IEC WD4 42010)
6. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proc. of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 109–120 (2005)
7. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
8. Parr, T.: ANTLR Parser Generator (2011), http://www.antlr.org/
9. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)

10. Savolainen, J., Kuusela, J.: Framework for goal driven system design. In: Proc. of the 26th International Computer Software and Applications Conference, pp. 749–756 (2002)
11. Shaw, M., Garlan, D.: Software Architecture: Perspective of an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
12. Tang, A., Babar, M.A., Gorton, I., Han, J.: A survey of architecture design rationale. Journal of Systems and Software 79(12), 1792–1804 (2006)
13. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
14. Taylor, R., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice. Wiley, Chichester (2009)
15. Tyree, J., Akerman, A.: Architecture decisions: Demystifying architecture. IEEE Software 22(2), 19–27 (2005)
16. Zhu, L., Gorton, I.: UML profiles for design decisions and non-functional requirements. In: Proc. of the 2nd Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent., p. 8 (2007)

From EAST-ADL to AUTOSAR Software Architecture: A Mapping Scheme

Tahir Naseer Qureshi[1], DeJiu Chen[1], Henrik Lönn[2], and Martin Törngren[1]

[1] Department of Machine Design. The Royal Institute of Technology,
Stockholm, Sweden
{tnqu,chen,martin@md.kth.se}
[2] Volvo Technology Corporation, Electronics and Software,
SE-405 08 Gothenburg, Sweden
henrik.lonn@volvo.com

Abstract. This paper addresses the gap between models describing system requirements, functions and architecture at a higher level of abstraction (such as SysML models), with respect to software/hardware architecture description (such as the AADL models) as the means to improve the development process or embedded systems. The EAST-ADL and AUTOSAR are the two focused architecture description formalisms in the presented work. While EAST-ADL is an architecture description language providing an extension and profiling of SysML dedicated to automotive embedded systems, AUTOSAR provides means to describe software architecture architectures. The contribution of the paper is a relationship investigation between different concepts of the two languages. Three case studies, of a position control , fuel control and a brake-by-wire system, have been used to support and validate the work. The resulting mapping scheme provides a basis for automated architecture refinements and synthesis.

Keywords: AUTOSAR, Model-based Development, EAST-ADL, Embedded Systems, Methodology, Architecture Description Language, Brake-by-wire System, Model Transformation, Matlab, Simulink, SystemDesk, TargetLink.

1 Introduction

Embedded systems development complexity has increased considerably during the last few decades. This complexity spans over three dimensions i.e. product (features and interactions of components), organization (companies and development teams) and technology (tools and process) [19]. Some of the common requirements to manage the complexities include separation of views like hardware and software and consideration of both functional and non-functional properties. Formalisms such as SysML[1] (Systems Modeling Language) and AADL[2] (Architecture Analysis & Design

[1] http://www.omgsysml.org/#Specification

[2] http://www.aadl.info/, http://standards.sae.org/as5506a

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 328–335, 2011.

Language) address these requirements and challenges by providing means for specifying embedded system architecture. These formalisms often target different abstraction levels and for most of the cases have a weak or no integration with each other. This is one of the challenges in industrial adaptation of such formalisms for improving the overall development efficiency for embedded systems. The solutions for this challenging problem can include but is not limited to an automated tool support for architecture refinement and synthesis and methodological guidelines.

We have addressed the above mentioned need by considering two different formalisms for architectural specifications i.e. EAST-ADL [1, 8] and AUTOSAR [2] for automotive embedded system from control systems development viewpoint. EAST-ADL shares the same core meta-constructs and complements AUTOSAR with additional levels of abstractions and concepts such as requirements engineering and safety [1]. AUTOSAR models thus represent an implementation of an architecture specified by EAST-ADL. Further, EAST-ADL constructs can be used to annotate AUTOSAR elements with requirements, error models, etc. To efficiently and correctly define an AUTOSAR implementation based on EAST-ADL model, it is useful to have guidelines, mapping patterns and tool support. We consider the following needs:

- A detailed investigation of the mapping between the artefacts for enabling transformation of architectural specification between these formalisms.
- An investigation of relation between the EAST-ADL and AUTOSAR methodologies from behavioural specification and implementation view.
- An investigation of tool support for formal guidelines and possible integration scenarios to utilize the two formalisms and their methodologies.

This paper presents a part of work towards a well-defined path from an EAST-ADL functional architecture to a concrete AUTOSAR software architecture. The main contribution presented in this paper is a mapping scheme between the artefacts of the two formalisms related to the software architecture part of an embedded system. This mapping can be considered as a step towards the automatic generation of an AUTOSAR compliant architecture specification from EAST-ADL. Due to large span of the two languages, we have limited our work to behavioural aspects and associated timing constraints from control systems development view. Three case studies of a brake-by-wire, position control and a fuel control system are used as case studies to support the work.

It is assumed that the readers have some knowledge of AUTOSAR, its layered software architecture [6] and methodology [7] in addition to EAST-ADL, its methodology [9] and behaviour extension [10]. The readers are also referred to [11] for an overview of architecture description languages, associated problems and challenges related to the development of control systems for automotive systems. It is also recommended to refer to [18] for a detailed description including additional investigations and results of the presented work if desired.

The paper is organized as follows: In the following section, an overview of the related work is presented. This is followed by a discussion on the relationship work, observations and a brief account of the case studies in section 3. The paper is concluded with a discussion in section 4.

2 Related Work

A few efforts have been made to develop AUTOSAR system specification from different ADLs. In [13] a mapping between different EAST-ADL artefacts and AUTOSAR is presented for both structure and behaviour. However, the mapping focuses on only a few structural entities and a few events. Furthermore, due to the changes in the language regarding the inclusion of concepts such as mode, restructuring for modularity and renaming of a few artefacts, some parts of the mapping are no longer valid. Our work aims to provide a more detailed and refined mapping scheme with the current version of EAST-ADL.

In the EDONA project [14] an Eclipse based platform is developed for the integration of different tools used in the automotive industry. The approach is based on an AUTOSAR meta-model developed using EMF (Eclipse Modeling Framework). The ARGateway (AUTOSAR Gateway) is used as a means for model transformation from EAST-ADL design architecture model to an AUTOSAR model. The gateway is mainly based on the mapping provided in [13]. Therefore, it needs to be updated. The required updates can utilize the results of the work presented in this paper.

In [15] bidirectional transformations between SysML and AUTOSAR using graph transformations are presented. The transformation uses a SysML profile covering basic embedded system entities such as hardware, software, port etc. and an AUTOSAR meta-model. SystemDesk (an AUTOSAR modelling tool by dSpace [5]) API is used to transfer information to and from SystemDesk. The SysML profile in [15] is too generic and cover only a few automotive systems aspect as compared to EAST-ADL. SystemDesk is the common tool in [15] and our work. Therefore, a tool specific transformation can be performed by utilizing the results of the presented work and the SystemDesk API part from [15].

In the TIMMO [16] project, a language for timing design called TADL (Timing Augmented Description Language) is developed. The language and its methodology [17] is aligned with EAST-ADL [9] and AUTOSAR [7]. Especially the timing aspects of AUTOSAR are actually fully harmonized with the TIMMO concepts. As the name indicates, the TIMMO methodology is focused on timing analysis aspects. In contrast to the TIMMO effort focused on timing aspects, our effort focuses the behavioural aspects.

3 EAST-ADL and AUTOSAR Relationship Investigation

A two-step approach is adopted to meet our objectives. First, a position control systems is modelled in EAST-ADL using PapyrusUML modeller [4] for basic structural and behavioural artefacts. This is followed by repeating the same procedure for a fuel control system for additional behavioural aspects especially mode related behaviour. As a second step, a brake-by-wire (BBW) system provided by Volvo Technology is modelled in SystemDesk to validate the results (i.e. refined mapping scheme between EAST-ADL and AUTOSAR) from the first step. The fuel control and position control systems are example cases from dSpace [12] providing coverage of AUTOSAR artefacts sufficient to meet our objectives. Interested readers can also refer to [18] for more information about the case studies.

3.1 Functional and Behavioural Mapping

Tables 1 and 2 summarize the mapping between the functional and behavioural entities of EAST-ADL and AUTOSAR i.e. which AUTOSAR elements typically realize an EAST-ADL element. The tables only describe the mapping scheme. For information about the semantics, the readers are referred to EAST-ADL and AUTOSAR specifications.

Table 1. A functional (structural) mapping between EAST-ADL and AUTOSAR

EAST-ADL	AUTOSAR	Remarks
FunctionalDesignArchitecture	Software architecture	It is assumed that a design function type with name "FunctionalDesignArchitecture" is the top most function in the hierarchy of design function types and prototypes.
FunctionModeling::DesignFunctionType	Runnable and Atomic Software Component	The property isElementary determines if a design function prototype (or several) is conveniently realized by a runnable. An atomic software component contains at least one runnable through its internal behaviour definition. isElementary = true implies a single runnable can be used. isElemantary = false implies an atomic software component with one or more Runnables in its InternalBehavior or a composite component.
FunctionModeling::BasicSoftwareFunctionType	Basic Software Component	A basic software function type is a specialization of DesignFunctionType and a middleware abstraction. This corresponds to basic software component in AUTOSAR
FunctionModeling::LocalDeviceManager	Sensor Actuator Software Component	The LocalDeviceManager encapsulates the device-specific or functional parts of a Sensor or Actuator, device, interface etc.
FunctionModeling::FunctionFlowPort Direction={IN, OUT}	A port with a Sender/Receiver interface or an interrunable variable.	The direction of the EAST-ADL flow port determines if the corresponding AUTOSAR port has a provided or required interface. *direction = OUT* corresponds to provided port *direction =IN* corresponds to a required port. If the associated EAST-ADL DesignFunctionType is realized as an AUTOSAR runnable then the port is realized as an inter-runnable variable.
FunctionModeling::FunctionClientServerPort ClientServerType ={client, server}	A port with Client or Server Interface	The role as a client or server is defined by the property ClientServerType. This corresponds to a client and server interface respectively in AUTOSAR
FunctionModeling::ClientServerInterface	Client-Server Interface	Same Concept
FunctionModeling::Operation	Operation	Same concept and related to a client-server interface.

Table 2. Mapping of EAST-ADL artefacts corresponding to AUTOSAR behaviour

EAST-ADL	AUTOSAR	Description
Behavior::Mode	Mode	The concept is similar in both formalisms. A mode control can lead to the switches between different configurations or execution schemes
Behavior::ModeGroup	Mode Group	Same concept in both formalisms to organize a set of modes in a mutually exclusive group.
Behavior::FunctionTrigger TriggerPolicy={TIME, EVENT}	RTE Event	The execution behaviour of an EAST-ADL function is declared by function triggers. The type of corresponding AUTOSAR RTE Event is determined by the TriggerPolicy and the associated Event Function and constraint for the function trigger. TriggerPolicy=Time implies a periodic event. (Timing event in AUTOSAR) TriggerPolicy=Event implies other events.

Timing::EventFunctionClientServerPort
EventKind={sentRequest, receivedResponse, receivedRequest, sentResponse}
	Operation Invoked Event and Asynchronous Server Call Returns Event	An event function is associated with a function type in EAST-ADL. The type of event function determines the type of RTE Event in AUTOSAR.
		eventKind = receiveRequest implies Operation Invoked Event
		eventKind = receivedResponse implies Asynchronous Call Returns Event

Timing::EventFunctionFlowPort	Data Received Event	The port considered in this kind of event should have the value 'IN' for its 'Direction' property.

An illustration of the mapping is shown in Figure 1.

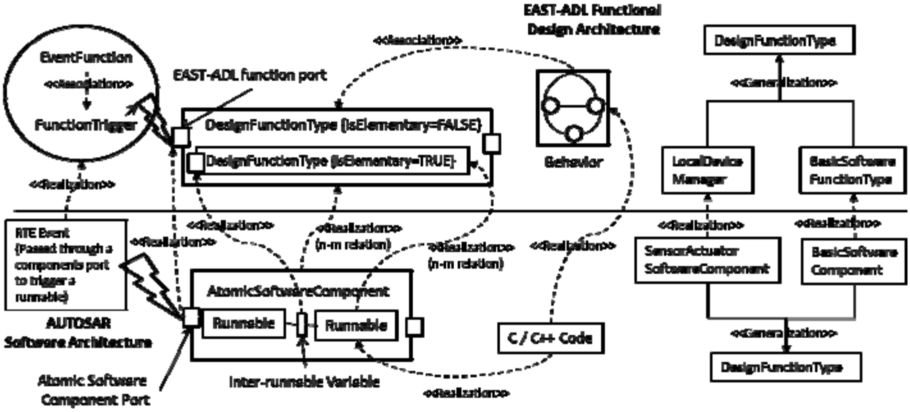


Fig. 1. An illustration of EAST-ADL and AUTOSAR mapping

3.2 Additional Observations

The following text will throw light on the factors which may affect the decisions required for refinement of software architecture from EAST-ADL to AUTOSAR.

Mode Switch Port and Event

Modes are declarative in EAST-ADL. Mode change and related communications e.g. mode switch event are not part of the EAST-ADL specifications. To handle this we can assume an EAST-ADL FunctionFlowPort to be equivalent to a mode switch port provided that it is referred in the function trigger properties in addition to the event condition for activation.

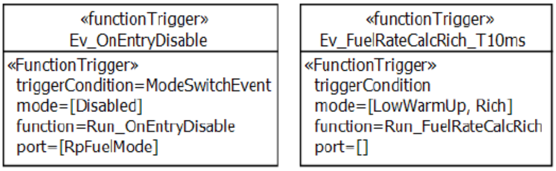


Fig. 2. A function trigger illustration in EAST-ADL

An illustration is shown in the left part of Figure 2 for a runnable which is activated on the *entry* of the mode called ‘Disabled’[3]. The policy selected for this trigger is 'EVENT'. The right side of Figure 2 refers to a periodic trigger with the value 'TIME' selected for the property 'TriggerPolicy'. Here the mode property specifies the modes during which this trigger is active.

Data Types and Prototypes

EAST-ADL only supports basic data types including Boolean, float, integer, string as well as composite data types. As a part of defining an AUTOSAR software architecture, these abstract data types are required to be mapped to concrete implementation data types with signedness, number of bits, coding, etc. Furthermore, in contrast with AUTOSAR, only one data type can be assigned to a single port in EAST-ADL. This is handled by the use of port groups. Depending on the type of realization i.e. an atomic software component or a runnable, a port specifies an inter (RTE) or intra (Variables) software interaction. This is described in Table 1 and illustrated in Figure 1. It should also be noted that there is a DataElementPrototype for every EAST-ADL port. The data types should have matching specifications. Several EAST-ADL ports may be aggregated in a single AUTOSAR interface. EAST-ADL port groups are candidates for aggregation depending on connection patterns.

Design Function Type Realization

For simplicity we proposed a one-to-one mapping between a design function type of EAST-ADL and an AUTOSAR runnable or software component. However, it is a n-to-m mapping as shown in Figure 1. This implies that several design function types can be realized by one runnable or vice versa. The same applies for the realization of design function type by AUTOSAR atomic software components.

[3] The string `ModeSwitchEvent' for the trigger condition in the figure is for illustration purpose only. This means that other constructs e.g. OCL (Object Constraint Language) can also be used depending on the user choice.

4 Discussion and Conclusion

We have investigated the support of EAST-ADL for AUTOSAR based system definition as a means to capture the early phase information and provide abstract representation of system. The main result of the work is verification and refinement of the existing relationship between the two formalisms. The work mainly considered the structural, behavioural aspects and related timing constraints. Although the work is performed using specific tool chain i.e. PapyrusUML modeller and SystemDesk, the mapping scheme is generic addressing both the updated and new concepts of EAST-ADL, therefore, it can be utilized for any AUTOSAR and EAST-ADL tool. For example, the results can be used directly in updating the ARGateway effort carried out in the EDONA project [14].

Some of the issues identified during the work which require consideration are the semantic gaps between these two formalisms. One such gap is the semantics related to events. While EAST-ADL events are introduced to specify timing constraints, AUTOSAR events are used only for triggering of a runnable. Further investigations for mode switching and related communication are also required for improving EAST-ADL support for behavioural aspects. An extension of EAST-ADL with few additional artefacts e.g. explicit initial mode can also be considered.

In addition to the mapping scheme presented in this paper, we have also performed an initial investigation on the relationship between EAST-ADL and AUTOSAR for tool support and methodological issues [18]. From methodological perspective, EAST-ADL is an add-on to the current development process providing early verification and validation resulting in reduction of the integration problems. An automated tool is found to be a necessity to improve the overall development process. This automated support can be for providing traceability support, integration of tools for analysis such as the one described in [3].

The presented work can be extended by providing the required tool support for automated model transformations between EAST-ADL and AUTOSAR tools. The possibility of the mapping of runnables to OS (Operating System) tasks and relation with other analysis tool for the type of analyses not currently available for architecture specified in EAST-ADL needs to be addressed in future. In addition to the presented work which focuses on the behavioural aspects or the TIMMO effort [17] for timing related properties, a detailed methodology taking into account all the available aspects e.g. behaviour, timing, safety and variability is also required.

References

1. EAST-ADL Consortium Website, http://www.atesst.org (accessed January 2011)
2. AUTOSAR Website, http://www.autosar.org/ (accessed January 2011)
3. Biehl, M., Sjöstedt, C.-J., Törngren, M.: A Modular Tool Integration Approach - Experiences from two Case Studies. In: 3rd Workshop on Model-Driven Tool & Process Integration at the European Conference on Modelling Foundations and Applications (June 2010)
4. PapyrusUML Website, http://www.papyrusuml.org (accessed January 2011)

5. dSpace GmbH Website, http://www.dspaceinc.com (accessed January 2011)
6. The AUTOSAR Consortium, Layered Software Architecture, Tech. Rep. V2.2.1, R3.0, Rev 001 (2008)
7. The AUTOSAR Consortium, AUTOSAR Methodology, Tech. Rep. V1.2.1, R3.0, Rev 001, http://www.autosar.org/download/AUTOSAR_Methodology.pdf (2008)
8. The ATESST2 Consortium, EAST-ADL Domain Model Specification, Project Deliverable 4.1.1, http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf (June 2010)
9. The ATESST2 Consortium, "Methodology Guidelines When Using EASTADL2," Project Deliverable 5.1.1, http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D5.1.1_V1.1.pdf (June 2010)
10. The ATESST2 Consortium, Update Suggestions for Behavior Support, Project Deliverable 3.1, Appendix A3.4, http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D3.1_A3.4_V1.1.pdf (June 2010)
11. Lönn, H., Freund, U.: Automotive Architecture Description Languages. In: Automotive Embedded Systems Handbook (2009)
12. dSpace GmbH, dSpace HelpDesk (2009) (Available online for the dSpace Software Users)
13. Cuenot, P., Frey, P., Johansson, R., Lönn, H., Reicser, M.O., Servat, D., Kolagari, R.T., Chen, D.: Developing Automotive Products Using the EAST-ADL2, an AUTOSAR Compliant Architecture Description Language. In: Ingniurs de lAutomobile, vol. 793, p. 58 (2008)
14. Environnements de Dveloppement Ouverts aux Normes de l'Automobile (EDONA) Website, http://www.edona.fr (accessed January 2011)
15. Giese, H., Hildebrandt, S., Neumann, S.: Towards "Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization. In: 5th Workshop on Model-Based of Embedded Systems (MBEES)
16. TIMMO Consortium Website, http://www.timmo.org (accessed January 2011)
17. The TIMMO Consortium, Methodology Version 2, Project Deliverable 7 (2009), http://www.timmo.org/pdf/D7_TIMMO_Methodology_Version_2_v10.pdf
18. Qureshi, T.N., Chen, D., Lönn, H., Törngren, M.: From EAST-ADL to AUTOSAR, Technical Report KTH-TRITA-MMK 2011:12, ISSN 1400-1179, ISRN KTH/MMK/R-11/12-SE
19. Törngren, M., Chen, D., Malvious, D., Axelsson, J.: Model-Based Development of Automotive Embedded Systems. In: Automotive Embedded Systems Handbook (2009)

Software Language Engineering of Architectural Viewpoints

Elif Demirli and Bedir Tekinerdogan

Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey
{demirli,bedir}@cs.bilkent.edu.tr

Abstract. A common practice in software architecture design is to apply architectural views to design software architecture for the various stakeholder concerns. Architectural views are usually developed based on architectural viewpoints which define the conventions for constructing, interpreting and analyzing views. So far most architectural viewpoints seem to have been primarily used either to support the communication among stakeholders, or at the best to provide a blueprint for the detailed design. In this paper we provide a software language engineering approach to define viewpoints as domain specific languages. This enhances the formal precision of architectural viewpoints and leads to executable views that can be interpreted and analyzed by tools. We illustrate our approach for defining domain specific languages for the viewpoints of the Views and Beyond approach.

Keywords: Architectural Viewpoints, Software Language Engineering, Domain Specific Modeling, Tool Support.

1 Introduction

An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. Architectural views conform to viewpoints that represent the conventions for constructing and using a view. An architectural framework organizes and structures the proposed architectural viewpoints. Different architectural frameworks have been proposed in the literature [2]. Organizing the system as a set of viewpoints has also been addressed in enterprise application system using so-called enterprise architecture frameworks [12][13]. The notion of viewpoint now plays an important role in modeling and documenting architectures. So far most architectural viewpoints seem to have been primarily used either to support the communication among stakeholders, or at the best to provide a blueprint for the detailed design. From a historical perspective it can be observed that viewpoints defined later are more precise and consistent than the earlier approaches but a close analysis shows that even existing viewpoints lack some precision. Moreover, since existing frameworks provide mechanisms to add new viewpoints the risk of introducing imprecise viewpoints is high. The development of a proper and effective architecture is highly dependent on the corresponding documentation. An

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 336–343, 2011.

incomplete or imprecise viewpoint will impede the understanding and application of the viewpoints to derive the corresponding architectural views, and likewise lower the quality of the architectural document.

The key premise in this paper is that a viewpoint can be considered as a domain specific language, and views are models or programs of that language. As such, to enhance the definition of the viewpoints we think that these should be also formally defined as domain specific languages. In this paper we provide a software language engineering approach to define viewpoints as domain specific languages. This will enhance the formal precision of architectural viewpoints and likewise helps to share the additional benefits of domain specific languages, i.e. defining executable views. In the paper, we illustrate our approach using an example viewpoint: decomposition viewpoint of Views and Beyond (V&B) [2] approach.

The remainder of the paper is organized as follows. In section 2 we define the background of architecture framework and software language engineering. In section 3, we show the definition of domain specific language for decomposition viewpoint of the V&B approach. Section 4 presents the related work. Section 5 provides the conclusions.

2 Model-Driven Development

Architecture design is basically about *modeling* the system from different perspectives. Historically, *models* have had a long tradition in software engineering and have been widely used in software projects. The primary reason for modeling is usually defined as a means for communication, analysis or guiding the production process. Models are different in nature and quality. Mellor et al. [9] make a distinction between three kinds of models, depending on their level of precision. A model can be considered as a *Sketch*, as a *Blueprint*, or as an *Executable*. According to [9] an executable model is a model that has everything required to produce the desired functionality of a single domain. Executable models are more precise than sketches or blueprints, and can be interpreted by model compilers.

In model-driven software development the concept of *models* can be considered as executable models as defined by the above characterization of Mellor et al. [9]. This is in contrast to model-based software development in which models are used as blueprints at the most.

The language in which models are expressed is defined by meta-models. As such, a model is said to be an instance of a meta-model, or a model *conforms to* a meta-model. A meta-model itself is a model that conforms to a meta-meta-model, the language for defining meta-models. In model-driven development, models are usually organized in a four-layered architecture. The top (M3) level in this model is the so called meta-metamodel, and defines the basic concepts from which specific meta-models are created at the meta (M2) level. Normal user models are regarded as residing at the M1 level, whereas real world concepts reside at level M0.

2.1 Architectural Description from a Model-Driven Development Perspective

In fact we can state that the current architectural modeling practices can be categorized as *model-based development*, rather than *model-driven development.* In the last two to three decades architectural modeling and the corresponding notations have evolved from simple sketches to more precise models as defined by architectural view concept. Yet, the view models can usually not be considered as executable models. Moreover, the link between architectural models, and the link from architectural models are merely implicit and not formal.

In architecture modeling literature the notion of meta-model is not explicitly used. The concepts related to architectural description are formalized and standardized in ISO/IEC 42010:2011 [7]. The standard holds that an architecture description consists of a set of *views*, each of which conforms to a *viewpoint*. Here the concept of view appears to be at the same level of to the concept of *model* in the model-driven development approach. The concept of viewpoint, representing the language for expressing views, appears to be on the level of meta-model.

Although the ISO/IEC 42010 standard does not explicitly use the terminology of model-driven development the concepts as described in the standard seem to align with the concepts in the meta-modeling framework. In Fig. 1, we provide a partial view of the standard that has been organized around the meta-modeling framework. An *Architecture Description* is a concrete artifact that documents the *Architecture* of a *System of Interest*. The concepts *System-of-Interest* and *Architecture* reside at layer M0. *System-of-Interest* defines a system for which an *Architecture* is defined. *Architecture* is described using *Architectural Description* that resides at level M1. *Architectural Description* includes one or more *Architectural Views* that represent the system from particular stakeholder concern's perspective. Architectural views are described based on *Architectural Viewpoint*, the language for the corresponding view. *Architectural Viewpoints* are organized in *Architectural Framework.* The latter two reside at level M2. The standard does not provide a concept that we could consider at level M3, and as such we have omitted this in Fig. 1.

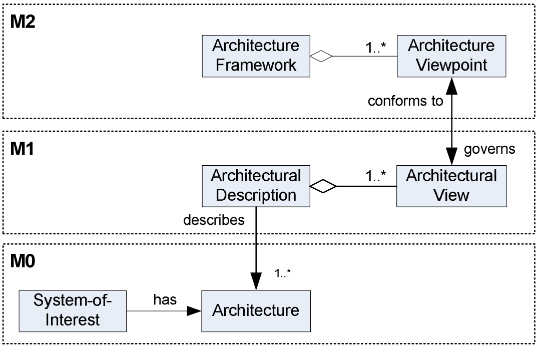


Fig. 1. Architectural Description Concepts from a meta-modeling perspective

2.2 Elements of Domain Specific Languages

Meta-models define the language for the models. The application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of these languages is usually called *software language engineering* [8]. A proper definition of meta-models is important to enable valid and sound models. In both the software language engineering [8] and model-driven development domains [9], a meta-model should include the following elements:

- *Abstract Syntax*: the vocabulary of concepts provided by the language and how they may be combined to create models.
- *Concrete Syntax*: the notation that facilitates the presentation and construction of models or programs in the language. It can be visual or textual.
- *Well-formedness rules (Static Semantics)*: definitions of additional constraint rules on abstract syntax that are hard or impossible to express in standard syntactic formalisms of the abstract syntax.
- *Semantics*: the definition of the meaning of the concepts in the abstract syntax.

Given these elements of a language we can also evaluate viewpoints, the languages for defining views. A coarse-grained evaluation would be to check whether these elements are defined for the viewpoints. This does not really provide much information since all the viewpoints seem to somehow describe the above elements albeit in a different degree. To be able to define the degree to which each element is addressed we propose the evaluation framework as defined in Table 1. The table distinguishes among four levels L1 to L4 indicating the quality and completeness of the element. As it can be seen in the table, a lower quality indicates that the corresponding element has not been described (missing, not defined) whereas a higher value indicates that the given element is completely defined and validated.

Table 1. Assessment framework for evaluating Architectural Viewpoints

	L1	L2	L3	L4
Abstract Syntax	Missing or vague	Clear textual description	Meta-model defined. Non-validated models	Validated models
Concrete Syntax	Not defined	Informal	Semi-Formal	Formal
Static Semantics	Not defined	Incomplete constraints in natural language	Complete constraints in natural language	Formal, Executable constraints

3 Defining Viewpoints as Domain Specific Languages

In this section we will illustrate the modeling of viewpoints as domain specific languages to show how existing viewpoints can be even further formally specified to lift these to the level of executable models. We have chosen the decomposition style of the V&B framework [2], as example viewpoint. We will follow the process as defined in the previous section. For the DSL, we first present the abstract syntax that defines the language abstractions and their relationship. The abstract syntax is defined after an analysis of the viewpoint description in the corresponding textbook [2].

Based on these descriptions and the defined meta-model we provide the grammar which defines syntactic rules of the language together with textual concrete syntax.

The grammar is defined using Xtext a language development framework provided as an Eclipse plug-in [4]. The grammar of the language is defined in Xtext's EBNF grammar language and the corresponding generator creates a parser, an AST-meta model as well as a full-featured Eclipse Text Editor from that. The visual concrete syntax is defined using Graphical Modeling Framework (GMF) plug-in of Eclipse [4]. Constraints on viewpoint elements and relations are implemented as static semantics which is implemented writing validation codes in Java. We consider only the elements as defined in Table 1 and do not consider the discussion on semantics. After presenting the language for decomposition viewpoint, a short discussion of the viewpoint specification with respect to our evaluation framework is provided.

3.1 Decomposition Style

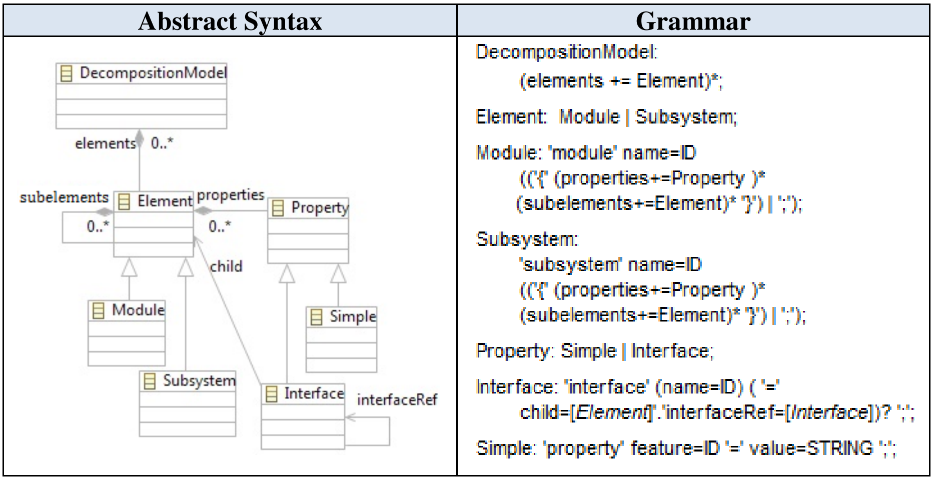
Based on these descriptions and the defined meta-model we provide the grammar which defines syntactic rules of the language together with textual concrete syntax. The *Decomposition style* is used to show how system responsibilities are partitioned across modules and how these modules are decomposed into submodules. The decomposition view of the architecture depicts the overall structure of the architecture which is reasonably decomposed into modular implementation units. It is regarded as a fundamental view of the architecture since it serves as an input for other views (e.g. work allocation view) and helps to communicate and learn the structure of the software. We have defined a DSL for decomposition style based on the textual specification given in [2]. The meta-model elements of this style are provided below.

3.1.1 Abstract Syntax

A model of the abstract syntax for the decomposition style is given in the left part of Fig. 2. The root element is DecompositionModel. A valid decomposition model consists of Elements. An element can either be a Module or Subsystem. Module denotes principal unit of implementation. Subsystem differs semantically from the module in the way that it can be developed, executed and deployed independent of other system parts. The decomposition relation between elements is established via the aggregation relation indicating that an element consists of other subelements. Element can have two types of properties: Interface and Simple property. The element’s interface is documented with interface property. An element’s interface can be declared as a reference to one of its children’s interface. Simple property is a generic property which allows specifying new properties in view document.

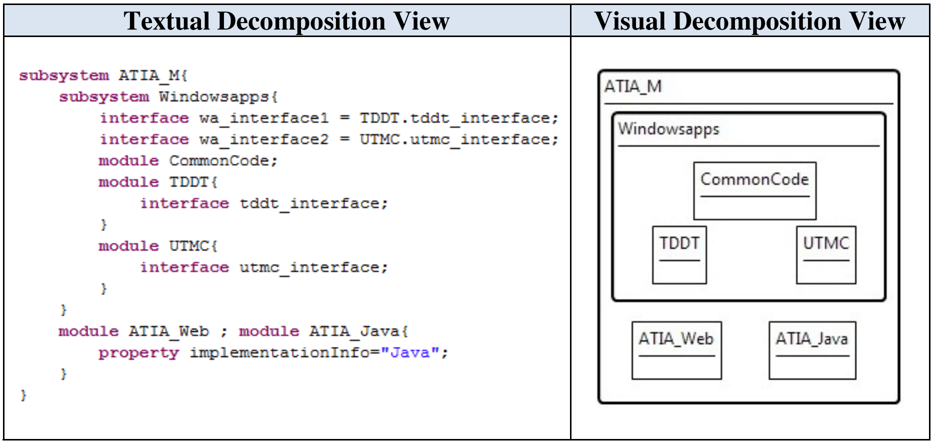
3.1.2 Grammar and Concrete Syntax

The grammar for decomposition style is given in the right part of Fig. 2. An example decomposition view implemented using our DSL is shown in Fig. 3. The textual concrete syntax is defined for both elements and properties of the elements. The visual concrete syntax is defined only for elements. No explicit relation is modeled in order to express decomposition. Subelements are directly placed into the parent element.



```
DecompositionModel:
      (elements += Element)*;
Element:  Module | Subsystem;
Module: 'module' name=ID
      (('{' (properties+=Property )*
      (subelements+=Element)* '}') | ';');
Subsystem:
      'subsystem' name=ID
      (('{' (properties+=Property )*
      (subelements+=Element)* '}') | ';');
Property: Simple | Interface;
Interface: 'interface' (name=ID) ( '='
      child=[Element]'.'interfaceRef=[Interface])? ';';
Simple: 'property' feature=ID '=' value=STRING ';';
```

Fig. 2. Abstract Syntax and Grammar for Decomposition Style



```
subsystem ATIA_M{
    subsystem Windowsapps{
        interface wa_interface1 = TDDT.tddt_interface;
        interface wa_interface2 = UTMC.utmc_interface;
        module CommonCode;
        module TDDT{
            interface tddt_interface;
        }
        module UTMC{
            interface utmc_interface;
        }
    }
    module ATIA_Web ; module ATIA_Java{
        property implementationInfo="Java";
    }
}
```

Fig. 3. Example decomposition view with textual and visual concrete syntax

3.1.3 Static Semantics

In addition to extracting the abstract syntax and the grammar we can also derive the well-formedness rules of views, the static semantics, from the viewpoint descriptions. In the decomposition style, two constraints have been defined: no loops are allowed in decomposition graph and a module can have only one parent. From the language perspective, those constraints are too high level to implement. We merged these constraints and shortly defined that no element can have the same name. Doing so we prevented both <A contains B, B contains A> case and <A contains B, C contains B> case. We have implemented this constraint in Java as a validation rule that applies on the language model.

3.1.4 Evaluation

The above results show that we could map a viewpoint to a domain specific language that can be used to define executable models or views. However, the overall effort also provides us insight in the degree of formal precision of the current viewpoint description. When we apply our evaluation framework on decomposition style specification of V&B framework, we get the following results. The abstract syntax definition falls into L2 of our evaluation framework. The concepts to be used in the language are defined textually. The textual description is clear; it can be easily translated to a formal model. However, no meta-model or grammar is provided to describe the concepts. Since both informal and semiformal notations are provided the concrete syntax definition can be considered at level L3. Finally, the well-formedness rules on the concepts of the language are properly specified in natural language. However, they are too high level to directly implement as executable well-formedness rules. Therefore, we consider these at level L3. It should be noted that with the domain specific language engineering approach we have lifted the precision degree to level L4.

4 Related Work

In the enterprise architecture (EA) design community several authors have focused on the formalization of architectural viewpoints. Different attempts have been made before to model viewpoints as domain specific languages. ArchiMate [1] is an EA modeling language that is specified by concepts that focus on business, applications and technology domains. Those concepts form the base metamodel of ArchiMate language. A set of viewpoint languages are defined by composing the concepts available in the metamodel. Contrary to their approach, our viewpoint languages do not depend on a predefined set of concepts. Each viewpoint has an independent language that defines its own concepts. This design choice makes it easy to introduce new viewpoints to the framework. However, it is difficult to define new viewpoints in ArchiMate if the required concepts are not available at the base metamodel. An additional extension mechanism is needed for this purpose [10].

Another example to attempts on formalizing EA viewpoints is about RM-ODP viewpoints. Vallecillo et al. initially focused on formally specifying the abstract languages provided by viewpoint specifications using a rewriting logic based framework Maude [3]. Later on, they also tackle the viewpoint formalization problem from model-driven development perspective and defined UML profile for viewpoints of RM-ODP [11]. Lastly, they define textual notation for ODP specifications together with tool support [5]. The main difference of their approach and our study is the level of formality of the targeted viewpoint specifications. RM-ODP is specified by a standard [6] that precisely defines the syntax and semantics of the language. So, the task of formalizing RM-ODP viewpoint specifications is transforming the present languages to executable languages and defining notations for using the language. However, in our work, we also address viewpoint specifications those are not specified precisely as languages. We offer software language engineering as a method for lifting existing viewpoint specifications to formal language level and provide a complete description of the method.

5 Conclusions

In this paper, we have illustrated the adoption of software language engineering approach for modeling architectural viewpoints. The key premise behind this assumption is that viewpoints are in fact domain specific languages, and as such should be considered and developed like that. To validate our statement we have analyzed the viewpoints in the Views and Beyond approach [2], and defined all these viewpoints as domain specific languages. In the paper, as an example, we have presented the definition of decomposition viewpoint DSL.

We believe that by adopting a software language engineering approach for architectural viewpoints we have also shown the connection with software architecture design modeling and the fields of software language engineering and model-driven software development in general. We hope that this work has paved the way for further research in this direction.

In our future work we will apply the same approach to other architecture viewpoint frameworks. The V&B approach was a case study for us but we do not foresee serious obstacles in applying the same approach for other software architecture viewpoints and enterprise architecture viewpoints. We will elaborate on the tool and consider the integration of viewpoints for nonfunctional concerns. Further, we plan to enhance the tool for supporting architectural analysis.

References

[1] Archimate 1.0 Specification, The Open Group, Tech. Rep. C091 (February 2009)

[2] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Reading (2010)

[3] Durán, F., Vallecillo, A.: Formalizing ODP Enterprise specifications in Maude. Computer Standards & Interfaces 25(2), 83–102 (2003)

[4] Eclipse Modeling Framework Web Site, http://www.eclipse.org/emf/ (accessed on June 2011)

[5] González, D.R., Vallecillo, A., Romero, J.R.: On the Synchronization of ODP Textual and Graphical Specifications. In: Proc. of WODPEC 2010, Vitoria, Brazil, October 25, pp. 376–381 (2010)

[6] [ISO/IEC 10746-2:1996] International Organization for Standardization & International Electrotechnical Commission. Information Technology - Open Distributed Processing - Reference Model: Foundations (ISO/IEC 10746-2) (1996)

[7] [ISO/IEC 42010:2011] Systems and Software Engineering – Architecture Description (ISO/IEC 42010) (2011)

[8] Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman Publishing Co., Inc., Boston (2009)

[9] Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principle of Model Driven Architecture. Addison Wesley, Reading (2004)

[10] Peña, C., Villalobos, J.: An MDE Approach to Design Enterprise Architecture Viewpoints. In: IEEE 12th Conference on Commerce and Enterprise Computing (CEC), November 10-12, pp. 80–87 (2010)

[11] Romero, J.R., Troya, J.M., Vallecillo, A.: Modeling ODP Computational Specifications Using UML. The Computer Journal 51, 435–450 (2008)

[12] TOGAF 1995 -The Open Group Architecture Framework, Version 8.1.1 (1995), http://www.opengroup.org/architecture/togaf8-doc/arch/

[13] Zachman, J.A.: A Framework for Information Systems Architecture. IBM Systems Journal 26(3), 276–292 (1987)

ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking

Josef Adersberger[1] and Michael Philippsen[2]

[1] QAware GmbH, Aschauer Str. 32, 81549 Munich, Germany
josef.adersberger@qaware.de
[2] University of Erlangen-Nuremberg, Computer Science Department,
Programming Systems Group, Martensstr. 3, 91058 Erlangen, Germany
philippsen@cs.fau.de

Abstract. The decay of software architecture - the divergent evolution of architecture models and the derived code - is one of the reasons for a decreasing maintainability of software systems. Several approaches for architecture-to-code consistency checking exist that stop the decay by detecting a divergence after evolution steps of either the architecture or the corresponding code. Known approaches have two main insufficiencies. First, the effort to derive and maintain the consistency checks is higher than necessary or they cannot be applied a posteriori. Second, they are not well integrated into UML-based model driven engineering. In the paper we present ReflexML: A UML-embedded mapping of architecture models to code plus a rich set of predefined consistency checks based on that mapping. The mapping is described with a UML profile that allows to attach AOP type patterns to an UML component model to define its reflexion on code elements. This abolishes the two insufficiencies of current approaches. We apply ReflexML to an industry project to demonstrate its effectiveness and its capability of a seamless integration into a pre-existing UML architecture model.

Keywords: traceability, reflexion model, architecture consistency, UML, AOP.

1 Introduction

Maintainability of software is mainly driven by its architecture. Software architecture has a classic divide-and-conquer core: All concerns of a software are split into coherent parts (components) which are then loosely coupled via clearly defined interfaces. Most modern development methodologies require to develop a software architecture and derive the implementation structure from there. But then in many real-world projects the architecture and the implementation structure diverge while the software evolves. This leads to a decreasing maintainability called architecture erosion [15] that is a typical effect of software aging [13]. Architecture erosion can be prevented by enforcing so-called architecture-to-code consistency, i.e., if code-level dependencies comply with the dependencies and component semantics defined on the architecture level.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 344–359, 2011.

There are two types of such inconsistencies [9] or architecture violations [15], namely **divergence** (a dependency between two code elements is not allowed according to the architecture model) and **absence** (a dependency in the architecture model is not represented in the code). To stop architecture erosion, methods are needed that perform architecture-to-code consistency checks after each evolution step of either the architecture model or the code.

In general, any approach to check architecture-to-code consistency has to address the following two topics: First, **architecture-to-code traceability** is required to map architecture elements to code elements. This mapping is also known as reflexion model [9]. Without being able to identify the associated code elements of an architecture element, no architecture compliance check can be applied. Second, **architecture compliance checks** are required to reveal the consistency between architecture and code. Such checks detect dependencies on code-level that are disallowed according to the architecture model.

We think that any approach should implement the following five requirements to be applicable in all scales of real-world projects. These requirements also solve the two main insufficiencies of current approaches: The high effort to define and maintain both the required mapping and the compliance checks as well as the lacking integration into model-driven engineering.

1. Single source: The native architecture models should be used as input for the approach to avoid duplication of architecture information. Otherwise maintaining both the basic architecture models and the consistency checking models would be costly and error-prone.
2. Expressive mapping: Defining every relationship by enumerating all mapped code elements is infeasible in large projects. Instead we need an expressive query-like way to describe the mapping of architecture elements to code elements to reduce the effort to create and maintain the mapping.
3. Stable mapping: The mapping should be stable with the evolution of the architecture model or the code. If the architecture is refactored or code elements are created, moved, or deleted the mapping should either stay valid or needed modification should be simple to identify.
4. Semantically rich architecture model: With semantically rich constructs like components, interfaces, and hierarchical compositions a rich set of pre-defined architecture compliance checks can be expressed. Semantically poorer models require to define a large set of explicit rules to ensure architecture compliance. E.g., the semantically poorer boxes-and-lines model only allows to define valid dependencies (the lines) between sets of classes (the boxes).
5. A priori and a posteriori application: It should be possible to perform compliance checking both from the start of the implementation of an architecture or during/after the implementation.

We present ReflexML that addresses both topics and fulfills all of the above requirements. We limit the scope of ReflexML according to the following two assumptions. First, the architecture model is described with UML component models. Alternative to UML a couple of other architecture description languages

(ADLs) are available. Most of them like AADL [5] focus on the behavioral aspect of a software architecture, are very formal and thus not commonly used in practice. In a first step we focus on architecture erosion of the static structures not of the behavior. But we will extend ReflexML to prevent behavioral architecture erosion in a future version by integrating dynamic UML models. Compared to the widespread use of UML by practitioners, even ADLs that focus on the static structures of a software architecture like xADL [4] are only used in niches. This is our reason to focus only on UML. Second, the considered granularity level of code elements are types. Type members like methods or attributes map to the same architecture-level element as the type itself. Dependencies introduced by type members are considered as dependencies of the type.

The **basic idea** of ReflexML is to define the traceability of UML component models to code by means of AOP type pattern expressions. To do so, UML component model elements are decorated with these expressions as tagged values. Beside this, the main contribution of the paper is the set of architecture consistency checks. These checks are based on UML component model semantics and software architecture principles.

With this basic idea we fulfill all the above requirements. The *single source* in our approach is a UML component model. All consistency checks can be performed with the component model and the current code as input. We do not need an external mapping model. Even an a posteriori adoption is possible by decorating pre-existing component models. If adopted a priori, the ReflexML-based architecture-to-code traceability information could also be used as an additional input for code generation. *Expressive mapping*: With type pattern expressions we use a powerful concept to describe a set of types. Type patterns were developed in the area of aspect-oriented programming to apply crosscutting code to a potentially large sets of types. Type patterns are more powerful in our context than regular expressions as they are aware of object-oriented concepts like inheritance. *Stable mapping*: At code-level the mappings are as stable as possible because each mapping is not enumerated but expressed by a type pattern. The type pattern is based on package structures, names, and inheritance that tend to be more stable than just enumerating elements. At architecture level we benefit from the fact that the mapping is directly integrated into architecture models. In case of architecture evolution the mapping information stays attached to the architecture element. Furthermore, constraints are defined that enforce the stability of the mapping itself. *Semantically rich architecture model*: We use the UML component model to represent an architecture model because it supports semantically rich elements like interfaces, components, and composite structures. *A priori and a posteriori application*: The UML reflexion profile can be applied whenever it is required on a given component model. So both a priori and a posteriori application is possible.

Below we introduce a sample application that we use to illustrate our concepts throughout the paper. We then present the two parts of ReflexML: The UML reflexion profile to define the traceability of UML component models to code

(Sect. 3) and a rich set of architecture compliance checks based on that traceability (Sect. 4). Section 5 shows the results of a case study applying ReflexML to an industry project. Section 6 covers the related work.

2 Sample Application

To illustrate ReflexML we use a sample Java application. Figure 1 shows its component model and code structure. It consists of two top-level components: A component *Mail* to send e-mails and a component *Monitor* that triggers e-mails on certain events. The component *Mail* has an embedded component *MailSender* whose concern it is to actually send e-mails via the *javax.mail* interface. Both the interfaces *IMail* and *IMailSender* depend on the *javax.mail* interface as in their signatures they use the type *MessagingException*.

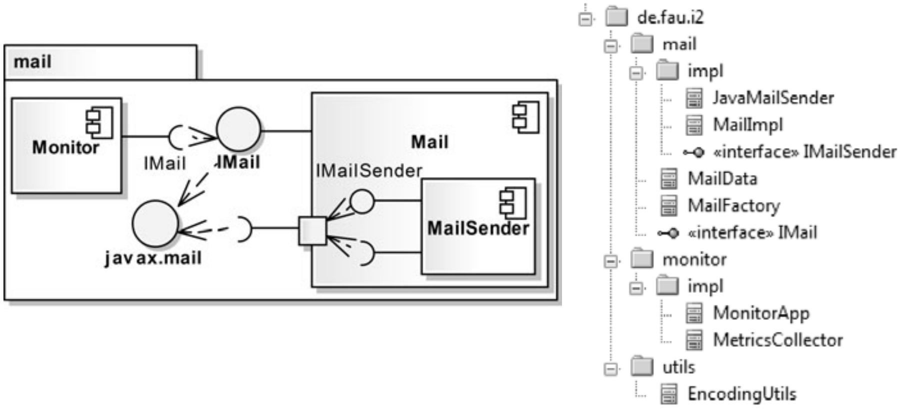


Fig. 1. Sample UML component diagram (left) and code structure (right)

The code structure is arranged according to the common best practice. Each top-level component is represented as project top-level package. The interface parts of the component *Mail* reside directly inside the top-level package. They consist of the interface type itself, a data type used to encapsulate mail data, and a factory to obtain access to the interface. The component implementation parts reside inside an *impl* sub-package as well as the interface and the implementation of the sub-component *MailSender*.

3 Architecture-to-Code Traceability

The architecture model is described with UML component models. A UML component model consists of three main element types: Components, interfaces, and their relationships. A component can require or provide an interface. Interfaces can have dependencies to each other. Ports are not considered in ReflexML as they have no relevant semantics for architecture compliance checking: A relationship is just traversed at a port. The UML reflexion profile is an efficient and robust way to describe the mapping between architecture and code for traceability. It leverages the UML profile facility to decorate UML component model elements with expressions that match sets of types. These expressions are called reflexion expressions adapting the idea of reflexion models [9].

3.1 UML Reflexion Profile

Figure 2 shows the reflexion profile. It consists of three stereotypes: *reflectedElement* to annotate either components or interfaces with reflexion expressions that map them to code elements. For packages there is the stereotype *architectureModel* that indicates that the package and its potential sub-packages are containing a complete architecture model. There is also the stereotype *reflectedExclusion* (defined for packages) that is used to identify a set of types that should not be considered for architecture compliance checks (e.g., library types or utility classes). Each stereotype uses a tagged value (*reflexion* or *exclusionReflexion*) to hold the reflexion expression.

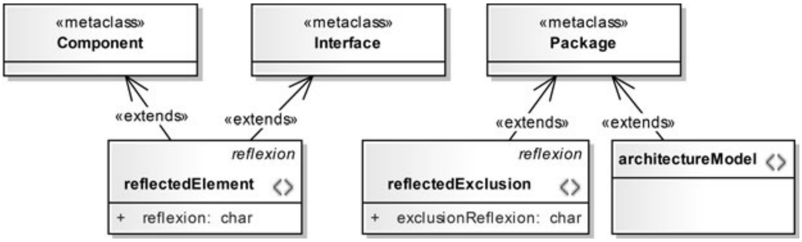


Fig. 2. UML reflexion profile

The reflexion profile also introduces constraints to UML component models. These constraints are based on the principles of component-oriented software development as described in [16] but not yet included in the semantics of the UML component model although they are required for more rigorous and thorough architecture compliance checks.

Constraint A: Dependencies are only allowed towards interfaces. A component can require or provide an interface. An interface can depend on another interface. This constraint ensures the principle of information hiding. Sample: The component *Monitor* may not directly depend on the component *Mail*. It has to use the interface *IMail*.

Constraint B: Only dependencies to interfaces in the same namespace[1] are allowed. This constraint ensures the principle of information hiding. Sample: The component *MailSender* may not have a direct dependency to *javax.mail*

[1] Note that in UML a component also represents a namespace.

which is part of the top-level package. It has to explicitly import the interface to its namespace via a port element first.

Constraint C: Each component and interface element of a component model has to be reflected to code (has to be of stereotype *reflectedElement*).

3.2 Reflexion Expression Syntax

An expressive reflexion syntax should have the following features to be amenable for software practitioners:

- Wildcard predicates for package and type names (e.g. "all classes with suffix *Entity*").
- Recursive and non-recursive predicates (e.g. "all types in package P and in all of its sub-packages" as well as "only all types directly inside package P").
- Class- and package-level predicates (e.g. "only class A, B and C (but not D) in package P" and "all classes in package F").
- Awareness of inheritance structure (e.g. "all subtypes of interface *IEntity*").
- At least the following Boolean operators must be available to build-up higher-level expressions: *AND*, *OR*, *NOT* (e.g. "all classes in package *P AND NOT* class D" or "class *A OR* class *B OR* class C in package P").

The reflexion expression syntax of ReflexML fulfills these requirements. To describe a set of types the AspectJ type pattern syntax[2] is used. See [1] for the detailed semantics of these type patterns. Basically a ReflexML type pattern is a logical expression on a set of types identified by their fully qualified name. The following three wildcards are available to match a set of types:

- "*" matches zero or more characters other than ".", e.g., *java.util.*Map*
- ".." matches any sequence of characters that start and end with a ".", e.g., *de.fau.i2..**. Thus it also matches a single point.
- "+" matches all subtypes of a type (given by preceding type name pattern), e.g., *java.util.AbstractMap+*

Atomic type patterns can also be combined with the boolean operators && (*AND*), ||(*OR*), and ! (*NOT*). For example:
*java.util.AbstractMap+ && !java.util.*HashMap*

Figure 3 shows the UML reflexion profile in use. The component model elements of the running example are decorated with mapping information. Additionally, an exclusion pattern excludes utility classes from further analysis. Please note that we do not use the circle notation of interfaces in the reflected version. This enhances the visibility of the tagged values but leads to a more complex diagram.

According the concept of reflexion models [8,9] the reflexion expressions of a component model must comply to the following constraints:

[2] www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html

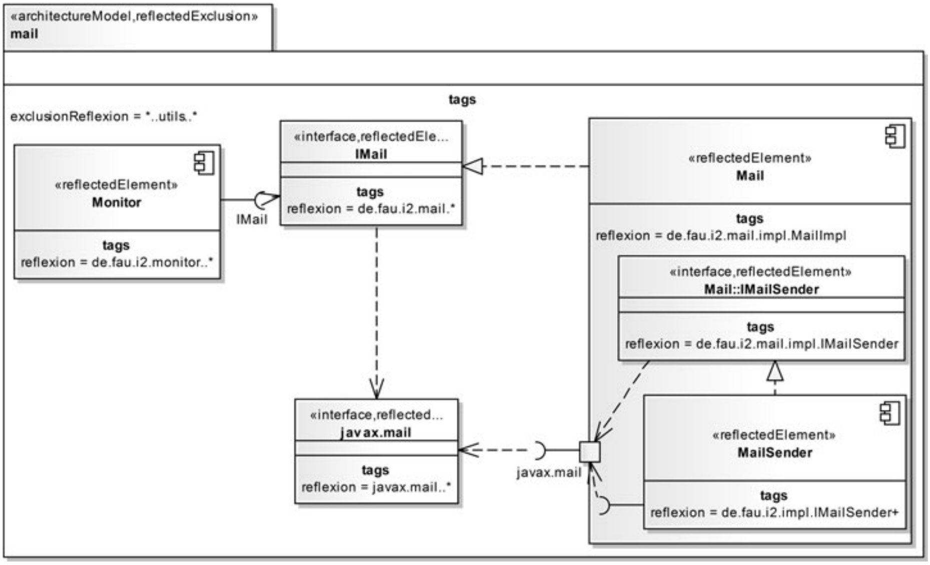

«architectureModel,reflectedExclusion»
mail
tags
exclusionReflexion = *..utils..*
«reflectedElement»
Monitor
tags
reflexion = de.fau.i2.monitor..*
IMail
«interface,reflectedEle...
IMail
tags
reflexion = de.fau.i2.mail.*
«reflectedElement»
Mail
tags
reflexion = de.fau.i2.mail.impl.MailImpl
«interface,reflectedElement»
Mail::IMailSender
tags
reflexion = de.fau.i2.mail.impl.IMailSender
«interface,reflected...
javax.mail
tags
reflexion = javax.mail..*
javax.mail
«reflectedElement»
MailSender
tags
reflexion = de.fau.i2.impl.IMailSender+

Fig. 3. Sample reflected component model

Constraint D: Relations between component model elements (components and interfaces) and code elements (types) must be 1-to-N relationships: Each type must be mapped to exactly one component model element (or excluded from analysis). And each component model element must be mapped to one or more types. This is an indication whether the component model is a valid and complete abstraction of the code structure. Sample: The implementation class of component *MailSender* (*JavaMailSender*) is only mapped to that component (and there is no mapping to its parent component *Mail*).

Constraint E: Any existing architecture-level dependency must be represented by at least one code-level dependency (no absence). Sample: The class *JavaMailSender* does not use artifacts of the interface *javax.mail* but it uses a proprietary solution instead. In this case the architecture and code would be inconsistent because the implementation would violate the architecture-level policy to use *javax.mail*.

4 Architecture Compliance Checks

Architecture compliance checks detect divergences between the implementation structure and the architecture model. The checks are based on the basic principles of component-oriented software development, namely information hiding [12] and the semantics of a component [16,11]. Architecture compliance can only be tested if all the constraints from Sect. 3 are met. To reason about the architecture compliance each code-level dependency is projected to architecture-level by following the mapping of each code element related by the dependency to an architecture element. All types of code-level dependencies are considered.[3] For each dependency 9 checks (see below) are applied. Each check can have four different results:

- OK: The dependency complies with the architecture. The result OK represents a convergence between architecture and code.
- ERROR: The dependency does not comply with the architecture. The result ERROR represents a divergence between architecture and code.
- WARNING: Although the dependency complies with the architecture it can be considered as an architectural flaw. A WARNING represents a convergence between architecture and code but has an additional informative character on potential architectural flaws.
- NONE: The check is not applicable to a dependency.

A code-level dependency represents a divergence if at least one of the following 9 checks applied to it has the result ERROR. Otherwise a dependency is convergent to the given architecture.

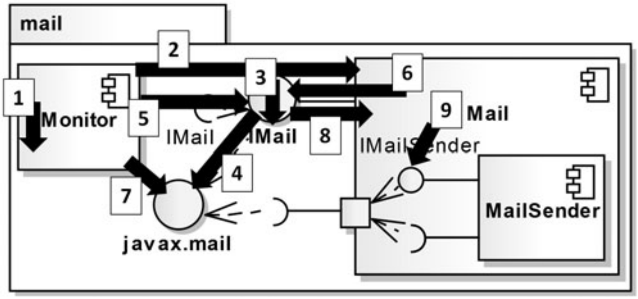


Fig. 4. Architecture compliance checks in the sample application

We use the term “artifact” to refer to all code-level elements that are mapped to a specific component model element (e.g. “provided interface artifact” for all code-level elements that are mapped to the provided interface of a component). NONE is the default result of each check.

Check 1: A dependency between artifacts of the same component is OK due to the high coherence principle. Sample: The class *MonitorApp* of component *Monitor* is allowed to have a dependency on class *MetricsCollector* (see arrow #1 in Fig. 4).

Check 2: A dependency between artifacts of different components is an ERROR. This check ensures the principle of information hiding. Sample: The class

[3] For Java code: Method access, field access, inheritance, and declaration as field-type/parameter-type/return-type/exception-type/generic type binding.

MonitorApp may not depend on *MailImpl* as this class is part of another component (see arrow #2).

Check 3: A dependency between artifacts of the same interface is OK due to the high coherence principle. Sample: The interface *IMail* may depend on class *MailData* and the class *MailFactory* may also depend on *IMail* (see arrow #3).

Check 4: If a component-level dependency between two interfaces exists, code-level dependencies between both corresponding interface artifacts are OK. A dependency between two interface artifacts of unrelated interfaces is an ERROR. This check ensures the compliance of an interface-to-interface relation on component-level with the corresponding dependencies on code-level. Sample: The interface *IMail* may depend on the class *MessagingException* assigned to the interface *javax.mail* at component-level (see arrow #4).

Check 5: A dependency between component artifacts and interface artifacts of a required interface is OK. This check ensures the compliance of a component-to-interface relation on component-level with the corresponding dependencies on code-level. Sample: The class *MonitorApp* may depend on the interface *IMail*, on the class *MailFactory*, and on the class *MailData* (see arrow #5).

Check 6: A dependency between component artifacts and all artifacts of the provided interfaces is OK. This check allows dependencies according to the semantics of the relation between a component and its provided interfaces. The component has to know the interface in order to implement the associated contract. Sample: The class *MailImpl* may depend on the interface *IMail*, on the class *MailFactory*, and on the class *MailData* (see arrow #6).

Check 7: A dependency between component artifacts and interface artifacts of an interface which is not required or provided at component-level and is not embedded in the component is an ERROR. This check ensures the compliance of a component-to-interface relation on component-level with the corresponding dependencies on code-level. This is the inverse check to checks 5 and 6. Sample: The class *MonitorApp* may not depend on any artifact of the interface *javax.mail* (see arrow #7).

Check 8: A dependency between interface artifacts and all artifacts of the components that provide this interface causes a WARNING because in that case the implementation of the interface cannot easily be exchanged. In our sample the interface artifact *MailFactory* needs a dependency to the component artifact *MailImpl*. Even though this is not perfect, it is common practice (factory pattern). Hence we just warn. But it is an ERROR if such a dependency is visible to potential users of the interface (e.g. if the according types are used in public method signatures). In this case the dependency turns transitive and even if the interface is not used, check 2 will certainly fail upon the first use. Sample: It would be an ERROR if the class *MailFactory* would return *MailImpl* instead of *IMail* (see arrow #8). A dependency between interface artifacts and artifacts of components which do not provide this interface is also an ERROR.

Table 1. Overview of checks

	Component
Component	OK (1): Same component ERROR (2): Between components
Interface	WARNING (8): Interface to providing component but non-transitive ERROR (8): Interface to non-providing component or transitive to providing component

	Interface
Component	OK (5): Component to required interface OK (6): Component to provided interface OK (9): Component to provided interface of an embedded component ERROR (7): Component to not provided, required or embedded interface
Interface	OK (3): Same interface OK (4): Interface to dependent interface ERROR (4): Interface to unrelated interface

Check 9: A dependency between component artifacts and interface artifacts of the provided interfaces of directly embedded components is OK. This dependency is implicit on the architecture-level and has to be treated here as a special case of check 4. Sample: *MailImpl* may depend on *IMailSender* (see arrow #9).

Table 1 gives an overview of all checks and groups them according to the four different kinds of dependencies between architecture elements and their assigned code elements. With this table we can reason about the two basic properties of our set of checks: This set of checks is **complete**, because first all possible constellations of the projection of code-level dependencies to architecture-level are considered and evaluated. A code-level element can either be mapped to a component or an interface. Because a code-level dependency is directed and has a code-level element at both ends, there are four possible constellations: component-to-component, component-to-interface, interface-to-interface, and interface-to-component (check Table 1). Second, as also can be seen in the table, completeness means that for each constellation there are both rules that reason a convergence (OK, WARNING) and a divergence (ERROR). The set of checks is also **free of contradictions**, as there are no two checks that have contradicting results for the same dependency. Checks of different constellations are free of contradiction because each considered set of dependencies is disjoint. This also holds for the shown checks of one constellation. In all cells of the table, the checks are disjoint to each other.

5 Case Study

To evaluate ReflexML we applied it a posteriori in several real-world industry projects in context of code reviews. The running sample presented throughout this paper is part of one of these reviewed projects. Below we describe the evaluation of ReflexML in one of the other projects in detail. The conclusions of

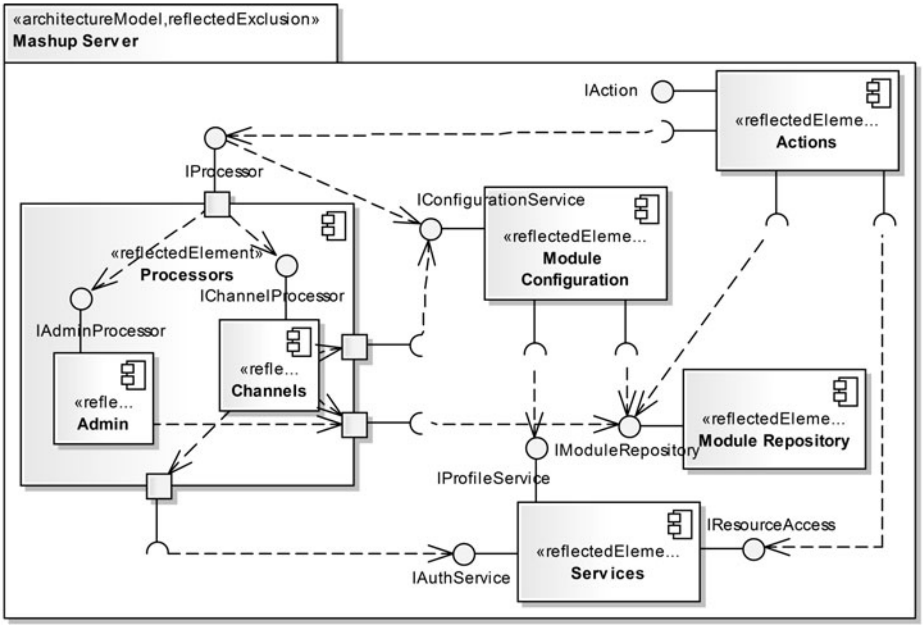


Fig. 5. Mashup server architecture

the evaluation in all other projects were equivalent to the conclusions below. The evaluation criteria were effectiveness (How much effort is it to introduce ReflexML?), single source model (How good can pre-existing models be used for ReflexML application?) and the quality of results (How good is the quality of detected violations?).

The analyzed system is a high traffic telecommunication industry web application called Mashup Server that is currently up and running. It is being developed since 2004 and has a code volume of 45,651 lines of code in 804 classes. The software architecture is defined and communicated with a UML component model holding 7 components, 9 interfaces, and 15 dependencies.

The architecture reflexion shown in Fig. 5 is the result of a 2 hour workshop with the software architect. During this workshop we incrementally enriched a pre-existing UML component model with reflexion expressions. Each increment was added in a define-analyze-correct cycle. We first defined a certain mapping and then analyzed the reported violations. We then used our ReflexML tooling to analyze the architecture-to-code consistency. If we did not consider a detected violation as an architectural flaw, we either corrected the mappings or adapted the architecture model. Hereby we detected two required but not yet defined architecture-level dependencies. Finally, the architecture model has been enriched with 17 reflexion expressions.

In the years of development, the team has used the following rules to derive code-level artifacts from architecture-level artifacts:

- For each component there is a top-level package <P>.
- The artifacts of all provided interfaces of a component belong into <P>.
- All component artifacts either belong into <P> but named with suffix *Impl* or into a sub-package of <P>.

Due to these rules, nearly all defined reflexion expressions follow common patterns. The pattern to describe the mapping of components if the corresponding code artifacts are created within sub-packages is:
*mashupserver.<P>..** && *!mashupserver.<P>.**

If the implementation artifacts are created within the top-level package but distinguishable by the suffix *Impl* the following pattern was used:
*mashupserver.<P>.*Impl*

Architecture-level interfaces are mapped to code with the following pattern (independent of the variant a component is mapped to code):
*mashupserver.<P>.** && *!mashupserver.<P>.*Impl*

There was one exception to the rules. The code-level artifacts of the component *Actions* were scattered over multiple top-level packages. But they also have one common characteristic: They are all subtypes of a certain class called *MashupAction*. This could also be expressed with the following reflexion expression (also excluding these classes from other component mappings):
mashupserver.dispatcher.MashupAction+

The tooling we used is based on Eclipse MWE[4] and Macker.[5] We wrote a model-to-text transformation that generates Macker rules based on the ReflexML checks defined in Sect. 4 and a given UML component model where the UML reflexion profile is applied. The transformation has generated 210 lines of Macker rule definition from the reflected mashup server architecture model containing 17 reflexion expressions.

Figure 6 depicts the detected violations on architecture elements. A node in Fig. 6 groups a component and its provided interfaces. Each edge represents a violation against a ReflexML check. The number of code-level dependencies causing a violation are noted next to the edge. We discuss the bold edges below. Table 2 shows the statistics for performing all ReflexML checks on the mashup server code. Every check from Sect. 4 that has ERROR as one of its potential results, has in fact signalled ERROR.

The analysis result was discussed with the software architect. ReflexML did not detect any false positives. But it discovered two major architectural flaws previously unknown to the developers:

- There are three code-level dependencies that did not have any match in the architecture. From *ModuleConfiguration* to *Processors*, from *Services* to *ModuleRepository*, and from *ModuleRepository* to *Processors*. Each of these dependencies does not make any sense from the architectural perspective. The architect has decided to have the programmers eliminate them.

[4] Modeling Workflow Engine (www.eclipse.org/modeling/emft): A toolset for model processing including model transformation and text generation.

[5] innig.net/macker: An open source tool to perform code-level dependency analysis.

Table 2. Analysis statistics

Number of:	
code-level dependencies	7300
errors (total)	207
errors (Check 2)	74
errors (Check 4)	113
errors (Check 7)	7
errors (Check 8)	13

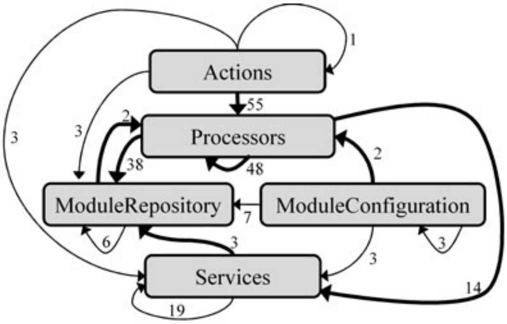


Fig. 6. Architecture violations overview

- Missing separation of interface types and implementation types in the component *Processors*. This is indicated by the high number of incoming (55+2+2), recursive (48), and outgoing (38+14) violations of component *Processors*. The architect decided to have his team refactor the component accordingly.

ReflexML was a success in this industry project. The effort to introduce ReflexML a posteriori was low (2 working hours). The integration of pre-existing architecture models was possible. They could seamlessly be enriched and used as input to perform ReflexML checks. No separate model had to be derived. The quality of the analysis results can be considered as high because ReflexML did not announce any false positives but signalled major architectural flaws that are being fixed.

6 Related Work

Several architecture compliance checking methods exist to check an implementation against an architecture model. [14] and [7] evaluate, compare, and categorize the different approaches for architecture compliance checking into four groups: (a) Reflexion Models [9] with high-level models and a declarative mapping to code. Convergences, divergences and absences are found automatically. Reflexion models are extended in [8] with the ability to define composite (hierarchical) high-level models. (b) Relation conformance rules are used to declare allowed or forbidden relations between code elements. [14] refers to the Dependency Structure Matrix technique as a special case to formulate relation conformance rules. (c) Component access rules are used to describe public and private types of code-level components (e.g. a package or namespace). (d) Source Code Query Languages allow to express code-level constraints based on an abstract code model. Relation conformance and component access rules can also be formulated by means of a source code query.

ReflexML adopts concepts from all four categories: The reflexion profile is based on reflexion models. The reflexion expression syntax is a source code query language. The architecture compliance checks are based on component access rules and relation conformance rules derived from the reflexion model.

We consider those architecture compliance checking approaches as closely related that enhance the reflexion model approach with a way to derive the architecture-to-code traceability. Hence, we compare ReflexML to them and focus on the requirements defined in Sect. 1. As ReflexML is designed for software practitioners we also discuss how available tools meet these requirements.

Terra's domain-specific textual language DCL [17] for architecture compliance checking combines the reflexion model and the relation conformance rule method. Relation conformance rules have to be manually derived from a pre-existing architecture model. DCL does not support a single source architecture model. Its reflexion expression syntax is powerful and meets all requirements of Sect. 3.2. The supported architecture model is not as semantically rich as a UML component model because it does not support an interface construct. DCL can either be used a priori and a posteriori in a project.

Biehl's approach [2] is based on reflexion models and defines a method focused on model-driven software development. The mapping is derived while architecture models are transformed into implementation models. A dependency clustering technique maps code elements that are created after the transformation process to architecture elements. The method uses UML class models and the according transformation procedures as the single source for compliance checking. Similar to ReflexML no separate model has to be maintained. The mapping is expressive as it is formulated along the transformation functions. In case of a code evolution the mapping is stable as long as the (fuzzy) clustering algorithm is reliable. The architecture is described with a UML class model. However in our opinion the abstraction level of UML class models is too low for describing large-scale architectures. Also Biehl et al. do not support an a posteriori application because a certain model-driven approach has to be adopted from the beginning of a project.

Beside these two approaches that did not yet find their ways into publicly available tools, many commercial and open source tools are available for architecture compliance checking.[6] Table 3 compares them with ReflexML according their architecture model, their support of single source architecture models, and shows whether they are open source software.

Other than ReflexML, none of them support a single source architecture model like a UML component model. For each tool a separate architecture compliance

[6] Structure101, headwaysoftware.com/products/structure101,
SonarJ, hello2morrow.com/products/sonarj,
Bauhaus Suite, axivion.com,
Sotograph, hello2morrow.com/products/sotograph,
ConQAT, conqat.in.tum.de/index.php/ConQAT,
Lattix LDM, lattix.com,
Dependometer, sourceforge.net/projects/dependometer,
Macker, innig.net/macker,
XRadar, xradar.sourceforge.net,
Classycle, classycle.sourceforge.net

Table 3. Tool comparison

Product	**Architecture model**	**Single source models**	**License**
structure101	boxes and lines, layers	no	commercial
SonarJ	layers, slices	no	commercial
Bauhaus Suite	boxes and lines, components	no	commercial
Sotograph	layers, components	no	commercial
ConQAT	boxes and lines	no	open source
Lattix LDM	boxes and lines	no	commercial
Dependometer	layers, slices	no	open source
Macker	boxes and lines	no	open source
XRadar	boxes and lines	no	open source
Classycle	boxes and lines, layers	no	open source
ReflexML	components	yes	open source

checking model has to be derived from an existing architecture model. Most tools work with semantically poor architecture models like boxes-and-lines or layers. Only some tools like Sotograph [3] or the Bauhaus Suite support even semantically rich models comparable to UML component models with constructs like components, interfaces and connectors.

We do not discuss approaches to architecture-to-code traceability as to our knowledge there is none that supports architecture compliance checks. But ReflexML is inspired by some ideas of that area, e.g., the extension of a common-purpose architecture model to support the traceability to code [10] or high-level mapping expressions like OCL used in model weaving [6].

7 Conclusion

Architecture-to-code compliance checking is an important tool to stop a decreasing maintainability because of architecture erosion. In this paper we have presented ReflexML for architecture compliance checking. It enforces that an architecture model has to be a valid and complete abstraction of the implementation structure throughout the whole software development process. A main contribution of our work is the UML reflexion profile, i.e., an efficient mapping of architecture elements to code that is more expressive than classic wildcard-based approaches since it uses AOP type patterns. The main artifacts of UML-based architecture development (a UML component model and the corresponding code) stay the main artifacts during a ReflexML enriched development. The real-world case study and the running sample application demonstrate that the UML reflexion profile is lightweight, easy-to-use, and applicable. The insufficiencies of other approaches, i.e., the high effort to introduce and conduct them, are abolished with ReflexML, even if it is introduced a posteriori. Our second contribution is the set of constraints and checks that prevent a divergent evolution of architecture and code. They allow to detect a variety of architecture violations just by decorating existing component models with reflexion expressions. We are currently working on integrating ReflexML to various software development tools.

Acknowledgments. This research work is funded by the AIF ZIM program of the German Federal Ministry of Economics and Technology.

References

1. Avgustinov, P., Hajiyev, E., Ongkingco, N., de Moor, O., Sereni, D., Tibble, J., Verbaere, M.: Semantics of static pointcuts in AspectJ. SIGPLAN Not. 42(1), 11–23 (2007)
2. Biehl, M., Löwe, W.: Automated architecture consistency checking for model driven software development. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 36–51. Springer, Heidelberg (2009)
3. Bischofberger, W., Kühl, J., Löffler, S.: Sotograph - A pragmatic approach to source code architecture conformance checking. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 1–9. Springer, Heidelberg (2004)
4. Dashofy, E.M., Van der Hoek, A., Taylor, R.N.: A highly-extensible, XML-based architecture description language. In: WICSA 2001: Proc. of the Conf. on Softw. Architecture (Amsterdam), pp. 103–112 (2001)
5. Feiler, P., Lewis, B., Vestal, S., Colbert, E.: An overview of the SAE architecture analysis and design language (AADL) standard. In: Architecture Description Languages. IFIP Int. Fed. for Inf. Proc., vol. 176, pp. 3–15 (2005)
6. Groher, I., Voelter, M.: XWeave: models and aspects in concert. In: AOM 2007: Proc. Intl. Work. Aspect-Oriented Modeling, Vancouver, pp. 35–40 (2007)
7. Knodel, J., Popescu, D.: A comparison of static architecture compliance checking approaches. In: WICSA 2007: Proc. Conf. Softw. Architecture, Mumbai, pp. 12–21 (2007)
8. Koschke, R., Simon, D.: Hierarchical reflexion models. In: WCRE 2003: Proc. Conf. Reverse Eng., Victoria, pp. 36–45 (2003)
9. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: bridging the gap between source and high-level models. In: SIGSOFT 1995: Proc. Symp. Foundations of Software Eng., Washington, D.C, pp. 18–28 (1995)
10. Murta, L.G., Hoek, A., Werner, C.M.: Continuous and automated evolution of architecture-to-implementation traceability links. Automated Softw. Eng. 15(1), 75–107 (2008)
11. OMG. UML 2.3 Superstructure Specification. OMG (May 2010)
12. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)
13. Parnas, D.L.: Software aging. In: ICSE 1994: Proc. Intl. Conf. Softw. Eng., Sorrento, pp. 279–287 (1994)
14. Passos, L., Terra, R., Valente, M.T., Diniz, R., Mendonca, N.d.C.: Static architecture-conformance checking: An illustrative overview. IEEE Softw. 27(5), 82–89 (2010)
15. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
16. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (2002)
17. Terra, R., Valente, M.T.: A dependency constraint language to manage object-oriented software architectures. Softw. Pract. Exper. 39(12), 1073–1094 (2009)

Software Is a Directed Multigraph

Robert Dąbrowski, Krzysztof Stencel, and Grzegorz Timoszuk

Institute of Informatics
Warsaw University
Banacha 2, 02-097 Warsaw, Poland
{r.dabrowski,k.stencel,g.timoszuk}@mimuw.edu.pl

Abstract. The *architecture of a software system* is typically defined as the organization of the system, the relationships among its components and the principles governing their design. By including artifacts coresponding to software engineering processes, the definition gets naturally extended into the *architecture of a software system and process.* In this paper we propose a holistic model to organize knowledge of such architectures. This model is graph-based. It collects architectural artifacts as vertices and their relationships as edges. It allows operations like metric calculation, refactoring, bad smell detection and pattern discovery as algorithmic transformations on graphs. It is independent of development languages. It can be applied for both formal and adaptive projects. We have implemented prototype tools supporting this model. The artifacts are stored in a graph database. The operations are defined in a graph query language. They have short formulation and are efficiently executed by the graph database engine.

Keywords: architecture, graph, metric, model, software.

1 Introduction

As long as there were no software systems, managing their architecture was no problem at all; when there were only simple systems, managing their architecture became a mild problem; and now we have gigantic software systems, and managing their architecture has become an equally gigantic problem (to paraphrase Edsger Dijkstra).

Nowadays software systems are being developed by teams that are: changing over time; working under time pressure; working over incomplete documentation and changing requirements; integrating unfamiliar source-code in multiple development technologies, programming languages, coding standards; productively delivering only partially completed releases in iterative development cycles.

When at some point development issues arise (bugs, changes, extensions), they frequently lead to refactoring of the software system and the software process. Even if the issues get addressed promptly, they often return in consecutive releases due to volatile team structure, insufficient flow of information, inability to properly manage architectural knowledge about the software system and the software process.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 360–369, 2011.

Unsurprisingly such challenges have already been identified and software engineering is focused on their resolution.

In particular there emerged a number of software development methodologies (e.g. structured, iterative, adaptive), design models (e.g. Entity Relationship Diagram, Data Flow Diagram, State Transition Diagram), development languages (e.g. functional, object-oriented, aspect-oriented) and production management tools (e.g. issue trackers, build and configuration managers, source-code analyzers). Although they address important areas, it is still a challenge to integrate those methodologies, standards, languages, metrics, tools into a consistent environment. Such an environment should (1) include all software system and software process artifacts; (2) identify their dependencies; (3) facilitate systematic build of deliverables. Furthermore, it should be resilient to changes of the development team. This property can be achieved provided all architectural knowledge is preserved in this environment's repository.

For software practitioners this current lack of integration of architectural knowledge is a historical condition: while software was limited to a small number of files delivered in one programming language and built into a single executable, it was possible to browse the artifacts in a *list* mode (file by file; or procedure by procedure). Next, as software projects evolved to become more complex and sophisticated, the idea of a software project organized according to a *tree* (folders, subfolders and files; or classes, subclasses and methods) emerged to allow browsing artifacts in a hierarchical approach.

This is no longer enough. We believe that although software engineering is going in the right direction, the research will lack proper momentum without a new sound model to support integration of current trends, technologies, languages. A new vision for architectural repository of software system and software process is required and this paper aims to introduce one in order to trigger a discussion.

Our concept can be summarized as follows. All software system and software process artifacts being created during a software project are explicitely organized as vertices of a *graph* (being the next step after the *list* and *tree*) connected by multiple edges that represent multiple kinds of dependencies among those artifacts. The key aspects of software production like quality, predictability, automation and metrics are easily expressible in graph-based terms. The integration of source code artifacts and process artifacts in a single model opens new possibilities. They include e.g. defining new metrics and qualities that take into account all architectural knowledge and not only the source code.

This concept of a graph-based model for software and software process has been briefly anounced in [4]. In this paper we present a detailed definition of the model and demonstrate by example that its implementation if feasible using graph databases.

The rest of the paper is organized as follows. In Section 2 we analyze the background that motivated our approach. In Section 3 we provide a definition of the graph-based model for architectural knowledge management. In Section 4 we describe our prototype implementation using a graph database. Section 5 concludes and enumerates challenges for further research.

2 Related Work

The idea of software development described in this paper is not an entirely novel one. It has been contributed to by several existing approaches and practices.

Software engineering strives for quantitative assessment of software quality and software process predictability. Typically this is achieved by different metrics. Frequently there are many contradicting definitions of a given metric (i.e. they depend on the implementation language). It has been suggested by Mens and Lanza [11] that metrics should be expressed and defined using a language-independent metamodel based on graphs. Such graph-based approach allows for an unambiguous definition of generic object-oriented or higher-order metrics.

Also Gossens, Belli, Beydeda and Dal Cin [7] considered view graphs for representation of source code. Such graphs are convenient for program analysis and testing at different levels of abstraction (e.g. white-box analysis and testing at the low level of abstraction; black-box analysis and testing at the high level of abstraction). A graph-based approach integrates the different techniques of analysis and testing.

Modern software models often describe systems by a number of (partially) orthogonal views (e.g. state machine, class diagram). Abstract models are often transformed into platform-specific models, and finally into the code. During such transformations it is usually not possible to keep a neat separation into different views (e.g. the specification language of the target models might not support all such views). The target model, however, still needs to preserve the behavior of the abstract model. Therefore, model transformations have to be capable of moving behavioral aspects across views. Derrick and Wehrheim [5] studied aspects of model transformations from state-based views (e.g. class specifications with data and methods) into protocol-based views (e.g. process specifications on orderings of methods) and vice versa. They suggested that specification languages for these two views should be equipped with a joint, formal semantics which enables a proof of behavior preservation and consequently derives conditions for the transformations to be behavior-preserving. Also Fleurey, Baudry, France and Ghosh [6] have observed that it is necessary to automatically compose models to build a global view of the system. The graph-based approach allows for a generic framework of model composition that is independent from a modeling language.

The use of components is beneficiary for the development of complex software systems. However, component testing is still one of the top issues in software engineering. In particular, both the developer of a component and the developer of a system, while using components, often face the problem that information vital for certain development tasks is not available. One of its important consequences is that it might not only obligate the developer of a system to test the components used, it might also complicate these tests. Beydeda and Gruhn [2] have focused on component testing approaches that explicitly respect this lack of information during development.

As Kühne, Selic, Gervais and Terrier [9] have noticed, an automated transition from use cases to activity diagrams would provide significant, practical help. Additionally, traceability could be established through automated transformation, which could then be used to relate requirements to design decisions and test cases. They proposed an approach to automatically generate activity diagrams from use cases while establishing traceability links. Such approach has already been implemented (e.g. RAVEN, *ravenflow.com*).

Osterweil [12] perceived software systems as large, complex and intangible objects developed without a suitably visible, detailed and formal descriptions of how to proceed. He suggested that not only the software, but also software process should be included in software project as programs with explicitly stated descriptions. According to Osterweil, software architect should communicate with developers, customers and other managers through a software process program, indicating steps that are to be taken in order to achieve product development or evolution goals. Osterweil postulates that developers would benefit from communicating by software process programs, as reading them should indicate the way in which work is to be coordinated and the way in which each individual's contribution is to fit with others' contributions. In that sense software process program would be yet another artifact in the graph we propose in this paper.

An RDF (Resource Description Framework) model [10] is also worth mentioning. The model presented in this paper is somehow similar to RDF idea. RDF defines triples *subject-predicate-object* which are similar to graph relationships (triples: *vertex-egde-vertex*). It is usually stored in textual formats (XML or N3 format). Several languages have already been proposed to query this model, like: Sesame [3] and SPARQL [13].

3 Model

In this section we introduce a graph-based model for software engineering methodologies. The model is based on *directed multigraphs*.

Definition 1. *Let $\mathcal{S}$ be a software-intensive system. Let $\mathcal{A}$ denote the set of all types of artifacts that are created during construction of S, let $\mathcal{D}$ denote the set of all types of dependencies among those artifacts. In the remaining part we assume $\mathcal{A}, \mathcal{D}$ to be given and denote $\mathcal{S} = \mathcal{S}(\mathcal{A}, \mathcal{D})$.*

The set $\mathcal{A}$ is a dictionary of attributes that annotate artifacts created during development of $\mathcal{S}$. For the simplicity of reasoning we assume $\mathcal{A}$ to be predefined in the rest of the paper. There remains a challenge to derive a representative and consistent classification (a superset) of such attributes, though during a given software project only a subset of $\mathcal{A}$ would be typically used.

Example 1. Typically, $\mathcal{A}$ may contain some of the following values: class; coding standard; field; grammar; interface; library; method; module; requirement; test suite; use case; unit test.

Analogically, the set $\mathcal{D}$ is the dictionary of labels describing dependencies traced among the artifacts. Again, in the remainder of this paper we assume it to be predefined, although the set of actual dependencies may be software-specific and derivation of a common superset remains a challenge.

Example 2. Typically, $\mathcal{D}$ may contain some of the following values: apply to, call, contain, define, depend on, generate, implement, limit, require, return, override, use, verify.

Definition 2. *The* software graph $\mathcal{G}$ *is an ordered triple* $\mathcal{G}(\mathcal{S}) = (\mathcal{V}, \mathcal{L}, \mathcal{E})$*, where* $\mathcal{V}$ *is the set of vertices that represent the artifacts of software system or software process,* $\mathcal{L} \subseteq \mathcal{V} \times \mathcal{A}$ *is the labeling of vertices with their attributes,* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{D} \times \mathcal{V}$ *is the set of directed edges that trace dependencies between artifacts.*

Example 3. Typically, $\mathcal{E}$ may contain some of the following values: a class *calls* a class; a class *contains* a field; a class *contains* a method; a class *implements* an interface; a coding standard *limits* a module; a grammar *generates* a class; a method *calls* a method; a module *depends on* a module; a requirement *defines* a module; a unit test *verifies* a method.

$\mathcal{G}$ is a *multigraph*, that is there can be more than one edge in $\mathcal{E}$ from one vertex in $\mathcal{V}$ to another vertex in $\mathcal{V}$. $\mathcal{G}$ is a *directed graph*, that is forward and backward relations traced among artifacts are distinguished.

Example 4. Figure 1 shows an example software graph $\mathcal{G}$ where $\mathcal{A} = \{$ Abstract class, Class, Field, Method $\}$, $\mathcal{D} = \{$ CALL, CONTain, EXTend, OVERride $\}$.

The model integrates all artifacts created during a software project. It provides a graph-based abstraction of software engineering methodology. Being graph-based, the abstraction is well recognized in software community; in particular for many problems there already exist efficient graph algorithms.

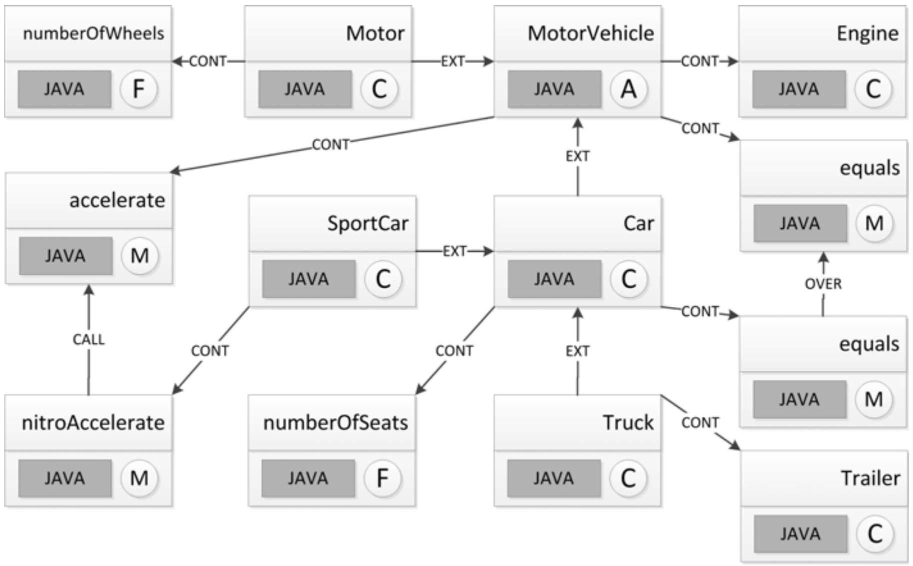


Fig. 1. An example software graph

We provide now several examples to demonstrate how the model can be applied to collect software architectural knowledge and to analyze its properties. For this purpose, we introduce some model transformations. The list of transformations presented in this paper is not exclusive and there remains a challenge to provide a canonical classification of such operations (including basic operations like adding or deleting graph nodes, or graph edges). However they can be summarized by the following intuitive set of main transformation types: an *evaluation* that maps a graph into a real number; a *selection* that maps a graph into one of its subgraphs; and a *transition* that maps a graph into a new graph (and in particular may introduce new vertices or edges).

First we define the *diagram* transformation that limits the graph to a given scope of artifacts and dependencies. The transformation is particularly useful for providing human-convenient representation of the graph, as in a non-trivial software project the model itself may grow large.

Definition 3. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ and subsets of its artifact types $\mathcal{A}' \subseteq \mathcal{A}$ and dependency types $\mathcal{D}' \subseteq \mathcal{D}$, its* diagram *is a selection $\mathcal{G}|_{\mathcal{A}',\mathcal{D}'} = (\mathcal{V}', \mathcal{L}', \mathcal{E}')$, where $\mathcal{V}' = \{v \in \mathcal{V} | \exists_{a \in \mathcal{A}'} (v, a) \in \mathcal{L}\}$, $\mathcal{L}' = \mathcal{V}' \times \mathcal{A}'$ and $\mathcal{E}' = \mathcal{E} \cap (\mathcal{V}' \times \mathcal{L}' \times \mathcal{V}')$.*

In particular, this transformation allows generating the *class* and *entity relationship diagrams* directly from the model.

Example 5. Figure 2 shows the graph $\mathcal{G}_1$ that is a selection $\mathcal{G}|_{A',D'}$ where $\mathcal{A}' = \{$ Abstract class, Class $\}$, $\mathcal{D}' = \{$ Contain, Extend $\}$.

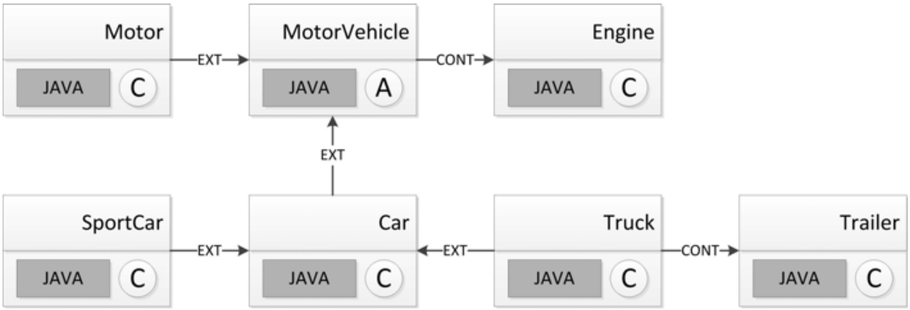


Fig. 2. The result of an example diagram transformation

Software architects may choose to stop distinguishing certain differences in artifact or dependency types (adapt a higher level of abstraction, e.g. hide fields and methods while preserving class dependencies). For this purpose we define the *map* transformation. The transformations can be combined, e.g. the map transformation combined with the diagram transformation is useful for generating visual representation (e.g. two or three-dimensional) of a given software graph.

Definition 4. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ and $t : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$, its* map *is a transition $\mathcal{G}|^t = \{\mathcal{V}, \mathcal{L}, \mathcal{E}'\}$, where $\mathcal{E}'$ is the set of new edges resulting from a transitive closure of t calculated on the neighboring edges of vertices in $\mathcal{G}$.*

Example 6. Figure 3 shows the graph $\mathcal{G}_2$ that is the result of a combination of a map and a selection $\mathcal{G}^f|_{\mathcal{A}',\mathcal{D}'}$ where $\mathcal{A}' = \{$ Abstract class, Class $\}$, $\mathcal{D}' = \{$ Call, Contain, Extend $\}$ and $f : \{$ Call, Contain, Extend $\} \mapsto \{$ Depend $\}$.

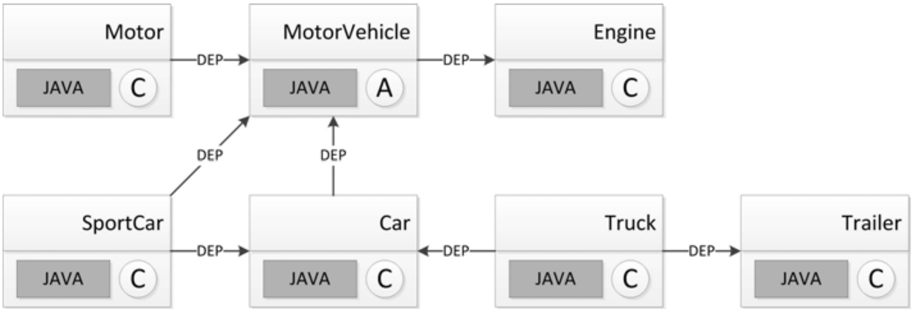


Fig. 3. The result of the example transformations composed of a map and a selection

Software architects need to assess the model quantitatively. For this purpose, we introduce *metric* transformations. The graph-based approach not only allows using existing metric that can be efficiently calculated using graph algorithms, but also allows designing new metrics. The metrics that integrate both system and process artifacts are particularly interesting.

Definition 5. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$, its* metric *is an evaluation $m : \mathcal{V}, \mathcal{L}, \mathcal{E} \mapsto \mathcal{R}$ ($\mathcal{R}$ being real numbers) which can be calculated by a graph algorithm on $\mathcal{V}, \mathcal{L}, \mathcal{E}$.*

Sometimes vertices that meet certain conditions need to be discovered. For this purpose we introduce *detection* transformations.

Definition 6. *For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$ and $f : \mathcal{V} \mapsto bool$, its* detection *is a selection $\mathcal{G}^f = (\mathcal{V}', \mathcal{L}, \mathcal{E}')$, where $\mathcal{V}' = \{v \in \mathcal{V} | f(v) = true\}$, $\mathcal{E}' = (\mathcal{V}' \times \mathcal{L} \times \mathcal{V}')$.*

This way *discovery of bad smells* can be conducted using detection transformation. In particular, we can easily find classes that define own fields but do not redefine the comparison method.

Example 7. Figure 4 shows the graph $\mathcal{G}_3$ that is a detection $\mathcal{G}|^f$ where $f(v)$ evaluates to *true* iff: v is of type *Class* and does have a neighbor of type *Field* and does not have a neighbor *equals*() of type *Method*.

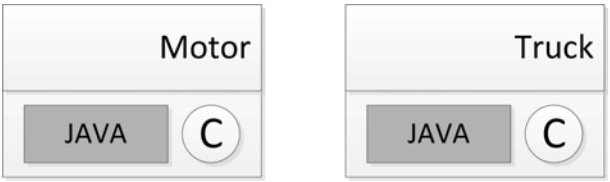


Fig. 4. A bad smell detected

4 Model Implementation

We have decided to implement the repository with a graph database. Graph databases are a member of the family of NoSQL databases that directly store unconstrained graph structures. Therefore they are well-suited for the needs of our approach. Graph databases provide efficient traversal between the vertices, called index-free adjacency. The graph structure in such a database is explicit, thus joins and index probes are not necessary to walk the graph from one vertex to another. This facility is important for model browsing tools and IDEs.

Graph databases provide also implementations of query languages and transactional operations. A query language is needed to easily define and efficiently execute graph transformations sketched in Section 3. Transactional operations are necessary for large teams who work concurrently on the same repository.

For our implementation we have selected an open-source graph database neo4j (*neo4j.org*). Neo4j offers high-availability facilities that make it feasible to build repositories for large projects. To express model operations we have selected a specific graph query language Gremlin (*github.com/tinkerpop/gremlin*). Gremlin is a path language similar to XPath, however a number of additional facilities like backtrack and loops make Gremlin Turing-complete. Thus, we can code in Gremlin any calculation, selection and transition as described in Section 3.

We have implemented in Gremlin a number of graph transformations. As an example we show a selection of classes with a specific bad smell: namely classes that add own fields but do not redefine the comparison method *equals*. The result of this transformation applied to the graph from Figure 1 is shown on Figure 4.

The respective query in Gremlin follows.

As this code shows, a relatively complex search condition has a concise formulation in Gremlin.

```
g = new Neo4jGraph('Repository')\
g.V{ it.TYPES_KEY == '[JAVA_CLASS]'\
&&  !it.outE('CONTAINS').inV{it.NAME_KEY=='equals'}\
&&   it.outE('CONTAINS').inV{it.TYPES_KEY=='[JAVA_FIELD]'} \
&&   it.outE('EXTENDS').inV.loop(2){\
        !it.object.outE('CONTAINS').inV{it.NAME_KEY=='equals'} } \
}.NAME_KEY
```

V is the collection of all vertices of graph g. The query performs a filter to this collection. The first part of the condition selects nodes that are Java classes. The second drops all nodes that stretch an edge *contains* towards a node describing a method *equals*. The third keeps only those classes that have own Java field. The forth is the most complex since it utilities Gremlin's *loop* step. This conditions traverses upwards the inheritance lattice and stops when it finds a class having a method *equals*. Only when such an ancestor is found, the tested class is added to the result of the selection. When this step is finished, the query projects its result to the value of the $NAME\_KEY$ property. Eventually, we get the following answer conformant with the contents of Figure 4.

```
==>Motor
==>Truck
```

5 Conclusions

Following the research on architecture of software [8] and software process, we propose an approach that avoids separation between software and software process artifacts as the one worth taking [12]. Implementation of such approach has already became feasible - starting with a graph-based model and using graph databases [1] as the foundation for artifact representation.

The concept is not an entirely novel one, rather it should be perceived as an attempt to support existing trends with a sound and common foundation. A holistic approach is required for current research to gain proper momentum, as despite many advanced tools, current software projects still suffer from a lack of visible, detailed and complete setting to govern their architecture and evolution.

We are also aware that the scope of research required to turn this idea into an actual contribution to software engineering requires further work. In particular, the following research areas seem to be especially inspiring: assessing a representative number of existing projects in an effort to provide a systematic classifications of artifact types $\mathcal{A}$ and dependency types $\mathcal{D}$; perhaps the artifact types and dependency types should evolve rather to be trees then mere lists; designing metric (in graph-based terms, so they can be calculated by graph algorithms) to assess software quality and software process maturity; implementing graph algorithms to calculate those metrics; classifying existing software and its process according to the model, in particular calculating metric in order to assess software quality and software process maturity, which would eventually allow comparing software projects with one another; defining UML diagrams as reports obtained from the integrated software graph as a combination of its transformations; precise definitions for the model and its components (views, maps), new components enriching the model; productive implementation of the graph based on graph databases; a project query language that would operate on the graph model and allow architects and developers to conveniently filter, zoom and drill-down the project's architectural information.

References

1. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Computing Surveys 40(1) (2008)
2. Beydeda, S., Gruhn, V.: State of the art in testing components. In: Proceedings of Third International Conference on Quality Software, pp. 146–153. IEEE Computer Society, Los Alamitos (2004)
3. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: Proceedings of the First International Semantic Web Conference, pp. 54–68 (2002)
4. Dąbrowski, R., Stencel, K., Timoszuk, G.: Software is a directed multigraph (and so is software process). arXiv:1103.4056 (2011)
5. Derrick, J., Wehrheim, H.: Model transformations across views. Science of Computer Programming 75(3), 192–210 (2010)
6. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A generic approach for automatic model composition. In: Proceeding of MoDELS Workshops, pp. 7–15 (2007)
7. Gossens, S., Belli, F., Beydeda, S., Dal Cin, M.: View graphs for analysis and testing of programs at different abstraction levels. In: Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering, pp. 121–130. IEEE Computer Society, Los Alamitos (2005)
8. Kruchten, P., Lago, P., van Vliet, H., Wolf, T.: Building up and exploiting architectural knowledge. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 291–292. IEEE Computer Society, Los Alamitos (2005)
9. Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.): ECMFA 2010. LNCS, vol. 6138. Springer, Heidelberg (2010)
10. Lassila, O., Swick, R.R.: Resource description framework (RDF) model and syntax specification. W3C Recommendation (1999)
11. Mens, T., Lanza, M.: A graph-based metamodel for object-oriented software metrics. Electronic Notes in Theoretical Computer Science 72(2), 57–68 (2002)
12. Osterweil, L.: Software processes are software too. In: Proceedings of the 9th International Conference on Software Engineering, pp. 2–13. IEEE Computer Society, Los Alamitos (1987)
13. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (2008)
14. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Communications of the ACM 16, 372–378 (1973)
15. Hasse, P., Broekstra, J., Eberhart, A., Volz, R.: A comparison of RDF query languages. The Semantic Web (2004)

An Architectural Approach to End User Orchestrations

Vishal Dwivedi[1], Perla Velasco-Elizondo[2], Jose Maria Fernandes[3],
David Garlan[1], and Bradley Schmerl[1]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA
[2] Centre for Mathematical Research (CIMAT), Zacatecas, ZAC, 98060, Mexico
[3] IEETA/DETI, Uni. of Aveiro, Campus Universitario de Santiago, 3810-193 Aveiro, Portugal

Abstract. Computations are pervasive across many domains, where end users have to compose various heterogeneous computational entities to perform professional activities. Service-Oriented Architecture (SOA) is a widely used mechanism that can support such forms of compositions as it allows heterogeneous systems to be wrapped as services that can then be combined with each other. However, current SOA orchestration languages require writing scripts that are typically too low-level for end users to write, being targeted at professional programmers and business analysts. To address this problem, this paper proposes a composition approach based on an end user specification style called SCORE. SCORE is an architectural style that uses high-level constructs that can be tailored for different domains and automatically translated into executable constructs by tool support. We demonstrate the use of SCORE in two domains - dynamic network analysis and neuroscience, where users are intelligence analysts and neuroscientists respectively, who use the architectural style based vocabulary in SCORE as a basis of their domain-specific compositions that can be formally analyzed.

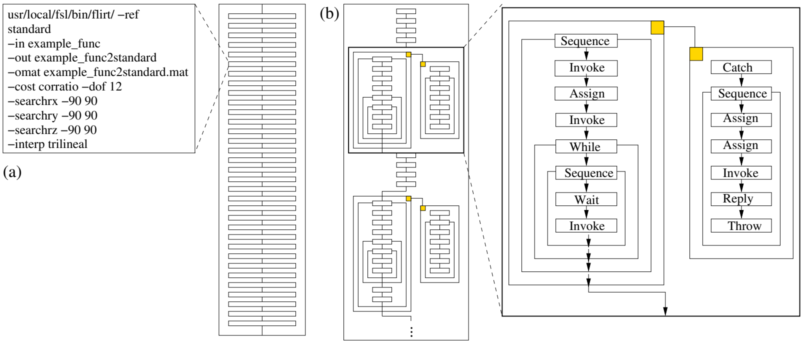
1 Introduction

Professionals in domains such as scientific computing, social-sciences, astronomy, neurosciences, and health-care are increasingly expected to compose heterogeneous computational entities to perform and automate their professional activities. Unlike professional programmers, these end users write programs to support the goals of their domains, where programming is a means to an end, not the primary goal [7]. However, studies have shown that such users spend about 40% of their time on programming activities [5], meaning that a large community of people are spending a lot of their time on programming tasks rather than on tasks directly related to their domain.

While in some cases end users may find it sufficient to use a single tool to accomplish their goals, very often one single tool may not provide all functionalities. Hence, the end users must compose functions from a number of tools, libraries, and APIs. To define such compositions, they need to either write glue code in the form of executable scripts, or use special-purpose tools that provide GUIs that generate such code, both of which require significant technical knowledge that they often lack.

Today, there is a large variety of approaches to support the composition of computational elements; however, they can be classified into two main categories: i) code scripts, and ii) orchestrations. However, neither of these fit naturally to the end users'

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 370–378, 2011.



```
usr/local/fsl/bin/flirt/ −ref
standard
−in example_func
−out example_func2standard
−omat example_func2standard.mat
−cost corratio −dof 12
−searchrx −90 90
−searchry −90 90
−searchrz −90 90
−interp trilineal
```

Fig. 1. Common modes of composition: (a) code scripts and (b) orchestrations

needs. For instance, a typical code script for neuroscience workflows (as shown in Fig. 1a) requires writing program calls to describe analyses. This not only requires knowledge of the scripting language, but also other technical details, e.g. the parameters used by each program call. Orchestration languages such as BPEL (as shown in Fig. 1b) offer an improvement over scripts by providing higher level constructs (e.g., services as opposed to command-line parameters). However, they too have a low level of abstraction, and are still close to program code. For instance, such BPEL scripts require specification of control logic (e.g. Sequence, While), data assignment (i.e. Assign) and error handling constructs (i.e. Throw). As can be seen, both approaches are too low level for technically naïve end users and therefore tedious and error-prone. For both these cases, detection of syntactic and semantic issues has to be performed manually. Although, at times GUIs and type-checkers aid syntactic verification, finding semantic issues that are more domain-specific is difficult because specifications written in terms of low-level code constructs are not convenient for describing semantic information. Additionally, the analysis of other relevant properties such as performance or deadlock is even harder to support on script code. This often leads to technically-nave end users resorting to opportunistic programming and copy-paste, wherein they make frequent mistakes [2]. In either case, creating compositions is difficult for end users because of:

- Complexity due to low-level details: Existing languages and tools require end users to have knowledge of a myriad of low-level technical detail such as parameters, file systems, paths, operating systems, etc.

- Lack of support for error resolution: Few mechanisms exist today for helping users detect syntactic and semantic problems with their compositions. Further, identifying and fixing quality attribute problems (such as security and privacy issues) in their specifications is difficult for end users.

- Conceptual mismatch: End users often think in terms of tasks they want to accomplish, while current composition mechanisms force them to think in terms of technology with which the task is implemented. For example, "Remove Image Noise" as opposed to calling the specific program(s) to perform this function.

We believe an architectural specification can alleviate the above problems by providing domain-specific abstractions that are close to the way that end users think about their problems, but that can still be mapped to code that can be executed on traditional platforms such as SOAs. In this paper, we propose how this can be achieved using architectural styles [14] that provide an abstract vocabulary of components (and the constraints that direct their usage) that can be used by end users to design compositions.

2 Design Approach

We propose a dataflow-based architectural style called SCORE (Simple Compositional ORchestration for End users) that can be used for assembling computations in various domains. SCORE provides a vocabulary that can be tailored for different domains and does not require writing low-level code. Instead of using directly executable scripts, we propose using multi-layered styles for representing workflows, where each layer handles different concerns. The use of such styles gives us leverage to use existing architectural analysis techniques to provide advice and guarantees to users about their compositions via various formal analyses. The end users can specify their compositions in terms of an assembly of high-level functions. These functions can be translated into lower-level orchestrations using tool support.

2.1 Using Architectural Styles as a Basis for Abstraction and Refinement

Software architecture provides the high-level structure of a system, consisting of components, connectors, and properties [14]. While it is possible to model the architecture of a system using such generic high-level structures, it is crucial to use a more specialized architectural modeling vocabulary that targets a family of architectures for a particular domain. This specialized modeling vocabulary is known as an architectural style [14] and it defines the following elements:

- **Component types:** represent the primary computational elements and data stores.

- **Connectors types:** represent interactions among components.

- **Properties:** represent semantic information about the components and connectors.

- **Constraints:** represent restrictions on the usage of components or connectors, e.g. allowable values of properties, topological restrictions.

Acme [1] is an architectural definition language (ADL) that provides support to define such styles. Acme's predicate-based type system allows styles to inherit characteristics from other styles. When a style element (or the style itself) inherits other elements, not only does it inherit the properties, but also the constraints defined on its usage. We find this characteristic of Acme useful for many of the problems that we discussed in Section 1. Specifically, for mapping a functional concept to its technical solution, styles that determine functional vocabulary can be inherited and refined to the styles that consist of components and implement them.

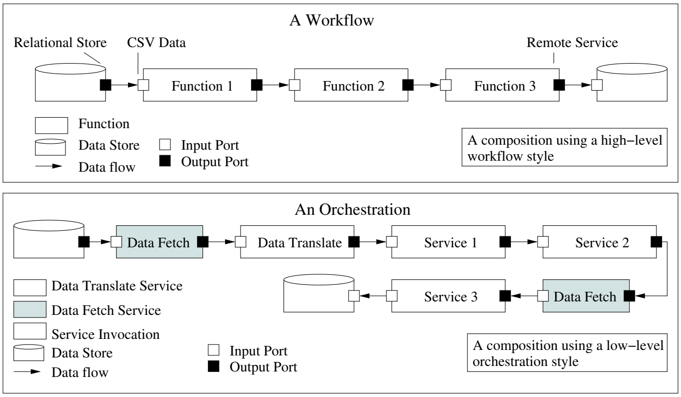


Fig. 2. An illustration of a mapping between a workflow to an orchestration style

For example, a high-level workflow, as in Fig. 2, can be mapped to a low-level orchestration if both of these are specified using architectural styles that follow inheritance relationships. The workflow in Fig. 2 composes three functions with different input and output data requirements and location constraints; its corresponding low-level orchestration includes services for individual functions and additional components for data translation and data fetching to compose a sound service orchestration. Note that this is not a 1-to-1 mapping between components, but it is derived from rules. For instance, the port properties of components DataStore and Function 1 in the workflow can point to a difference in data-type and location, leading to insertion of two components that can address the mismatch.

Although simple, this example gives a glimpse of how abstract models can be helpful to end users by providing just the necessary details allowing for a simpler end user specification. These abstract models can be translated into an executable specification using additional properties and constraints. SCORE is based on this approach, where end users can use a high-level style to compose functions that can be compiled into low-level orchestration. We call this functional composition an ‘end user orchestration’.

3 SCORE

SCORE is an architectural style that provides a restricted vocabulary for the specification of workflows in a dataflow like specification. It abstracts the specification of workflows to the essential types and the properties of concern that match the computation model required by (end user) scientific communities. The SCORE style specifies rules that are evaluated at design time, enforcing restrictions on the kinds of components users can compose. Writing these rules involves some degree of technical expertise, but these are associated with the architectural style, which is written once by a designer, and then used by end users for modeling workflows based on the style.

3.1 SCORE Vocabulary

Table 1 shows SCORE architectural types, functions and constraints that are used to specify workflows using SCORE. These constrants are based on Acme's first order predicate logic, where they are expressed as predicates over properties of the workflow elements. The basic elements of the constraint language include constructs such as conjunction, disjunction, implication and quantification. An important role of the SCORE style description is to define the meaning of semantic constructs in terms of the syntactic properties of the style elements. We achieve this, at least to a certain extent, by enforcing domain-specific constraints. These not only prohibit end users from creating inappropriate service compositions, but also promote soundness by ensuring feedback mechanism via marking errors when a component fails to satisfy any such constraints.

Table 1. SCORE composition elements

Components	Description
`DataStore`	Components for Data-access (such as file/SQL data-access)
`LogicComponent`	Components for conditional logic (such as join/split etc)
`Service`	Components that are executed as a service call
`Tool`	Components who's functionality is implemented by tools
`UIElement`	Special-purpose UI activity for human interaction
Connectors	Description
`DataFlowConnector`	Supports dataflow communication between the components.
`DataReadConnector`	Read data from a DataStore Component
`DataWriteConnector`	Write data to a DataStore Component
`UIDataFlowConnector`	Provides capabilities to interact with UIElements
Ports	Description
`configPort`	Provides an interface to add configuration details to components
`consumePort`	Represents data-input interface for a component.
`providePort`	Represents data-output interface for a component.
`readPort`	Provides data-read interface for DataStore component
`writePort`	Provides data-write interface for DataStore component
Roles	Description
`consumerRole`	Defines input interface to DataFlow/UIDataflow connectors
`providerRole`	Defines output interface to DataFlow/UIDataflow connectors
`dataReaderRole`	Defines input interfaces for the DataRead/DataWrite connectors
`dataWriterRole`	Defines output interfaces for the DataRead/DataWrite connectors
Acme Functions	Description
`Workflow.Connectors`	The set of connectors in a workflow
`ConnectorName.Roles`	The set of the roles in a connector
`self.PROPERTIES`	All the properties of a particular element
`size( )`	Size of a set of workflow elements
`Invariant`	A constraint that can never be violated
`Heuristic`	A constraint that should be observed but can be selectively violated
Constraint types	Example
`Structural`	Checking that connectors have only two roles attached `rule onlyTwoRoles = heuristic size(self.ROLES) = 2;`

`Structural`	Checking if a specific method of the service called exists
	`rule MatchingCalls = invariant forall request:`
	`!ServiceCallT in self.PORTS |exists response:`
	`!ServiceResponseTin self.PORTS|`
	`request.methodName == response.methodName;`

`Property`	Checking if all property values are filled in
	`rule allValues = invariant forall p in self.PROPERTIES`
	`| hasValue(p);`

`Membership`	Ensuring that a workflow contains only 2 types of components
	`rule membership-rule = invariant forall e: Component`
	`in self.MEMBERS |declaresType(e,ComponentTypeA) OR`
	`declaresType(e,ComponentTypeB);`

Table 2. Types of analyses

	STRUCTURAL ANALYSIS	**TYPE**
`Data Integrity`	Data-format of the output port of the previous connector matches the format of the input port	Predicate based
`Semantic correctness`	Membership constraints for having only limited component types are met	Predicate based
`Structural soundness`	All Structural constraints are met, and there are: - no dangling ports - no disconnected data elements	Predicate based
	DOMAIN-SPECIFIC ANALYSES	**TYPE**
`Security/Privacy Analysis`	Identify potential security/privacy issues based on rules	Program based
`Order Analysis`	Evaluate if ordering of two services makes sense	Program based

Properties and constraints on architectural elements can be used to analyze systems defined using SCORE. Table 2 displays some examples of analyses that are built using SCORE properties, such as analyzing a workflow for structural soundness, and various domain-specific analyses based on workflow properties. Some of the examples of such analyses written in Acme ADL are presented in [4]. The rules for these analyses are written as predicates that are analyzed for correctness while end users design workflows.

4 SCORE in Practice

As shown in Fig. 3, SCORE can be specialized to various domains through refinement and inheritance. This requires construction of sub-styles that extend the basic SCORE dataflow style by adding additional properties, domain-specific constraints, and rules that allow the correct construction of workflows within that domain. For our initial prototype we have defined sub-styles for a couple of domains - neuroscience and network analysis, that we use for modeling workflows in these two domains.

For the neuroscience domain, we experimented with using SCORE for defining workflows that can automate FMRI [1] data pre-processing steps for which neuroscientists currently write detailed code-scripts (as shown in Fig. 1a), replacing them with a tool-assisted workflow (shown in Fig. 4) that is based on SCORE type system. SCORE provides the basic functional vocabulary for constructing workflows, while the low-level styles extend this dataflow-based vocabulary to include additional details about how tools like FSL [2] execute these high-level functions. Thus, not only does SCORE help to define neuroscience workflows at a functional level, it supports analysis such as checking for ordering, and security based on various domain-specific constraints. Fig. 4 for instance, gives an example where one such analysis has gone wrong because of the inappropriate ordering of services in the defined workflow.

Similarly, SCORE was also used to model workflows for dynamic network analysis - a domain that involves creating network models from unstructured data, and then use those models to gain insight about social phenomena through analysis and simulation. This was primarily used for our large SOA based platform named SORASCS [13] that

[1] FMRI (functional magnetic resonance imaging) is a neuroimaging technique in the neuroscience domain to understand the behavior of the human brain.

[2] The FSL brain imaging tool-suite: www.fmrib.ox.ac.uk/fsl

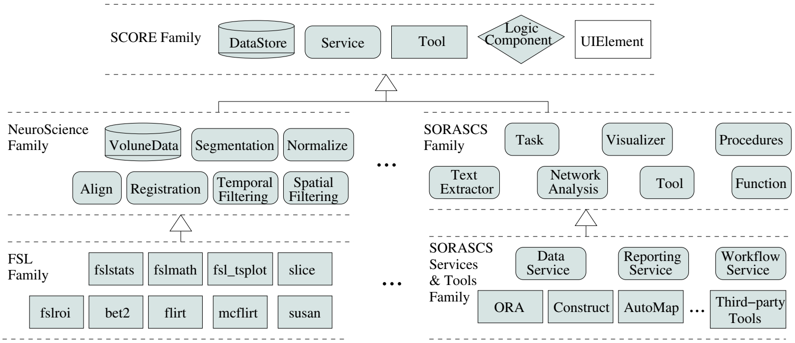
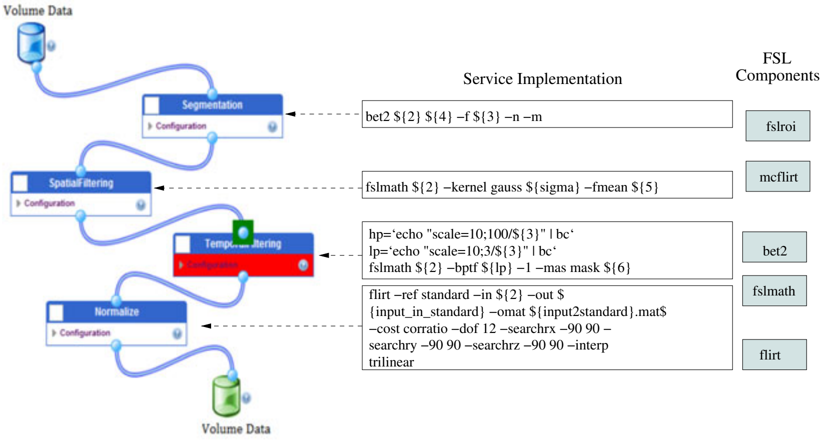


Fig. 3. Style derivation by inheritance



```
bet2 ${2} ${4} –f ${3} –n –m
```

```
fslmath ${2} –kernel gauss ${sigma} –fmean ${5}
```

```
hp=‘echo "scale=10;100/${3}" | bc‘
lp=‘echo "scale=10;3/${3}" | bc‘
fslmath ${2} –bptf ${lp} –1 –mas mask ${6}
```

```
flirt –ref standard –in ${2} –out $
{input_in_standard} –omat ${input2standard}.mat$
–cost corratio –dof 12 –searchrx –90 90 –
searchry –90 90 –searchrz –90 90 –interp
trilinear
```

Fig. 4. A pre-processing workflow with an ordering problem

provides an end user friendly SOA based platform to analysts to combine services from various tools in the intelligence analysis domain.

5 Related Work

SCORE can be characterized as providing an abstract vocabulary for composing computations, which can be analyzed for both syntactic and semantic errors, and reduces the conceptual mismatch between end user's functional vocabulary and low-level code constructs (required by current composition mechanisms). We use this characterization to compare SCORE with the related work.

Abstraction: UML-based languages like BPMN have been widely used for documenting abstract compositions. However, their primary use-case has been documentation and not execution. They do not support analysis, and when used to capture details tend to be extremely complicated [10]. There have been other efforts such as SAS language [3] for modeling functional and QoS requirements by Esfahani et al at, and MDA based approaches [9] for composition using SOA profiles. However, such ontologies and profiles don't scale and lack the capability to be extended across different domains. SCORE in comparison, supports functional composition that can be refined, and compiled to low level specifications enabling an easier composition.

Error resolution: Most of the current composition languages provide type-checkers for syntactic verification, but they lack capabilities to resolve domain-specific errors. Almost all such composition languages have a relatively fixed schema that don't allow adding additional attributes that can be useful for expressing domain-specific constraints. In particular, the focus of most of these approaches has been to analyze soundness [11], concurrency [8] or control flow errors [15]. In comparison, SCORE provides support for adding properties and constraints, allowing designers to write domain-specific analyses that other languages cannot support.

Conceptual mismatch: Domain-specific compositions have been used for various scientific dataflow languages such as SCUFL in Taverna [6], and LONI Pipeline [12] for neurosciences. However, most of these approaches have a fixed type system that cannot be extended or refined as we do in SCORE. One of the benefits of such refinement is that the same set of high-level styles can be extended to other domains - for instance, in our case dynamic network analysis, and neurosciences share a common model of computation, but have no similarity in terms of design concepts.

6 Conclusions and Future Work

In this paper we proposed an architectural-style based approach for service composition using an end user specification style called SCORE. The goal of SCORE is to address the requirements of end users who are primarily concerned with composition of computational elements and the analysis of the resulting compositions, but have limited technical expertise to write detailed code.

The tool support constructed using SCORE component types, allows the visual composition of tools, services and data that can be executed by a run-time platform. Although, style-based composition helps to constrain the usage of the component types, it is still a challenge to design an optimal type system for a domain; however, such an upfront investment by style designers could be helpful for end users who can use such a family of component types in their tools. As a future work, we would like to extend SCORE to other domains, and support new types of analyses. We are also working on the problem of mismatch repair given the domain-specific constraints. These would require generating alternative compositions based on the constraints of the styles.

Acknowledgments. This work was supported in part by the Office of Naval Research (ONR-N000140811223), and the FCT Portuguese Science and Technology Agency (under the CMU-Portugal faculty exchange program). Additional support was provided by the Center for Computational Analysis of Social and Organizational Systems

(CASOS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, or the U.S. government.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6, 213–249 (1997)
2. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In: Proc. of the 27th Int. Conf. on Human Factors in Computing Systems (CHI), pp. 1589–1598 (2009)
3. Esfahani, N., Malek, S., Sousa, J.P., Gomaa, H., Menascé, D.A.: A modeling language for activity-oriented composition of service-oriented software systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 591–605. Springer, Heidelberg (2009)
4. Garlan, D., Schmerl, B.: Architecture-driven modelling and analysis. In: Proc. of the 11th Australian Workshop on Safety Critical Systems and Software (SCS), pp. 3–17. Australian Computer Society, Inc., Darlinghurst (2006)
5. Howison, J., Herbsleb, J.D.: Scientific software production: Incentives and collaboration. In: Proc. of ACM CSCW, pp. 513–522 (March 2011)
6. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: A tool for building and running workflows of services. Nucleic Acids Research 34 (Web Server Issue), W729–W732 (2006)
7. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothemel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. ACM Comput. Surv. 43, 21:1–21:44 (2011)
8. Koshkina, M., van Breugel, F.: Modelling and verifying web service orchestration by means of the concurrency workbench. SIGSOFT Software. Engineering Notes 29, 1–10 (2004)
9. Mayer, P., Schroeder, A., Koch, N.: MDD4SOA: Model-driven service orchestration. In: Proc. of the 2th Int. IEEE Enterprise Distributed Object Computing Conference, pp. 203–212. IEEE Computer Society, Los Alamitos (2008)
10. Ossher, H., Bellamy, R.K.E., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M., Desmond, M., de Vries, J., Fisher, A., Krasikov, S.: Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges. In: OOPSLA, pp. 848–864 (2010)
11. Puhlmann, F., Weske, M.: Interaction soundness for service orchestrations. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 302–313. Springer, Heidelberg (2006)
12. Rex, D.E., Ma, J.Q., Toga, A.W.: The loni pipeline processing environment. Neuroimage 19, 1033–1048 (2003)
13. Schmerl, B., Garlan, D., Dwivedi, V., Bigrigg, M., Carley, K.M.: SORASCS: A case study in SOA-based platform design for socio-cultural analysis. In: Proc. of the 33rd Int. Conf. on Software Engineering (ICSE), pp. 643–652 (2011)
14. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
15. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000)

Using Model Transformation Techniques for the Superimposition of Architectural Styles

Elena Navarro[1], Carlos E. Cuesta[2], Dewayne E. Perry[3], and Cristina Roda[1]

[1] Computing Systems Department, University of Castilla-La Mancha
enavarro@dsi.uclm.es, cristinarodasanchez@gmail.com
[2] Dept. LSI2 (Computing Languages and Systems II), Rey Juan Carlos University
carlos.cuesta@urjc.es
[3] Electrical and Computer Engineering Department, The University of Texas at Austin
DewaynePerry@engr.utexas.edu

Abstract. Software Architecture is a key artifact in the software development process, as it provides a bridge between the requirements of the system-to-be and its final design. Architectural description is therefore a critical step, which can be assisted by the use of Architectural Styles. Styles make it possible to reuse architectural knowledge by providing guidelines for its description, and by constraining the configuration and behavior of the target system. The architect must superimpose these constraints, but this could be an error-prone task unless some kind of automatic support is provided. Therefore, this paper presents a proposal that generates proto-architectures by superimposing architectural styles on the initial requirements' operationalization, using model-to-model (M2M) transformation techniques. Our proposal includes a tool called MORPHEUS, which applies QVT as the transformation language; a real-world example is provided to explain how the superimposition process works, and how the QVT language is used to express these style-based transformations.

Keywords: architectural style, model-driven development, architectural description, model transformations.

1 Introduction

The specification of the *Software Architecture* (SA) is always a challenging activity. It is a decision making process that establishes strict compromises at the architectural level. These compromises must be reached in order to elaborate a specification that is able to meet both functional and non-functional requirements. In this context, the proper use of *Architectural Styles* can be a great asset for the process.

According to Perry & Wolf [15], an architectural style is something that "abstracts elements and formal aspects from various specific architectures". This definition is deliberately open; it can be used to describe full systems, or just a specific aspect of the architecture at hand [16], which can be composed to others. In addition, a style may also specify constraints on those elements and/or formal aspects, [18] as well as on the system behavior. Thus, they can affect the configuration of the architecture.

However, to the best of our knowledge, there are very few proposals that clearly establish how the architectural styles can be automatically or semi-automatically used

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 379–387, 2011.

to describe the SA. Many papers have focused their efforts on the classification of Architectural Styles, or their evaluation; but hardly any of them have focused on their automatic application. Our proposal is, then, a system to automatically superimpose the mandatory constraints of an architectural style on top of the system-to-be.

This work is structured as follows. After this introduction, section 2 presents a brief introduction to the methodological context in which this proposal has been elaborated, along with a case study developed for validation purposes. Section 3 describes the way in which our proposal introduces architectural styles into the generation of the proto-architecture [1], using model transformation techniques. Finally, section 4 discusses related work and section 5 presents the conclusions.

2 Context of the Work

During the description of the SA, the architect should weigh the impact of its decisions at the architectural level and the relationships they have with other decisions and requirements before realizing them in the system-to-be. The ATRIUM methodology [10] provides support in this context. It has been designed using the *Model-Driven Development* (MDD) approach and encompasses these activities:

- *Modelling Requirements*. This activity allows the analyst to identify and specify the requirements of the system-to-be by defining the *ATRIUM goal model* guiding the architect from *goals* that the systems should achieve, till *requirements* that the system should meet, and *operationalizations* that describe solutions to the established *requirements*.
- *Modelling Scenarios*. This activity focuses on the identification of the set of scenarios that defines the system's behaviour under certain architectural decisions, described in the ATRIUM goal model as *operationalizations*.
- *Synthesize and transform*. This activity has been defined to generate the *proto-architecture* [1] of the specific system. It synthesizes the architectural elements that make up the system, as well as the structure of the future system, from the ATRIUM scenario model.

They must be iterated over in order to define and refine the different models and allow the architect to reason about both the requirements and the architecture. One of the inputs for the *Synthesize and transform* activity is the selected Architectural Style. It must be taken into account that this Architectural Style imposes constraints in terms of the structure and the behaviour of the final description. For this reason, this activity must apply these constraints automatically, conforming to them during the generation of the architecture, hence avoiding a task that could be cumbersome and error-prone for the architect, if it was performed by hand.

ATRIUM has been validated in a case study that is associated with the European project *Environmental Friendly and cost-effective Technology for Coating Removal* (EFTCoR) [7]. The goal of this project is to design a family of robotic systems capable of performing maintenance operations for ship hulls. The Robotic Devices Control Unit (RDCU) integrates all the required functionality to manage the EFTCoR. We have focused our efforts in its SA because of the strict constraints that have to be satisfied in terms of safety, performance, and reliability.

3 Using M2M Transformations for Architectural Styles

As presented in section 2, the architectural elements, their behavior, and the structure of the proto-architecture (the output of the activity *Synthesize and Transform*) are synthesized from the scenario model considering the constraints imposed by the selected architectural style.

Several languages have been proposed to define M2M transformations. Several surveys, such as that presented by Czarnecki and Helsen [4] identify the features that a M2M transformation language should satisfy in order to fulfill the MDD approach. Considering these, QVT Relations [12] was selected as our M2M transformation language because it provides facilities to manage source and target model; it defines an *incrementality* feature, i.e. it is able to update the generated model according to the changes in the source model; and it offers *directionality* and *tracing*.

The *Rules Organization* feature of QVT has been used to classify the different transformations. Specifically, they have been catalogued as follows:

- *Architectural Generation patterns*. They describe those transformations that are applicable to most of the existing architectural metamodels because they are focused on the generation of components, connectors and systems.
- *Idioms.* They describe low-level transformations that are specific to an architectural metamodel. We have made this distinction in order to facilitate the generation of the proto-architecture according to the architectural metamodel selected by the architect.
- *Architectural Styles*. These transformations are oriented to the application of the constraints imposed by the Architectural Style selected during the activity *Modelling Requirements*.

Therefore, using the same scenario model, different proto-architectures can be generated, depending on the selected architectural metamodel, by applying its specific idioms, and the selected Architectural Style.

Table 1. Head of the transformation *ScenariosToArchmodelPRISMA*, which generates PRISMA proto-architectures from ATRIUM scenario models

```
import archpatt;
import archAcrosetStyle;
transformation ScenariosToArchmodelPRISMA (scenarios: ATRIUMScenarios,
archmodel: Archmodel)
   key archmodel::System {name};
   key archmodel::Component {name};
   key archmodel::Connector {name};
   key archmodel::Port {name, ArchitecturalElement};
   key archmodel::Attachment {name, System};
   key archmodel::Attachment {name, Architecturalmodel};
```

Table 1 shows the head of the transformation in charge of generating proto-architectures from ATRIUM scenario models. It can be seen that the *Architectural Generation patterns* and the A*rchitectural Style patterns* are imported. Table 1 also illustrates the declaration of the transformation, by means of a name (i.e. *ScenarioToArchmodelPRISMA*) and the identification of the candidate models. There are two candidate models: *scenarios,* which represents a candidate model that conforms to the ATRIUM Scenario metamodel; and *archmodel,* which represents a candidate model that conforms to the selected architectural metamodel to be instantiated. Specifically, we have used the *ATRIUM Scenarios* and *PRISMA* [14] metamodels to execute the transformations. In addition, the transformation can define *keys* to uniquely identify the elements and avoid duplicate instances.

In the following, we describe in greater detail the process to perform the automatic superimposition of Architectural Styles, by dealing with the constraints imposed by them in a generative way.

During the *Modelling Requirements* activity, the Architectural Style is selected; and this means that several constraints must be satisfied. The main idea of this proposal is to transform these constraints into QVT generation rules so that the generated proto-architecture is compliant with the selected Architectural Style.

The EFTCoR system uses a Layered Style; specifically, we used ACROSET [13], which is a *Domain Specific Software Architecture* (DSSA) that specializes the Layered Style identifying three kinds of subsystems (SUC, MUC, and RUC). These styles specify some constraints in terms of compositionality, by establishing that one layer only requires the services provided by the lower layer.

In order to generate the proto-architecture of the EFTCoR taking into account the constraints imposed by the ACROSET Style, a transformation was defined (see in Table 1 as *archAcrosetStyle*). A transformation in QVT is defined by means of a set of *relations* that must hold in order to apply successfully the transformation. *ApplyingACROSET2Systems* is one of these relations (see Fig. 1).

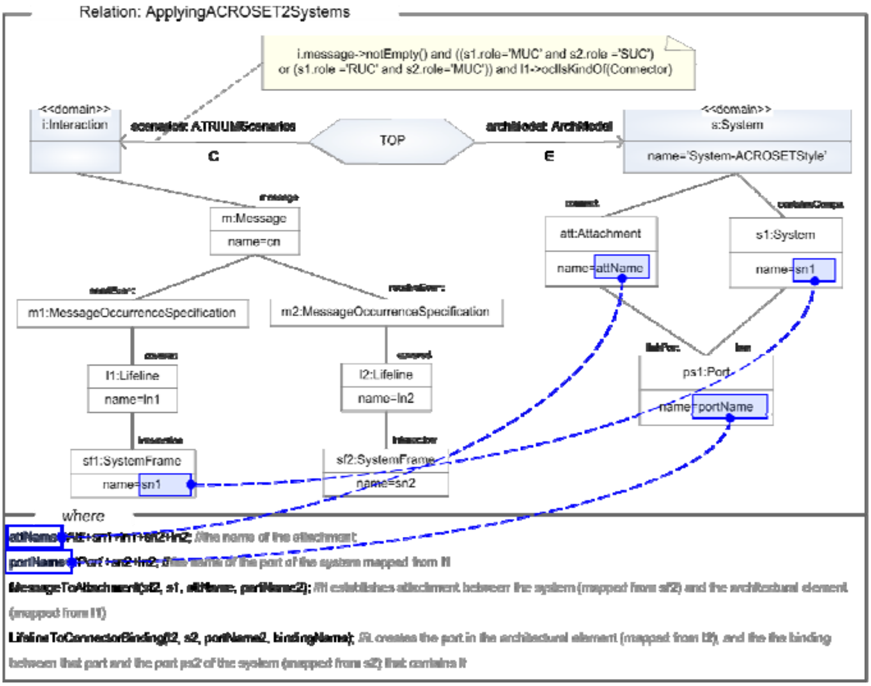

Relation: ApplyingACROSET2Systems
i.message->notEmpty() and ((s1.role='MUC' and s2.role ='SUC')
or (s1.role ='RUC' and s2.role='MUC')) and l1->oclIsKindOf(Connector)
<<domain>>
i:Interaction
scenarios: ATRIUMScenarios
TOP
archModel: ArchModel
C
E
<<domain>>
s:System
name='System-ACROSETStyle'
message
m:Message
name=cn
attach
att:Attachment
name=attName
containsComps
s1:System
name=sn1
sendEvent
receiveEvent
m1:MessageOccurrenceSpecification
m2:MessageOccurrenceSpecification
covered
covered
l1:Lifeline
name=ln1
l2:Lifeline
name=ln2
linkFor
has
ps1:Port
name=portName
interaction
interaction
sf1:SystemFrame
name=sn1
sf2:SystemFrame
name=sn2
where
attName='Att'+sn1+ln1+sn2+ln2; //the name of the attachment
portName='Port'+sn2+ln2; //the name of the port of the system mapped from l1
MessageToAttachment(sf2, s1, attName, portName2); //it establishes attachment between the system (mapped from sf2) and the architectural element (mapped from l1)
LifelineToConnectorBinding(l2, s2, portName2, bindingName); //it creates the port in the architectural element (mapped from l2), and the the binding between that port and the port ps2 of the system (mapped from s2) that contains it

Fig. 1. QVT relation to superimpose the ACROSET Style

The graphical syntax of QVT has been used to enhance the legibility. The hexagon in the middle helps us to represent the transformation by identifying the candidate models (in this case, *scenarios* and *archmodel*) along with their respective metamodels (*ATRIUMScenarios* and *Archmodel*). Each arrow represents a *domain* that is labeled as either *C* or *E* to determine if the transformation is executed in *checkonly* mode (it will only be checked if a valid matching exists that satisfy the relation) or in *enforce* mode (provided the matching fails, the target model will be modified to satisfy the relationship), respectively, in that direction. This allows the architect to either generate the proto-architecture or check whether inconsistencies between the proto-architecture and the scenario model arise.

In addition, whenever a relation is defined, clauses *where* and *when* can be defined. The *where* clause (see Fig. 1) specifies a condition that must be held by all the elements involved in the relation so that it can be successfully applied.

It is necessary to establish the roles played by the elements of the scenario model with regard to the Architectural Style. For this reason, the *role* attribute is defined to be used by the OCL expression in the *scenarios* domain (see again Fig. 1) to ensure that the relation is only applied when an interaction happens between elements that belong to the appropriate layers. This means that any interaction between other layers will not cause any sort of generation on the proto-architecture.

Fig. 2 shows an example of the result of this relation when it is applied on an ATRIUM Scenario. At the top of the figure, there is a scenario which establishes how should be the collaboration between the architectural and environmental elements, to meet the requirement “REQ.6” according to the operationalization “OPE.13”.

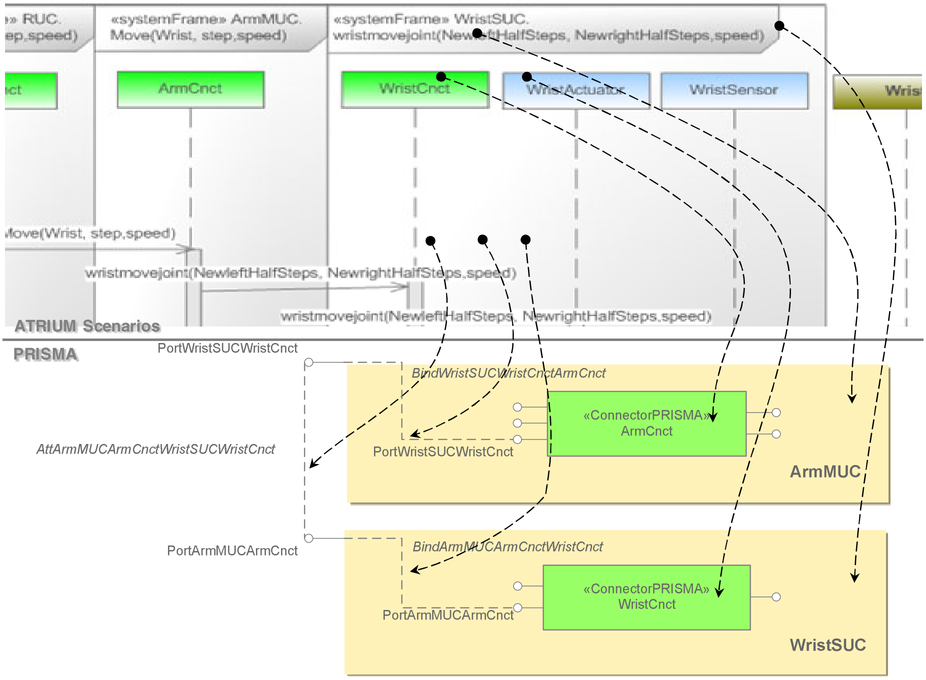
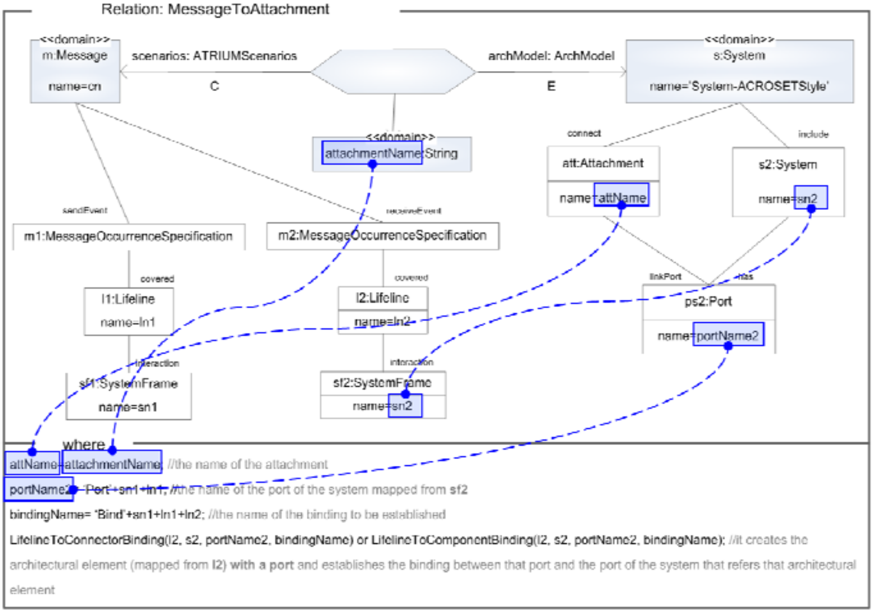


Fig. 2. ATRIUM Scenario (atop) and the generated proto-architecture

As shown in Fig. 1, a matching is established between the *SystemFrame* and the *System* because both of them bind their attribute "*name*" to the same variable. This means that when the Relation is applied to the Scenario in Fig. 2, a PRISMA *System* named *ArmMUC* is created in the PRISMA model, because the *systemFrame ArmMUC* contains the *Lifeline ArmCnct*. Both a *Port* and an *Attachment* are resolved in the *where* clause, and also created in the architectural model. These two elements are related, in order to define the connection between the System *s1*, and the System *s2* that will be created using the relation *MessageToAttachment* (see Fig. 3).

The Relation *MessageToAttachment* (Fig. 3) acts in a similar way to *ApplyingACROSET2Systems* (Fig. 1), generating the System *s2* from systemFrame *sf2* by matching their names, and attaching this to the other generated System, *s1*.



```
attName=attachmentName; //the name of the attachment
portName2='Port'+sn1+ln1; //the name of the port of the system mapped from sf2
bindingName= 'Bind'+sn1+ln1+ln2; //the name of the binding to be established
LifelineToConnectorBinding(l2, s2, portName2, bindingName) or LifelineToComponentBinding(l2, s2, portName2, bindingName); //it creates the architectural element (mapped from l2) with a port and establishes the binding between that port and the port of the system that refers that architectural element
```

Fig. 3. QVT Relation: Establishing connections between Systems

In the *where* clause of both the relations *ApplyingACROSET2Systems* and *MessageToAttachment*, another relation, *LifelineToConnectorBinding*, is specified. This relation helps to apply the constraints of the ACROSET Style, that is, the compositionality of the *Architectural Elements* belonging to the different layers. Due to space constraints, no more details are provided about this relation.

In summary, we have been able to generate the proto-architecture with only one ATRIUM scenario. The *incrementality* feature of QVT Relations makes possible to automatically update the proto-architecture as new scenarios are defined or exiting ones are modified. Finally, it should be noted that the ATRIUM development process is fully supported by a toolset, called MORPHEUS [11].

4 Related Work

Architectural Styles have received significant attention from both academia and industry. Preliminary work in the field designed software structures specialized for specific domains, such as avionics or missile control and command [5]. This initial work on the definitions of Architectural Styles is one of the cornerstones of SA.

However, most proposals on Architectural Style have focused their attention on recurrent issues. Most of the work has been oriented towards the description of new architectural styles. Some work has focused on the classification of architectural styles, and other proposals have tried to define primitives to describe and/or compose architectural styles [9] [19]. But, as far as we know, no proposals have paid attention to their automatic superimposition. This is the reason why, in the following, we focus our analysis on proposals for *model transformations* in the area of SA.

There are some proposals which actually intend to generate software architectures. The proposal by Bruin and van Vliet [2] describes a process for the generation of SA taking as inputs both a rich Feature-Solution graph and *Use Case Maps* (UCM). This proposal introduces architectural styles as a decision in the solution space, along with decision fragments. However, they do not deal with the automatic superimposition of the architectural style, nor they provide details about how this generation proceeds.

Castro et al. [3] have defined a methodology called TROPOS to guide the process of system specification from early requirements. Requirements are elicited with the *i** framework. The methodology proposes a refinement process from requirements to the SA. However, the (agent-oriented) architectural style is applied by hand.

To the best of our knowledge, the work presented by Sanchez et al. in [17] is the only using a similar generative perspective to ours. They present a process that combines MDD and AOSD to derive Aspect-Oriented Architectures from Aspect-Oriented Requirements models. Once the set of scenarios have been defined, they are transformed into an Aspect-Oriented Architecture by means of a set of transformation rules specified using QVT [12]. However, this proposal pays no attention to the superimposition of architectural styles.

ATRIUM faces many of the issues exhibited by these proposals. Architectural Styles are selected according to the specific needs of the system-to-be, and are automatically applied by means of M2M transformations. Also, it has been designed to make possible the use of that different architectural metamodels.

5 Conclusions and Future Work

ATRIUM aims at generating the proto-architecture of the system-to-be by means of a M2M transformation problem. QVT emerged as the best solution for this purpose, as it satisfies most of the requirements.

By using QVT Relations, a set of transformations has been defined to generate the proto-architecture from ATRIUM scenario models. It provides several advantages. The first is related to its applicability to the whole set of scenarios. QVT supports the definition of keys, which guarantee that the synthesis process prevents the creation of duplicated objects. In addition, it is not necessary to provide the entire set of scenarios to generate the proto-architecture. In fact, the generation can proceed with only one scenario. Thanks to QVT's *incrementality* feature, the generated proto-architecture can be automatically updated as new scenarios are defined to be used.

One of the main concerns in the definition of ATRIUM is traceability. Top-down traceability is provided because the proto-architecture is generated automatically by establishing the appropriate transformations. Bottom-up traceability can be achieved because QVT Relations derives a *Trace Class* from each relation in order to generate traceability mappings. This ability is highly meaningful because a mapping is established between every element in the proto-architecture and its related element/s in the ATRIUM Scenarios model.

Another challenge to be faced by our proposal is how to establish mechanisms to evaluate the proto-architecture being obtained. We are currently focusing on how to detect faults while the proto-architecture is being generated. Early detection of such faults will make a meaningful improvement in the development in terms of both quality and cost. The definition of an *evaluation model* that describes potential faults at the specification level would be a first step in this direction.

Acknowledgements. This work has been partially supported by grants (PEII09-0054-9581) from the JCCM, and also by grants (TIN2008-06596-C02-01, TIN2009-13838, and CONSOLIDER CSD2007-022) from the Spanish Ministry of Science. Professor Perry is supported in part by NSF CISE Grants IIS-0438967 and CCF-0820251.

References

[1] Brandozzi, M., Perry, D.E.: Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. In: Workshop from Soft. Req. to Arch., pp. 54–61 (2001)

[2] de Bruin, H., van Vliet, H.: Quality-Driven Software Architecture Composition. Journal of Systems and Software 66(3), 269–284 (2003)

[3] Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Software Development Methodology: The Tropos Project. Information Systems 27(6), 365–389 (2002)

[4] Czarnecki, K., Helsen, S.: Classification of model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)

[5] Delisle, N., Garlan, D.: Formally specifying electronic instruments. In: 5th Int. Workshop on Software Specification and Design, pp. 242–248. ACM, New York (1989)

[6] Garlan, D., Shaw, M.: An introduction to software architecture. In: Advances in Software Engineering and Knowledge Engineering, vol. 2, pp. 1–39 (1993)

[7] GROWTH G3RD-CT-00794, EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project, 5th Framework Program, Spain (2003)

[8] Medini QVT Relations (2008), `http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77&lang=en`

[9] Mehta, N.R., Medvidovic, N.: Composing architectural styles from architectural primitives. ACM SIGSOFT Software Engineering Note 28(5), 347–350 (2003)

[10] Montero, F., Navarro, E.: ATRIUM: Software Architecture Driven by Requirements. In: 14th IEEE Int. Conf. on Eng. of Complex Computer Systems (ICECCS 2009), Potsdam, Germany, June 2-4 (2009)

[11] Navarro, E., Gómez, A., Letelier, P., Ramos, I.: MORPHEUS: a supporting tool for MDD. In: 18th Int. Conf. on Information Systems Development (ISD 2009), Nanchang, China, September 16-19 (2009)

[12] OMG doc. ptc/05-11-01, QVT, MOF Query/Views/Transformations final adopted specification (2005)

[13] Ortiz, F.J., Alonso, D., Álvarez, B., Pastor, J.A.: A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle. In: 10th Ada-Europe Int. Conf. on Reliable Software Technologies, pp. 13–24. Springer, Heidelberg (2005)

[14] Pérez, J., Ali, N., Carsí, J.Á., Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: 9th Int. Symp. on Component-Based SE, pp. 123–138 (2006)

[15] Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)

[16] Perry, D.E.: Generic Architecture Descriptions for Product Lines. In: van der Linden, F.J. (ed.) Development and Evolution of Software Architectures for Product Families. LNCS, vol. 1429, pp. 51–56. Springer, Heidelberg (1998)

[17] Sánchez, P., Moreira, A., Fuentes, L., Araújo, J., Magno, J.: Model-driven development for early aspects. Information & Software Technology 52(3), 249–273 (2010)

[18] Shaw, M., Garlan, D.: Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, School of Comp. Science, CMU (December 1994)

[19] Zdun, U., Avgeriou, P.: A catalogue of architectural primitives for modeling architectural patterns. Information & Software Technology 50(9-10), 1003–1034 (2008)

DAMASCo: A Framework for the Automatic Composition of Component-Based and Service-Oriented Architectures*

Javier Cubo and Ernesto Pimentel

Dept. Computer Science, University of Málaga, Málaga, Spain
{cubo,ernesto}@lcc.uma.es

Abstract. Although the reuse of software entities has matured in recent years, it has not become standard practice yet, since reusing component-based or service-oriented architectures requires the selection, composition, adaptation and evolution of prefabricated software parts. Recent research approaches have tackled independently the discovery, composition, adaptation or monitoring processes. We present the DAMASCo architecture, a framework for composing pre-existing services and components. Using model transformation, context-awareness, semantic matchmaking, behavioural compatibility, dependency analysis, and fault tolerance, DAMASCo focuses on discovering, adapting and monitoring the composition of context-aware services and components in mobile and pervasive systems. DAMASCo is made up of a set of tools that implement the different processes, which have been evaluated on several examples.

Keywords: Service-Oriented Architectures, Component-Based Framework, Composition, Adaptation, Ontology-Based Discovery, Monitoring.

1 Introduction

The increased usage of mobile and portable devices has given rise over the last few years to a new market of mobile and pervasive applications. These applications may be executed on either mobile computers, or wireless hand-held devices, or embedded systems, or even on sensors or RFID tags. Their main goal is to provide connectivity and services at all time, adapting and monitoring when required and improving the user experience. These systems are different to traditional distributed computing systems, as they are sensitive to their context (location, identity, time and activity) [20], by being needed to adapt their behaviour at run-time according to environment changing conditions, as well as those of user preferences or privileges. To reduce efforts and costs, context-aware systems may be developed using existing *Commercial-Off-The-Shelf* (COTS) components or (Web) services. In contrast to the traditional approach in which software systems

* Work partially supported by projects TIN2008-05932 ReSCUE funded by Spanish Ministry of Science and Innovation (MICINN) and FEDER, P07-TIC-3131 funded by Andalusian Government, and FP7-256980 NESSoS funded by European Union.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 388–404, 2011.

are implemented from scratch, COTS and services can be developed by different vendors using different languages and platforms. Component-Based Software Engineering (CBSE) [41] and Service-Oriented Architecture (SOA) [21] promote software reuse by selecting and assembling pre-existing software entities (COTS and services, respectively)[1]. Thus, these software development paradigms allow building fully working systems as efficient as possible from an architectural point of view to improve the software reusability. Although the reuse of software has matured, it has not become standard practice yet, since reusing components or services requires the selection, composition, adaptation and evolution of prefabricated software parts, then it is a non-intuitive process.

Recent research approaches have tackled independently the discovery, composition, adaptation or monitoring, as it will be discussed in Section 5. This paper presents the DAMASCo (**D**iscovery, **A**daptation and **M**onitoring of Context-**A**ware **S**ervices and **Co**mponents) architecture and its dynamics. DAMASCo is a framework for automatically composing pre-existing services and components. Using model transformation, context-awareness, semantic matchmaking, behavioural compatibility, dependency analysis, and fault tolerance, the framework focuses on discovering, adapting and monitoring the composition of context-aware services and components in mobile and pervasive systems. DAMASCo is made up of a set of tools that implement the different steps in the composition process, evaluated on several examples. The contributions of DAMASCo are the following: (i) an interface model based on extended transition systems by considering context information and semantic representation, and conditions to control executions according to changes, (ii) a model transformation process that extracts transition systems from component-based or service-oriented architectures (interfaces) implemented in different existing languages and platforms, and vice versa, (iii) a service discovery process performing semantic matchmaking based on ontologies to compare contexts and operations, protocol compatibility, and service ranking, (iv) a verification model based on symbolic model checking to verify our interface model, (v) a service composition and adaptation process to automatically obtain an adaptation contract and an adaptor solving mismatch at different interoperability levels, and (vi) a service monitoring process to establish priorities between data dependencies in concurrent executions, and to offer services as faults or changes occur.

The paper is organised as follows. Section 2 introduces motivations and foundations of our proposal. Section 3 presents the DAMASCo architecture and details the dynamics of the framework tool support. In Section 4 some experimental results are shown. Section 5 compares DAMASCo to related works. Finally, in Section 6 some conclusions are drawn and plans for future work are outlined.

2 Motivations and Foundations

In this section, we motivate our approach and explain its foundations.

[1] In the sequel, we use the terms component and service indistinctly.

2.1 Problem Statement

In current mobile and pervasive systems, services are everywhere. As aforementioned, component-based and service-oriented systems are developed from the selection, composition and adaptation of pre-existing software entities.

Service discovery can be defined as the ability to find out the most suitable services for a client's request. In industrial platforms, the current service technology, XML-based SOA via SOAP, WSDL, and UDDI, only supports queries based on keywords and category. This may bring about low-precision results, as it is neither able to discover semantic capabilities of services nor be adapted to a changing environment without human intervention. However, different context and semantic information is used in real-world services to improve their features. Therefore, it is essential to consider context information in the discovery of services deployed as part of mobile and pervasive applications. But current programming technology offers only very weak support for developing context-aware applications. Furthermore, one of the main challenges in CBSE and SOA is to provide semantic representation instead of only a syntactic one. W3C recommends the use of OWL-S[2] to capture the semantic description of services by means of *ontologies*, a formal representation of a set of concepts within a domain by capturing the relationships between those concepts. Then, service discovery based on semantic matching is required in mobile and pervasive systems.

Services are checked to verify that are free of inconsistencies due to the contextual nature of the system. Once services are checked, they can be composed. However, while composing pre-existing components or services in mobile and pervasive systems, different issues related to faults or changes arise dynamically, and they have to be detected and handled. These issues can be classified into four main categories [24]: (i) mismatch problems, (ii) requirement and configuration changes, (iii) network and remote system failures, and (iv) internal service errors. The first refers to the problems that may appear at different interoperability levels (*i.e.*, signature, behavioural or protocol, quality of service, and semantic or conceptual levels), and the Software Adaptation paradigm tackles these problems in a non-intrusive way [7]. The second is prompted by continuous changes over time (new requirements or services created at run-time), and Software Evolution (or Software Maintenance) focuses, among other issues, on solving them in an intrusive way [34]. The third and fourth are related to networks (network connection break-off or remote host unavailable) and services (suspension of services during the execution or system run-time error) failures, respectively. Both are addressed by Fault Tolerance (or Error Recovery) mechanisms [42].

Unfortunately, in most cases it is impossible to modify services in order to adapt them. Thus, due to the black-box nature of the components or services, they must be equipped with external interfaces giving information about their functionality. Interfaces do not always fit one another and some features of services may change at run-time. Therefore, services require a certain degree of adaptation and monitoring in order to avoid mismatch and faults during the composition. Mismatch situations may be caused, for instance, when message

[2] `http://www.w3.org/Submission/OWL-S`. Accessed on 30 March 2011.

names do not correspond, the changes of contexts are not controlled, the semantic description of exchanged information is not considered, the order of messages is not respected, or a message matches with several other messages.

Current industrial platforms only provide some means to describe components or services at their signature level (*e.g.*, CORBA's IDL[3]). However, mismatch may occur at the four interoperability levels, and most of the time at the behavioural one, due to an incompatibility in the order of the exchanged messages between components or services, which can lead to deadlock situations. Therefore, it would be desirable to address adaptation at all four levels together.

Finally, the ability to automatically monitor the service composition is an essential step to substantially decrease time and costs in the development, integration, and maintenance of complex systems. Service monitoring is able to detect violations of expected behaviours or to collect information about service executions at run-time, and to trigger an appropriate handling of such a failure. Run-time monitoring of functional and non-functional behaviour is becoming an important and researched topic in the field of service-based systems, by expressing and verifying properties by means of rules, specification languages, temporal logic or event calculus. However, only a few works have tackled monitoring of the handling of service concurrent interactions through data dependency analysis by considering context changes. In addition, there exist error recovery techniques for run-time monitoring of handling faults. The choice of fault tolerance mechanisms to be exploited for the development of dependable systems depends on the fault assumptions and on the system's characteristics and requirements.

2.2 Foundations of the Architectural Model

DAMASCo focuses on discovery, adaptation and monitoring related to context-aware mobile and pervasive systems, where devices and applications dynamically find and use components and services from their environment. These systems constitute enterprise applications increasingly developed using COTS or services.

On one hand, COTS component middleware can be checked out from a component repository, and assembled into a target software system. Component middleware encapsulates sets of services in order to provide reusable building blocks that can be used to develop enterprise applications more rapidly and robustly than those built entirely from scratch. There are many examples of COTS component middleware, such as the CORBA or COM/DCOM[4].

On the other hand, a service-oriented architecture is a set of components which can be invoked and whose interface descriptions can be published and discovered. The goal of SOA is to achieve loose coupling among interacting services. In computing, an Enterprise Service Bus (ESB) [14] provides foundational services for more complex architectures via an event-driven and standards-based messaging engine. Although ESB is commonly believed to not be necessarily web-services based, most ESB providers now build ESBs to incorporate SOA principles and

[3] http://www.omg.org/spec/CORBA/. Accessed on 30 March 2011.

[4] http://www.microsoft.com/com. Accessed on 30 March 2011.

increase their sales, *e.g.*, Business Process Execution Language (BPEL) [1] or Windows Workflow Foundation (WF) [38]. WF belongs to the .NET Framework 3.5, which is widely used in many companies and increasingly prevalent in the software engineering community [46].

Considering that services are a kind of software component, and even they may be generated from components [43], we assume our framework deals with components as services. Therefore, we focus on SOA to detail the set of principles of governing concepts used during phases of systems development and integration, and provided by this approach. It is obvious that SOA foundations are necessary and beneficial to the industry. However, SOA needs to be more agile and easy to model service applications. Modelling techniques, designing architectures, and implementing tools to support adaptation and evolution of the dynamic architectural aspects in these systems represent new challenges in this research field, by overcoming the limitations of existing Architectural Description Languages (ADLs) [31] with respect to capture the business aspects of service-oriented systems [22]. So far, it is difficult for business experts to model and verify their business processes. To address this, we use a model-based service-oriented architecture approach that makes the design, development, and deployment of processes more agile. Figure 1 shows the SOA layered architecture using model-based techniques (based on IBM's SOA Solution Stack, S3 [3]).

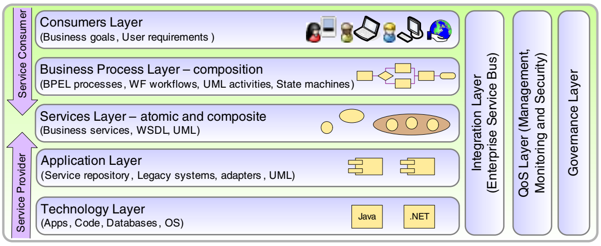

Service Consumer
Consumers Layer
(Business goals, User requirements)
Business Process Layer – composition
(BPEL processes, WF workflows, UML activities, State machines)
Services Layer – atomic and composite
(Business services, WSDL, UML)
Service Provider
Application Layer
(Service repository, Legacy systems, adapters, UML)
Technology Layer
(Apps, Code, Databases, OS)
Java
.NET
Integration Layer
(Enterprise Service Bus)
QoS Layer (Management, Monitoring and Security)
Governance Layer

Fig. 1. A SOA layered architecture using model-based techniques

We use a model-based methodology because it is the unification of initiatives that aim to improve software development by employing high-level, domain specific, models in the implementation, integration, maintenance, and testing of software systems. This technique refers to architecture and application development driven by models that capture designs at various levels of abstraction, being our model transformation process independent of the implementation. Model transformation provides a systematic and uniform view of incremental software development, making it easier to express and reason about adaptation and evolution. Since models tend to be represented using a graphical notation, the Model-Driven Architecture (MDA)[5] involves using visual-modeling languages. We adopt an expressive and user-friendly graphical notation based on transition systems, which reduces the complexity of modelling services and

[5] `http://www.omg.org/mda/`. Accessed on 30 March 2011.

components, and may be represented by using the metamodel and UML profile for SOA, SoaML[6]. In addition, in order to discover services, in DAMASCo, operation profiles of a signature refer to OWL-S concepts with their arguments and associated semantics. Once services have been discovered, in case there exists mismatch, an *adaptor* to solve problems is automatically generated using software adaptation. An adaptor is a third-party service in charge of coordinating services involved in the system according to a set of interactions defined in an *adaptation contract*. Finally, DAMASCo uses an ad-hoc composition language and error recovery mechanisms to handle service concurrent interactions and design a model to satisfy properties for composite service failure recovery. Therefore, our framework follows the existing need of proposing a model and architecture to get common efforts in the development of CBAs and SOAs [37].

3 DAMASCo Framework

This section presents the DAMASCo architecture and the framework's dynamics.

3.1 DAMASCo Architecture

We focus on avoiding the first type of faults presented in Section 2.1 related to the four interoperability levels. Based on CBSE, SOA and software adaptation, we combine efforts to tackle these levels together. We model services with interfaces constituted by context and semantic information, signatures, and protocol descriptions. We advocate the extension of traditional signature level interfaces with context information (signature and service levels), protocol descriptions with conditions (behavioural level), and semantic representation instead of only a syntactic one (semantic level). We also address the third and fourth type of errors related to dynamic faults by applying fault tolerance. Our whole process consists of a set of processes constituting the DAMASCo architecture, as shown in Figure 2 (detailed in Section 3.2). We focus on systems composed of a service repository, users (clients requesting services)[7], and a shared domain ontology.

3.2 Detailing the DAMASCo Framework

The different elements of DAMASCo architecture have been implemented in `Python` as a set of tools which constitute a framework integrated in the toolbox `ITACA` [12]. `ITACA`[8] (Integrated Toolbox for the Automatic Composition and Adaptation of Web Services) is a toolbox under implementation at the Software Engineering Group of the University of Málaga (GISUM) for the automatic composition and adaptation of services accessed through their interfaces.

We use throughout this section an on-line booking system as running example, consisting of clients and a service repository. Clients can perform different

[6] `http://www.omg.org/spec/SoaML/`. Accessed on 30 March 2011.

[7] We distinguish clients and services, although both refer to components or services.

[8] Accessible at `http://itaca.gisum.uma.es`

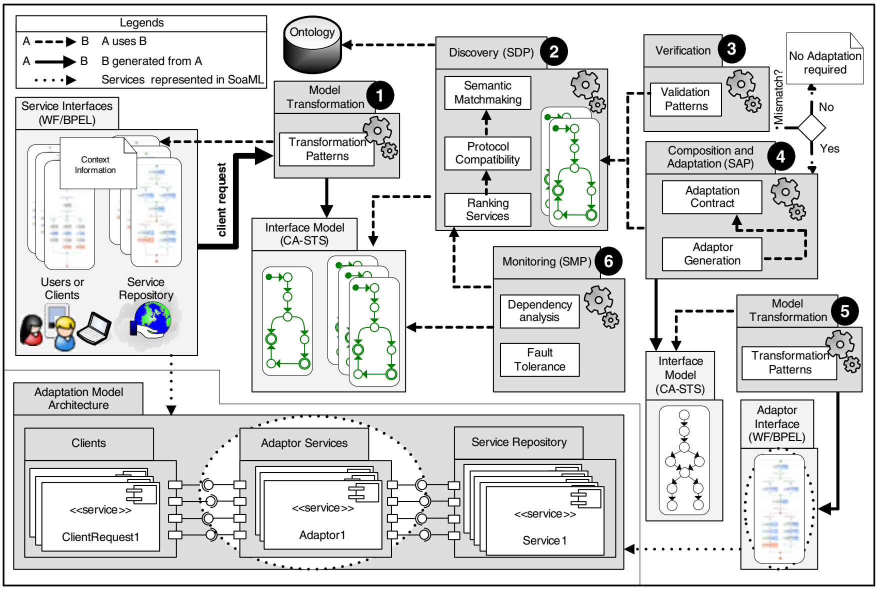


Fig. 2. DAMASCo Architecture

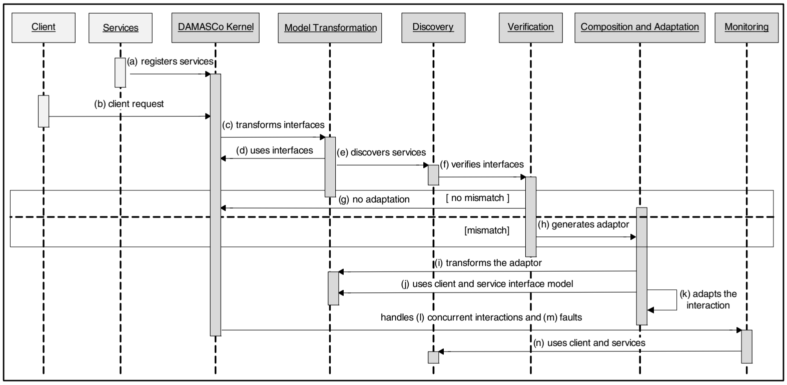


Fig. 3. Dynamics of the DAMASCo framework

requests: book a restaurant, a taxi, a flight, and so on. This case study corresponds to a context-aware pervasive system in which certain context information related to the client (location, privileges, device or language) can change at runtime. Depending on such variations, the system must adapt to work correctly in any situation. Since our approach focuses on solving this kind of mismatch and/or fault situations, it is very appropriate to use our framework in order to work correctly this system. Let us consider a client performs a taxi request.

Figure 3 depicts how after **(a)** services have been registered, the framework's elements interact when **(b)** the client performs the request from either a mobile device (PDA or smartphone) or a laptop, being executed the full process. The purpose of this section is to detail this process at an architectural point of view.

Service Interfaces. Each interface in DAMASCo is made up of a context profile, a signature, and a protocol specified as a transition system. At the user level, client and service interfaces can be specified by using (see Figure 4):

- Context information in XML files for context profiles. We assume context information is inferred by means of the client's requests (HTTP header of SOAP messages), in such a way that as a change occurs the new value of the context attribute is automatically sent to the corresponding service.
- IDL and WSDL descriptions are respectively used in component-based frameworks (*e.g.*, J2EE/.NET) and in service-oriented platforms (*e.g.*, BPEL/WF) for signatures. In WSDL, *e.g.*, services are defined as a collection of ports.
- Business processes defined in industrial platforms, such as BPEL processes or WF workflows, for protocols. We consider clients and services implemented as business processes which provide the WSDL and protocol descriptions.

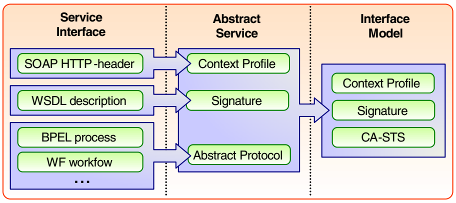


Fig. 4. Interface model obtention from service platforms

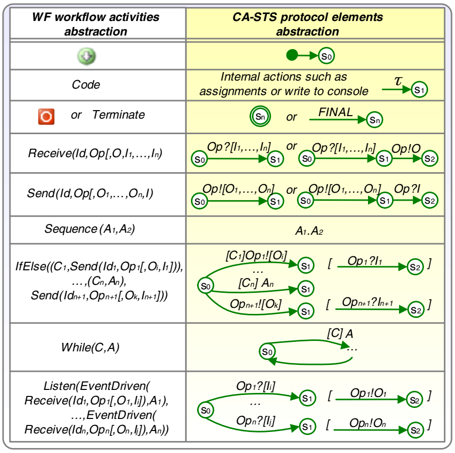

WF workflow activities abstraction
CA-STS protocol elements abstraction
S0
Code
Internal actions such as assignments or write to console
τ
S1
or Terminate
Sn
or
FINAL
Receive(Id,Op[,O,I1,...,In)
Op?[I1,...,In]
Op!O
Send(Id,Op[,O1,...,On,I)
Op![O1,...,On]
Op?I
Sequence (A1,A2)
A1.A2
IfElse((C1,Send(Id1,Op1[,Oi,I1])),...,(Cn,An),Send(Idn+1,Opn+1[,Ok,In+1]))
[C1]Op1![Oi]
[Cn] An
Opn+1![Ok]
[Op1?I1 S2]
[Opn+1?In+1 S2]
While(C,A)
[C] A
Listen(EventDriven(Receive(Id1,Op1[,O1,Ij]),A1),...,EventDriven(Receive(Idn,Opn[,On,Ij]),An))
Op1?[Ij]
Opn?[Ij]
[Op1!O1 S2]
[Opn!On S2]

Fig. 5. Patterns of our model transformation process WF - CA-STS

Model Transformation. First, **(c)** interface specifications are abstracted. Context-Aware Symbolic Transition Systems (CA-STSs) [18] are extracted from **(d)** the BPEL services or WF workflows (*e.g.*, Figure 5 depicts patterns from WF to CA-STS and vice versa, where *!* and *?* represent emission and reception, respectively), which implement the client and the services, by means of our model transformation process [16]. Different automata-based or Petri net-based models could be used to describe behavioural interfaces. We have defined CA-STS, which are based on transition systems (specifically based on STG [27]), because this kind of models are simple, graphical, and provide a good level of abstraction to tackle discovery, verification, composition and adaptation issues, in addition to capture the context information and their changes at run-time [13,23]. Furthermore, any formalism to describe dynamic behaviour may be expressed in terms of a transition system [23]. Thus, our approach becomes general enough to be applied in other fields or applications.

Semantic-Based Service Discovery. Then, **(e)** a service discovery process (SDP) [15] finds out services satisfying the client's request, *i.e.*, with compatible capabilities to the client requirements based on (i) similar context information, semantic matching of signature, and (ii) protocol compatibility, and (iii) a service ranking is performed. Using context information, the topics related to the building of systems which are sensitive to their context is covered. In addition, the advantage of using protocol compatibility is that the services selected not only match at signature, service and semantic levels, but also at behavioural level (solving variable correspondences or incompatible orders). Specifically, our process will identify mismatch situations by using ontologies[9] and synchronous product [2] to determine if adaptation is required or not. It generates correspondence sets among service interfaces involved in the interaction, used in the composition and adaptation process as an adaptation contract.

Verification of Interfaces. Next, before performing composition, **(f)** model checking techniques are used for validating a set of properties, such as determinism, state liveness, inter-communication liveness (request/response), and non-blocking states [19], for the CA-STS client and services selected in the discovery process. In particular, we use Ordered Binary Decision Diagrams (OBDD) [11] because of the use of context information and conditions over transitions, since a standard model checker validating on states only considers (sequences of) states through temporal formulae, but not sets of states satisfying boolean formulae.

Composition and Adaptation. If adaptation is required, then **(h)** a service composition and adaptation process (SAP) [16] is executed (otherwise **(g)** no adaptation). Thus, an adaptation contract solving mismatch problems is automatically obtained. Being given the CA-STSs corresponding to client and services, as well as the adaptation contract, a monolithic CA-STS adaptor specification can be generated [33], whose resulting composition preserves the properties previously validated in the verification process. Next, **(i)** the corresponding BPEL or WF adaptor service is obtained from the CA-STS adaptor specification using our model transformation process. This process prunes parts of the CA-STS specification corresponding to additional behaviours (interleavings) that cannot be implemented into executable languages (BPEL or WF) and keeps only executable paths. Finally, the whole system is deployed, allowing **(j)** the BPEL/WF client and services to interact via **(k)** the BPEL/WF adaptor.

This process is illustrated in Figure 6, which shows how a part of the composition corresponding to Client-Taxi request and Taxi service interfaces is synchronised through the adaptor by connecting parameters by means of placeholders.

[9] Ontology generated for our example using Protégé 4.0.2 can be found in `http://www.lcc.uma.es/~cubo/contextive/owls/ontologies/ebooking.owl.xml`

Specifically, the adaptation process synchronises the adaptor with the service interfaces through the same name of messages but using reversed directions. It can also be observed how the synchronisation solves behaviour mismatch such as 1-N correspondence (*user!* and *password!* with *login?*), variable correspondence (*[priv==“VIP”]priceTaxiVIP!* or *[priv==“Guest”]priceTaxiGuest!* with *priceTaxi?*) and incompatible order (sequence *reqTaxi!*, *user!* and *login!* with respect to *login?* and *getTaxi?*). Furthermore, our process can control dynamic context changes. We assume the dynamic context attribute $\tilde{priv}$ corresponding to the client privileges is *“Guest”* when the taxi request is issued. Then, that attribute changes to *“VIP”* at run-time before the request is received. Our process captures and handles that dynamic context change, and simulates the dynamic update of the environment according to the context changes at run-time [18]. The process carries on the execution by the branch corresponding to *“VIP”*.

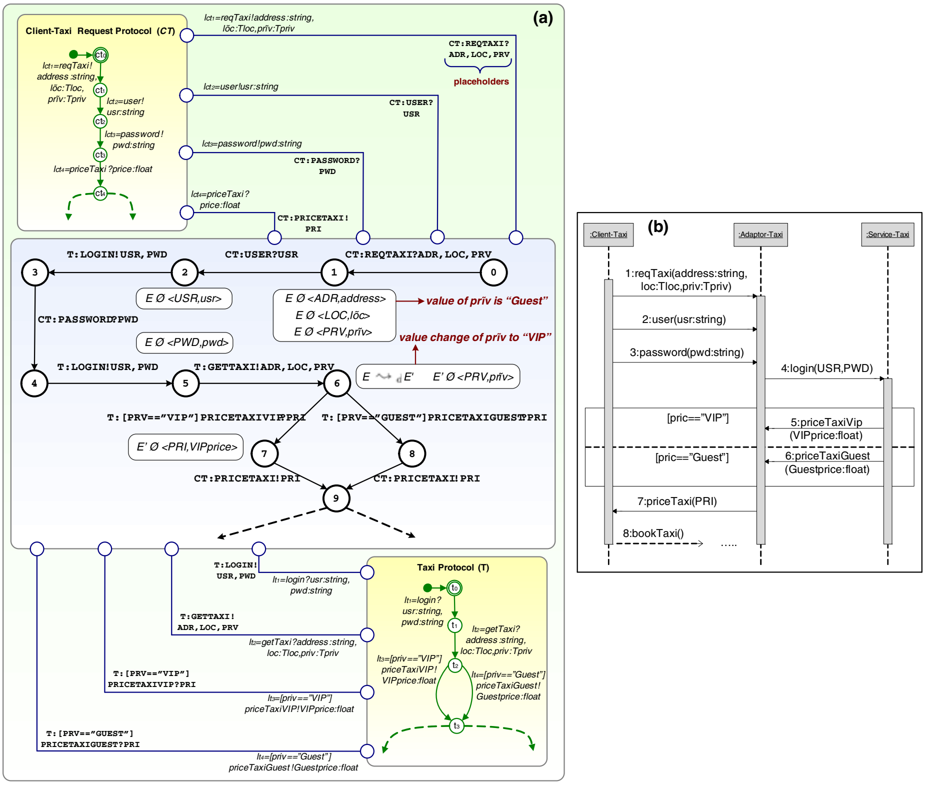

(a)
Client-Taxi Request Protocol (CT)
lct1=reqTaxi! address:string, lõc:Tloc, prĩv:Tpriv
lct2=user! usr:string
lct3=password! pwd:string
lct4=priceTaxi?price:float
ct0
ct1
ct2
ct3
ct4
lct1=reqTaxi!address:string, lõc:Tloc,prĩv:Tpriv
CT:REQTAXI? ADR,LOC,PRV
placeholders
lct2=user!usr:string
CT:USER? USR
lct3=password!pwd:string
CT:PASSWORD? PWD
lct4=priceTaxi? price:float
CT:PRICETAXI! PRI
T:LOGIN!USR, PWD
CT:USER?USR
CT:REQTAXI?ADR, LOC, PRV
3
2
1
0
E Ø <USR,usr>
E Ø <ADR,address>
E Ø <LOC,lõc>
E Ø <PRV,prĩv>
value of prĩv is “Guest”
CT:PASSWORD?PWD
E Ø <PWD,pwd>
value change of prĩv to “VIP”
T:LOGIN!USR, PWD
T:GETTAXI!ADR, LOC, PRV
4
5
6
E ⤳ E'
E' Ø <PRV,prĩv>
T:[PRV==”VIP”]PRICETAXIVIP?PRI
T:[PRV==”GUEST”]PRICETAXIGUEST?PRI
E' Ø <PRI,VIPprice>
7
8
CT:PRICETAXI!PRI
CT:PRICETAXI!PRI
9
T:LOGIN! USR, PWD
lt1=login?usr:string, pwd:string
Taxi Protocol (T)
t0
t1
t2
t3
lt1=login? usr:string, pwd:string
lt2=getTaxi? address:string, loc:Tloc,priv:Tpriv
T:GETTAXI! ADR, LOC, PRV
lt2=getTaxi?address:string, loc:Tloc,priv:Tpriv
lt3=[priv==”VIP”] priceTaxiVIP! VIPprice:float
lt4=[priv==”Guest”] priceTaxiGuest! Guestprice:float
T:[PRV==”VIP”] PRICETAXIVIP?PRI
lt3=[priv==”VIP”] priceTaxiVIP!VIPprice:float
T:[PRV==”GUEST”] PRICETAXIGUEST?PRI
lt4=[priv==”Guest”] priceTaxiGuest!Guestprice:float
(b)
:Client-Taxi
:Adaptor-Taxi
:Service-Taxi
1:reqTaxi(address:string, loc:Tloc,priv:Tpriv)
2:user(usr:string)
3:password(pwd:string)
4:login(USR,PWD)
[pric==”VIP”]
5:priceTaxiVip (VIPprice:float)
[pric==”Guest”]
6:priceTaxiGuest (Guestprice:float)
7:priceTaxi(PRI)
8:bookTaxi()
.....

Fig. 6. (a) Synchronised composition and (b) sequence diagram between Client-Taxi request and the Taxi service interface by means of the corresponding Adaptor

Run-Time Monitoring. Then, a service monitoring process (SMP) [18,17] focuses on **(l)** handling the concurrent execution of **(n)** the client with the composition of several services on the same user device, using an own composition language and mechanisms based on data semantic matching. Our approach aims at assisting the user in establishing priorities between dependencies detected, avoiding the occurrence of deadlock situations. In addition, **(m)** in case a problem occurs during composition, such as a connection loss, our fault-tolerance monitoring process will search on-the-fly for new service interfaces that respond to a specific client's request (see [17] for further details),

4 Evaluation and Discussion

In order to evaluate the benefits of our framework, DAMASCo has been validated on several examples. These examples include an on-line computer material store, a travel agency, an on-line booking system (our running example), and a road information system. These scenarios have been implemented in the WF platform by us and executed on an Intel Pentium(R)D CPU 3GHz, 3GB RAM computer, with Microsoft Windows XP Professional SP2. This represents an initial stage, checking our whole framework, but the main goal of our approach is to support industrial systems by directly validating pre-existing applications in the real-world. We have evaluated the experimental results in two separate parts: discovery and adaptation processes, and monitoring process.

Service Discovery (SDP) and Adaptation (SAP) Processes. We have validated the full discovery process and the adaptation contract generation in two case studies: an on-line booking system, and a road information system. For each case study, we have executed a client's request, and we have also studied three different versions for each, which are organised according to increasing size and complexity with respect to the number of interfaces involved, as well as the overall size of protocols as a total number of states and transitions.

Table 1 shows the experimental results (CPU load and execution time) corresponding to different versions of both case studies.

Table 1. Experimental results of the discovery and adaptation processes

Scenario	Size			Parameter	
(Version)	Interfaces	States	Transitions	CPU(%)	Time(s)
obs-v04	4	22	26	11,1	0,110
obs-v005	25	128	160	16,2	0,688
obs-v07	67	352	440	34,1	1,719
ris-v05	5	27	33	13,8	0,249
ris-v06	44	264	302	20,7	0,954
ris-v07	103	650	780	47,6	2,437

Figure 7 shows the scalability of our discovery and adaptation processes for different versions of both case studies with respect to the number of transitions. We could also study the scalability *w.r.t.* the number of interfaces or states, with

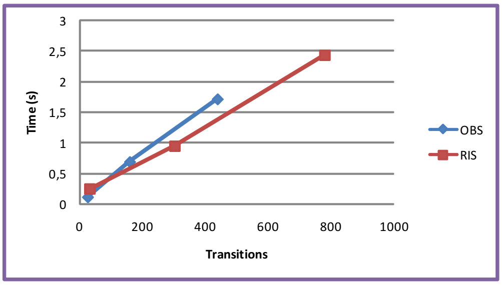


Fig. 7. Scalability of the discovery and adaptation processes for both case studies

similar results. One can observe when we increase the number of transitions the growth is linear, so complexity of service composition does not affect severely.

We have also evaluated the accuracy of our discovery process with precision, recall and F_1 measures [29] for different requests on the scenarios presented in Table 1. Precision is a measure of exactness, whereas recall is a measure of completeness, and F_1 is the harmonic mean of precision and recall that combines both, and the values range for these three measures is between 0 and 1. For all requests that we performed, the results of precision and recall were equal 1, which shows our discovery process has 100% precision and recall, and the score F_1 is always 1. These results prove the importance of the context-awareness, semantic matching and protocol compatibility mechanisms used in our process in order to discover services and generate a whole adaptation contract. In addition, we have validated the CA-STS interfaces corresponding to both case studies against a set of properties, by using our OBDD representation. The verification of those properties required less than 2 seconds for all the CA-STSs of each case study.

Service Monitoring Process (SMP). We have evaluated the composition language that allows the execution and management of concurrent interactions at the same time, by managing data dependencies to avoid inconsistent or deadlock situations. In order to evaluate this process, we have conducted a small experimental study with the assistance of twelve volunteers. This study helped us to determine how our approach behaves in terms of evaluating the benefits to find out data dependencies in concurrent executions and to handle those dependencies in terms of effort required, efficiency and accuracy of the dependencies detected. Users performed tests either in a manual or in an interactive (using the tool) way. In order to perform the tests, we provided them with a graphical representation of the interfaces and a specific domain ontology to be used in the concurrent interaction, for a specific scenario of each problem. Each user solved different problems using different specifications (manual or interactive) to prevent previous user knowledge of a particular case study.

Table 2 shows the scenarios of each case study used for our study. The scenarios are organised according to increasing size and complexity with respect to the number of interfaces (Client and Serv.) involved and the ontology, as well as the overall size of client protocols (Client Prot.) as a total number of states

Table 2. Experimental results for the Manual (M) and Interactive (I) specifications

Example	Size				Parameter					
	Interfaces		Client Prot.		Time(s)		Depend.		Errors	
	Client	Serv.	Sta.	Tran.	M	I	M	I	M	I
pc-v02	2	2	10	8	61,80	19,14	1	3	2	0
obs-v04	2	3	16	15	51,60	3,17	1	1	0	0
ris-v05	3	3	16	18	113,62	16,51	5	4	3	0
tra-v02	3	5	36	36	271,84	62,38	12	12	4	0

(Sta.) and transitions (Tran.). Tests considered all the client protocols interacting concurrently. The table also includes the comparison of experimental results using both manual and interactive specification of data dependencies and their corresponding execution priorities. We consider as parameters the time required to solve the problem (in seconds), the number of label dependencies detected (Depend.), and the number of errors in the specified data dependency set.

There is a remarkable difference in the amount of time required to solve the different scenarios between manual and interactive specification. We measure as errors the number of wrong, unnecessary or non-detected label dependencies. Our tool always detects all the data dependencies and it uses semantic matching to determine those dependencies, so this is a clear advantage, which increases with the complexity of the problem, compared to the manual specification. Thus, the time elapsed for detecting dependencies by using our tool experiences a linear growth with the size of the scenario. Therefore, scalability, efficiency and accuracy of our tool are satisfactory, and for instance, in the worst case (tra-v02) the time required is roughly 1 minute, which is a reasonable amount of time.

In Figures 8 can be observed graphically the experimental results of efficiency and accuracy of the handling data-based composition for several examples. In order to measure the efficiency and accuracy, we consider the required time and number of errors (for both manual and interactive specifications), respectively.

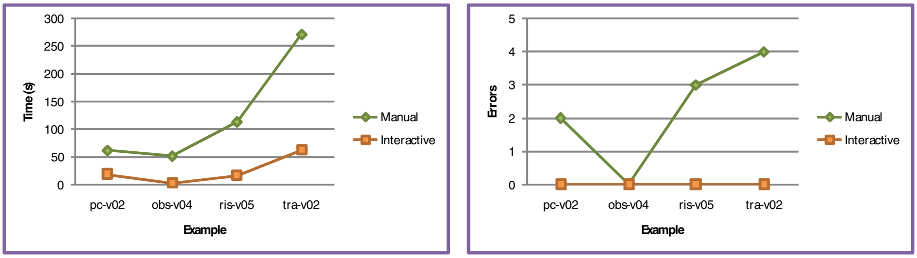


Fig. 8. Efficiency and accuracy of the monitoring process for several examples

5 Related Work

This section compares DAMASCo to related works to show the need of our framework, which ovecomes certain gaps in discovery, adaptation and monitoring.

Discovery. Service discovery is one of the major challenges in services computing, thus many attempts are currently being studied. Several works address service discovery focused on context-awareness and/or semantic-based composition [8,9,29]. Other approaches tackle protocol compatibility [25,26,30,44]. In [40], service discovery bases on structural and behavioural service models, as well as contextual information, but nothing about semantic description. Therefore, there do not exist methods to automatically and directly detect all the mismatch situations related to the four interoperability levels as is addressed in DAMASCo.

Adaptation Based on Model Transformation. Deriving adaptors is a complicated task. Many approaches tackle model-based adaptation at signature and behavioural levels, although only some of them are related with existing programming languages and platforms, using CORBA [5], COM/DCOM [28], BPEL [10], and SCA components [36]. In [32], the authors present an approach that supports behaviour analysis of component composition according to a software architecture. In comparison to the aforementioned works, DAMASCo focuses on modelling the behavioural composition, not only preventing mismatch problems, but also taking into account context changes and semantic matching in order to control states of inconsistence and to relate automatically message names even when their signatures do not match. In addition, our model transformation becomes general enough to be applied in different languages or platforms in addition to BPEL or WF.

Monitoring. Recent approaches have been dedicated to the run-time service interaction to compose and adapt their execution. There exists monitoring approaches [4,39], which define rules, specification languages, temporal logic or event calculus for expressing and verifying functional and non-functional properties. However, to the best of our knowledge, only recent approaches [6,35,45] have tackled monitoring by handling run-time concurrent interactions of service protocols through data dependency analysis. Compared to these works, DAMASCo does not only detect data dependencies, addressing both direction and order (priorities), but also allows context-aware protocol concurrent executions at run-time by using a composition language and semantic matching techniques.

6 Concluding Remarks

We have illustrated the need to support the modelling, discovery and variability of the adaptation process according to the dynamic aspects of component-based and service-oriented architectures in context-aware mobile and pervasive systems. Recent research approaches have tackled independently the discovery, composition, adaptation or monitoring processes. DAMASCo framework has been presented as a novel solution which combines efforts to address all those issues through model transformations of real-world applications. This paper presents the DAMASCo architecture. We have validated our framework on several examples, and we have shown it is scalable, efficient and accurate.

We are currently extending our framework to open systems by tackling dynamic reconfiguration of services (second type of faults presented in Section 2.1), by handling the addition and elimination of both services and requirements. As regards future work, we plan to address other non-functional requirements at the service level, such as temporal requirements or security, in addition to context information, as well as more semantic capabilities of the service interfaces considering the full power of the Web Semantic technologies, which includes automated Web Service discovery, execution, composition and interoperation.

References

1. Andrews, T., et al.: Business Process Execution Language for Web Services (WS-BPEL). BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems (2005)
2. Arnold, A.: Finite Transition Systems. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1994)
3. Arsanjani, A., Zhang, L.-J., Ellis, M., Allam, A., Channabasavaiah, K.: S3: A Service-Oriented Reference Architecture. IEEE IT Professional 9, 10–17 (2007)
4. Baresi, L., Guinea, S., Pistore, M., Trainotti, M.: Dynamo + Astro: An Integrated Approach for BPEL Monitoring. In: Proc. of ICWS 2009, pp. 230–237. IEEE Computer Society, Los Alamitos (2009)
5. Bastide, R., Sy, O., Navarre, D., Palanque, P.A.: A Formal Specification of the CORBA Event Service. In: Proc. of FMOODS 2000, pp. 371–396. Kluwer Academic Publishers, Dordrecht (2000)
6. Basu, S., Casati, F., Daniel, F.: Web Service Dependency Discovery Tool for SOA Management. In: Proc. of SCC 2007, pp. 684–685. IEEE Computer Society, Los Alamitos (2007)
7. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. In: Reussner, R., Stafford, J.A., Ren, X.-M. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 193–215. Springer, Heidelberg (2006)
8. Benatallah, B., Hacid, M.S., Rey, C., Toumani, F.: Request Rewriting-Based Web Service Discovery. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 242–257. Springer, Heidelberg (2003)
9. Brogi, A., Corfini, S., Popescu, R.: Semantics-Based Composition-Oriented Discovery of Web Services. ACM Transactions on Internet Technology 8(4), 19:1–19:39 (2008)
10. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
11. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
12. Cámara, J., Martín, J.A., Salaün, G., Cubo, J., Ouederni, M., Canal, C., Pimentel, E.: ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In: Proc. of ICSE 2009, pp. 627–630. IEEE Computer Society, Los Alamitos (2009)

13. Canal, C., Poizat, P., Salaün, G.: Model-Based Adaptation of Behavioural Mismatching Components. IEEE Transactions on Software Engineering 34(4), 546–563 (2008)
14. Chappel, D.A.: Enterprise Service Bus. O'Reilly, Sebastopol (2004)
15. Cubo, J., Canal, C., Pimentel, E.: Context-Aware Service Discovery and Adaptation Based on Semantic Matchmaking. In: Proc. of ICIW 2010, pp. 554–561. IEEE Computer Society, Los Alamitos (2010)
16. Cubo, J., Canal, C., Pimentel, E.: Context-Aware Composition and Adaptation Based on Model Transformation. Journal of Universal Computer Science 17(15), 777–806 (2011)
17. Cubo, J., Canal, C., Pimentel, E.: Model-Based Dependable Composition of Self-Adaptive Systems. Informatica 35, 51–62 (2011)
18. Cubo, J., Pimentel, E., Salaün, G., Canal, C.: Handling Data-Based Concurrency in Context-Aware Service Protocols. In: Proc. of FOCLASA 2010. Electronic Proceeding in Theoretical Computer Science, vol. 30, pp. 62–77 (2010)
19. Cubo, J., Sama, M., Raimondi, F., Rosenblum, D.: A Model to Design and Verify Context-Aware Adaptive Service Composition. In: Proc. of SCC 2009, pp. 184–191. IEEE Computer Society, Los Alamitos (2009)
20. Dey, A.K., Abowd, G.D.: Towards a Better Understanding of Context and Context-Awareness. In: Proc. of Workshop on the What, Who, Where, When and How of Context-Awareness, pp. 304–307 (2000)
21. Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice-Hall, Englewood Cliffs (2005)
22. Fiadeiro, J.L., Lopes, A.: A Model for Dynamic Reconfiguration in Service-Oriented Architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 70–85. Springer, Heidelberg (2010)
23. Foster, H., Uchitel, S., Kramer, J.: LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In: Proc. of ICSE 2006, pp. 771–774. ACM Press, New York (2006)
24. Gorbenko, A., Romanovsky, A., Kharchenko, V.S., Mikhaylichenko, A.: Experimenting with Exception Propagation Mechanisms in Service-Oriented Architecture. In: Proc. of WEH 2008, pp. 1–7. ACM Press, New York (2008)
25. Hameurlain, N.: Flexible Behavioural Compatibility and Substitutability for Component Protocols: A Formal Specification. In: Proc. of SEFM 2007, pp. 391–400. IEEE Computer Society, Los Alamitos (2007)
26. Han, W., Shi, X., Chen, R.: Process-Context Aware Matchmaking for Web Service Composition. Journal of Network and Computer App. 31(4), 559–576 (2008)
27. Hennessy, M., Lin, H.: Symbolic Bisimulations. Theor. Comput. Sci. 138(2), 353–389 (1995)
28. Inverardi, P., Tivoli, M.: Deadlock-free Software Architectures for COM / DCOM Applications. The Journal of Systems and Software 65(3), 173–183 (2003)
29. Klusch, M., Fries, B., Sycara, K.: Automated Semantic Web Service Discovery with OWLS-MX. In: Proc. of AAMAS 2006, pp. 915–922. ACM Press, New York (2006)
30. La, H.J., Kim, S.D.: Adapter Patterns for Resolving Mismatches in Service Discovery. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 498–508. Springer, Heidelberg (2010)
31. López-Sanz, M., Qayyum, Z., Cuesta, C.E., Marcos, E., Oquendo, F.: Representing Service-Oriented Architectural Models Using π-ADL. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 273–280. Springer, Heidelberg (2008)

32. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures. In: Proc. of WICSA 1999, pp. 35–49. Kluwer Academic Publishers, Dordrecht (1999)
33. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
34. Mens, T., Demeyer, S.: Software Evolution. Springer, Heidelberg (2008)
35. Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., Dustdar, S.: A Context-Based Mediation Approach to Compose Semantic Web Services. ACM Transactions on Internet Technology 8(1), 4:1–4:23 (2007)
36. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-Automated Adaptation of Service Interactions. In: Proc. of WWW 2007, ACM Press, New York (2007)
37. de Oliveira, L.B.R., Romero Felizardo, K., Feitosa, D., Nakagawa, E.Y.: Reference Models and Reference Architectures Based on Service-Oriented Architecture: A Systematic Review. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 360–367. Springer, Heidelberg (2010)
38. Scribner, K.: Microsoft Windows Workflow Foundation: Step by Step. Microsoft Press (2007)
39. Sheng, Q.Z., Benatallah, B., Maamar, Z., Dumas, M., Ngu, A.H.H.: Configurable Composition and Adaptive Provisioning of Web Services. IEEE Transactions on Services Computing 2(1), 34–49 (2009)
40. Spanoudakis, G., Mahbub, K., Zisman, A.: A Platform for Context Aware Runtime Web Service Discovery. In: Proc. of ICWS 2007, pp. 233–240. IEEE Computer Society, Los Alamitos (2007)
41. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison Wesley, Reading (2003)
42. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: Dependability in the Web Services Architecture. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) ADS 2003. LNCS, vol. 2677, pp. 90–109. Springer, Heidelberg (2003)
43. Tibermacine, C., Kerdoudi, M.L.: From Web Components to Web Services: Opening Development for Third Parties. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 480–484. Springer, Heidelberg (2010)
44. Wang, L., Krishnan, P.: A Framework for Checking Behavioral Compatibility for Component Selection. In: Proc. of ASWEC 2006, pp. 49–60. IEEE Computer Society, Los Alamitos (2006)
45. Yan, S., Wang, J., Liu, C., Liu, L.: An Approach to Discover Dependencies between Service Operations. Journal of Software 3(9), 36–43 (2008)
46. Zapletal, M., van der Aalst, W.M.P., Russell, N., Liegl, P., Werthner, H.: An Analysis of Windows Workflow's Control-Flow Expressiveness. In: Proc. of ECOWS 2009, pp. 200–209. IEEE Computer Society, Los Alamitos (2009)

A Service-Oriented Reference Architecture for Software Testing Tools

Lucas Bueno Ruas Oliveira and Elisa Yumi Nakagawa

Department of Computer Systems
University of São Paulo - USP
PO Box 668, 13560-970, São Carlos, SP, Brazil
{buenolro,elisa}@icmc.usp.br

Abstract. Software testing is recognized as a fundamental activity for assuring software quality. Aiming at supporting this activity, a diversity of testing tools has been developed, including tools based on SOA (Service-Oriented Architecture). In another perspective, reference architectures have played a significant role in aggregating knowledge of a given domain, contributing to the success in the development of systems for that domain. However, there exists no reference architecture for the testing domain that contribute to the development of testing tools based on SOA. Thus, the main contribution of this paper is to present a service-oriented reference architecture, named RefTEST-SOA (Reference Architecture for Software Testing Tools based on SOA), that comprises knowledge and experience about how to structure testing tools organized as services and pursues a better integration, scalability, and reuse provided by SOA to such tools. Results of our case studies have showed that RefTEST-SOA is a viable and reusable element to the development of service-oriented testing tools.

1 Introduction

Software testing is one of the most important activities to guarantee quality and reliability of software under development [1,2]. In order to systematize the testing activity, a diversity of testing tools has been developed, aiming at minimizing cost, time consumed, as well as errors caused by human intervention. Testing automation is therefore an important issue related to the quality and productivity of the testing processes and, as a consequence, of the software processes [1]. However, these tools have almost always been implemented individually and independently, presenting its own architectures and data structures. As a consequence, difficulty of integration, evolution, maintenance, and reuse of these tools is very common.

In another perspective, software architectures have played a major role in determining system quality, since they form the backbone to any successful software-intensive system [3]. Decisions made at the architectural level directly enable, facilitate, hamper or interfere with achieving business goals as well as meeting functional and quality requirements. Thus, software architecture is a

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 405–421, 2011.

structure (or a set of structures) of the system which comprises software elements, the externally visible properties of those elements, and the relationships among them [3]. In this context, reference architectures have emerged as an element that aggregates knowledge of a specific domain by means of modules and their relations. They promote reuse of design expertise by achieving solid, well-recognized understanding of a specific domain. Considering their relevance, reference architectures for different domains have been proposed and in fact contributed to the development of software systems [4]. Besides that, reference architecture for software testing domain can be also found [5,6].

In the context of software architecture, an architectural style has taken attention in the last years: the SOA (Service-Oriented Architecture) [7]. This style makes possible the improvement of integration, scalability, and reuse, since systems based on this style present independence regarding programming language and execution platform [7]. Considering the advantages of SOA, use of this architectural style to develop testing tools can be also found [8,9]. Besides advantages provided by SOA, specifically to the testing domain, SOA could make possible a complementary use of testing techniques and criteria, that is highly recommended by the testing literature [2]. Thus, different testing tools available as services could be used in an integrated way, composing a major testing environment. Therefore, considering the relevance of service-oriented testing tools, a reference architecture could support development of such tools.

The main objective of this paper is to present a service-oriented reference architecture, named RefTEST-SOA (Reference Architecture for Software Testing Tools based on SOA), for the software testing domain. Our research group has been working with software testing for many years and several tools for the testing activity have been developed in this period. This architecture encompasses therefore knowledge and experience of the testing domain and intends to facilitate development of testing tools organized as services, i.e., service-oriented testing tools. In order to have evidence about the viability of RefTEST-SOA, we present a case study using it to the development of a service-oriented testing tool. As main result, we observed possibility of testing tools actually composed by different services, promoting mainly reusability in the development of such tools.

The remainder of this paper is organized as follows. In Section 2, background about reference architecture, SOA, and related work are presented. In Section 3, RefTEST-SOA is presented. In Section 4, we present a case study involving development of a service-oriented testing tool from RefTEST-SOA. In Section 5, we discuss results and limitations of our work, as well as conclusions and future directions.

2 Background and Related Work

Reference architecture is a special type of architecture that provides major guidelines for the specification of concrete architectures of a class of systems [4]. In order to systematize the design of reference architectures, guidelines and processes have been also established [10,11]. Moreover, the effective reuse of knowledge of reference architectures depends not only on raising the domain knowledge, but also documenting and communicating efficiently this knowledge through an adequate architectural description. Commonly, architectural views have been used, together with UML (Unified Modeling Language) techniques, to describe reference architectures. Considering the relevance of reference architectures as basis of software development, a diversity of architectures has been proposed and used, including for Software Testing [5,6]. However, reference architectures for Software Testing domain proposed until now do not support development of service-oriented testing tools.

Regarding SOA, it has introduced the concept of business service (or simply service) as a fundamental unit to design, build, and composite service-oriented software systems [7]. A service provides usually business functionalities; furthermore, it is independent of the context and of the state of other services. For services to properly work, SOA requires establishment of mechanisms for communication among services, either using a direct communication or a broker (i.e., a mediator among the services). Besides that, to build service-oriented systems, it is important to have a highly distributed communication and integration backbone. This functionality can be provided by Enterprise Service Bus (ESB) [12] that refers to an integration platform to support a wide variety of communication patterns over multiple transport protocols and deliver value-added capabilities for SOA applications [7]. Through composition of simple services, more complex service-oriented systems can be built and, according to Papazoglou and Heuvel [7], in the more productive and agile way. In other words, SOA intends cooperation of low coupling services in order to create dynamic and flexible business processes. Service composition is therefore considered one of the most promising characteristic of SOA. In this context, concepts, such as service orchestration and service choreography [13], are important. To ensure quality and inter-operability among services, contracts can be used as a formal agreement to specify relationship between a service and its clients, expressing each part's rights and obligations [14].

With respect to related work, we can find general reference architectures and reference models. Well-known examples are S3 (Service-Oriented Solution Stack) reference architecture [15], OASIS reference model [16] and OASIS reference architecture [17]. They are domain-independent and intend to guide design of concrete architectures of systems based on SOA. In particular, S3 provides an architectural description of SOA through a structure organized in layers and presents concepts related to services mapped in technological aspects [15]. Otherwise, the reference model and reference architecture proposed by OASIS aim at defining a common vocabulary and understanding about elements and interactions in SOA, independently from implementation technologies [17]. Besides these domain-independent architectures and models, a diversity of service-oriented reference architectures for specific domains can be found [18], such as for e-learning [19] and e-working [20]. This diversity shows the relevance of reference architectures based on SOA, as well as the interest in the development of service-oriented systems for different domains.

In the testing domain, we have also observed interest in developing testing tools organized as services [8,9,21]. In particular, JaBUTiService [9] is a testing service that automates structural testing of programs in Java and AspectJ[1]. Eler et al. [21] propose an approach to improve web service testability by developing web services with built-in structural testing capabilities. Moreover, Bartolini et al. [8] propose a mechanism in order to conduct structural testing for composition of services. Since initiatives of service-oriented testing tools can be already found, establishment of a reference architecture that supports and facilitates development of such tools can be considered of great importance.

3 Establishing RefTEST-SOA

RefTEST-SOA is a reference architecture that intends to support development of testing tools organized as a set of services, i.e., service-oriented testing tools. In order to establish this architecture, we have used ProSA-RA [11], a process to build reference architectures, illustrated in Figure 1. In short, to establish reference architectures by using ProSA-RA, information sources are firstly selected and investigated (in Step RA-1) and architectural requirements are identified (in Step RA-2). After that, an architectural description of the reference architecture is established (in Step RA-3) and evaluation of this architecture is conducted (in Step RA-4). Following, each step in presented in more details in order to establish our reference architecture:

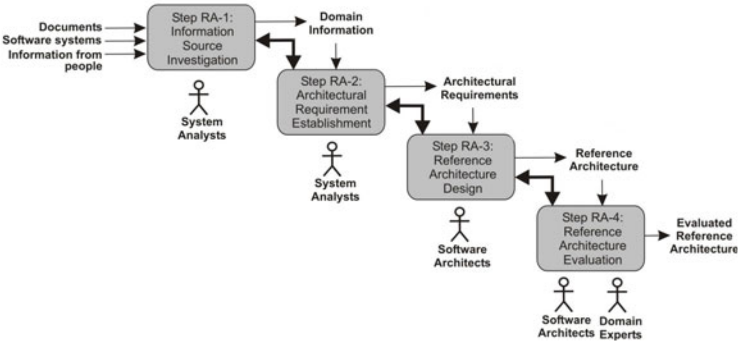

Documents
Software systems
Information from people
Step RA-1: Information Source Investigation
Domain Information
System Analysts
Step RA-2: Architectural Requirement Establishment
Architectural Requirements
System Analysts
Step RA-3: Reference Architecture Design
Reference Architecture
Software Architects
Step RA-4: Reference Architecture Evaluation
Evaluated Reference Architecture
Software Architects
Domain Experts

Fig. 1. Outline Structure of ProSA-RA (Adapted from [11])

3.1 Step RA-1: Information Source Investigation

In this step, we identified information sources, aiming to elicit requirements to our reference architecture. Different sources were considered, mainly related to SOA and software testing domain. Thus, the four sets of sources were: (i)

[1] http://www.eclipse.org/aspectj/

service-oriented tools of the testing domain; (ii) guidelines to development of service-oriented systems; (iii) service-oriented reference architectures related to other domains; and (iv) other reference architectures for the testing domain. It is worth highlighting that to identify information sources for the sets in (ii) and (iii), we conducted a Systematic Review[2], presented in more details in [18]. The use of systematic reviews to support gathering of information sources and requirements elicitation is already investigated elsewhere [23,24]. Following, each set is described in more details:

- **Set 1: Service-oriented testing tools:** An important information source is the set of testing tools organized as services; thus, we identified in the literature three testing tools [9,21,8]. Furthermore, we considered also one verification tool [25] and one software analysis tool [26], that present initiatives to be available as services. These tools were therefore studied and their internal structures were analyzed. Besides that, issues related to integration and communication mechanisms and service discovery were also analyzed. Therefore, it is important to observe that, in spite of increasing interest in developing systems based on SOA, there are few initiatives of software testing tools organized as services.
- **Set 2: Guidelines to the development of service-oriented systems:** Since domain-independent reference models and reference architectures for SOA have been proposed and used as successful basis to establish reference architectures for different domains, as pointed out in [18], we have also investigated them. In particular, we considered OASIS [17] and S3 [15]. They present the structure, functionalities, and characteristics that could be present in service-oriented systems. Based on them, we summarized the main concepts related to SOA that could be considered during establishment of our reference architecture. The concepts are: Service Description (SD), Service Publication (SP), Service Interaction (SI), Service Composition (SC), Polices (P), Governance (G), and Quality of Service (QoS). Table 1 summarizes these concepts and sources that contributed to identify them.

Table 1. Concepts Related to SOA

ID	Source	Concept
ST1	Arsanjani et al. (2007) [15]	SD, SP, SI, SC, P, G, QoS
ST2	Dillon et al. (2007) [27]	SP, SI, SC, QS
ST3	Lan et al. (2008) [28]	SP, SI, SC, P, G
ST4	OASIS (2006) [16]	SD, SP, SI, SC, P
ST5	OASIS (2008) [17]	SD, SP, SI, SC, P, G, QoS
ST6	Zimmermann (2009) [29]	P, G, QoS

[2] Systematic Review is a technique proposed by Evidence-Based Software Engineering and enables to have a complete and fair evaluation about a topic of interest [22].

- **Set 3: Service-oriented reference architectures:** We investigated also service-oriented reference architectures of different domains. As stated before, we conducted a Systematic Review in order to possibly identify all architectures [18]. A total of 11 architectures were studied and we observed that the concepts Service Description (SD) and Service Interaction (SI) are present in all architectures, even because these concepts are inherent characteristics of SOA. Furthermore, Service Publication (SP) is present in the most of architectures. Quality of Service (QoS) and Service Composition (SC) are present in a half of the reference architectures. Otherwise, Polices (P) and Governance (G) are considered in one-third of these architectures. Thus, in spite of relevance of all these concepts in service-oriented systems, considering an analysis of reference architectures, not all architectures have considered all concepts. Furthermore, these architectures were important in our work, since they provided knowledge and experience about how to structure and represent service-oriented reference architectures.
- **Set 4: Reference architectures for the testing domain:** Since reference architectures aggregate knowledge of a specific domain, we have investigated reference architecture previously proposed to the testing domain. As far as we know, two architectures were proposed to this domain [5,6]. Eickelmann and Richardson [5] presented an architecture divided in six functionalities — planning, management, measurement, fault analysis, development, and execution — inspired on software processes. Nakagawa et al. [6] proposed an architecture based on SoC (Separation of Concerns) and international standard ISO/IEC 12207 [30], aiming at being basis to tools that support primary, organizational, and supporting activities of the testing domain. Since this architecture presents a more complete documentation and it has been successfully used to develop testing tools, such as presented in [31], we have considered the knowledge about testing domain contained in this architecture to contribute to our architecture. In spite of that, it is worth highlighting that this architecture is not indicated to be basis to develop service-oriented testing tools.

3.2 Step RA-2: Architectural Requirement Establishment

Based on information and concepts identified in last step, we established the architectural requirements of our architecture. We identified a total of 39 requirements and we classified them in two sets: (i) architectural requirements of the testing domain that were obtained from Sets 1 and 4; and (ii) architectural requirements related to concepts of the SOA that were obtained from Sets 1, 2 and 3. Table 2 illustrates part of these requirements. First column refers to the requirement identification (R-T, i.e., Requirement related to Testing domain and R-S, i.e., Requirement related to SOA); second column refers to requirement description; and third column refers to concepts related to requirements. For instance, requirement R-T2 (The reference architecture must allow development of testing tools that support test criteria management.) is related to Testing Criterion concept. This same analysis was conducted to each requirement. As result,

Table 2. Part of the RefTEST-SOA requirements

ID	Requirement	CPT
R-T1	The reference architecture must enable development of testing tools that provide mechanisms to add test requirements, generating or importing them.	TR
R-T2	The reference architecture must allow development of testing tools that support test criteria management.	TCr
R-T3	The reference architecture must enable development of testing tools that provide mechanisms to automatically generate test cases.	TCa
R-T4	The reference architecture must allow development of testing tools that execute test artifacts using test cases.	TA
...	...	...
R-S1	The reference architecture must enable development of testing tools that are capable of storing and providing normative descriptions related to their correct use.	SD
R-S2	The reference architecture must allow development of testing tools that support publication of service description directly to consumers as well as through mediators.	SP
R-S3	The reference architecture must enable development of testing tools that can be used directly as well as through service bus.	SI
R-S4	The reference architecture must allow development of testing tools that can be built by means of service composition, as business processes, using service orchestration.	SC
R-S5	The reference architecture must allow development of testing tools that can be built by means of service composition, coordinating the collaboration among services using choreography.	SC
R-S6	The reference architecture must enable development of testing tools that have or are able to use mechanisms to capture, monitor, log, and signal non-compliance with non-functional requirements described in service agreements.	QoS
...	...	...

we found that concepts of the testing domain are: Test Requirement (TR), Test Case (TCa), Test Artifact (TA) and Testing Criterion (TCr). It is worth highlighting that these same concepts are established by a testing domain ontology, [32]. Besides that, we also found that concepts related to SOA are according to those previously presented in Table 1.

3.3 Step RA-3: Architectural Design

From the 39 architectural requirements and the concepts identified in the previous step, the architectural design of RefTEST-SOA was conducted. For this, we have firstly adopted the overall idea based on layers proposed by S3, since it is observed that S3 has become a "*de facto*" standard to organize service-oriented systems. Following, we have specialized the idea of S3 layers for the software testing domain using both the knowledge obtained from that domain and the structures found in service-oriented testing tools. Moreover, the four concepts of the testing domain previously found were inserted as core of the RefTEST-SOA. Figure 2 presents the general representation of RefTEST-SOA. Basically, it is composed by six logical elements that we have considered as layers:

- **Application Layer:** It contains concepts directly related to the testing domain. It is composed by four sets of services that we name testing services: (i) `Primary Testing Services` that refer to core services of a testing tool (i.e., they support management of test case, test criterion, test artifact, and test requirement); (ii) `Orthogonal Supporting Services` that support software engineering activities considered supporting activities by ISO/IEC

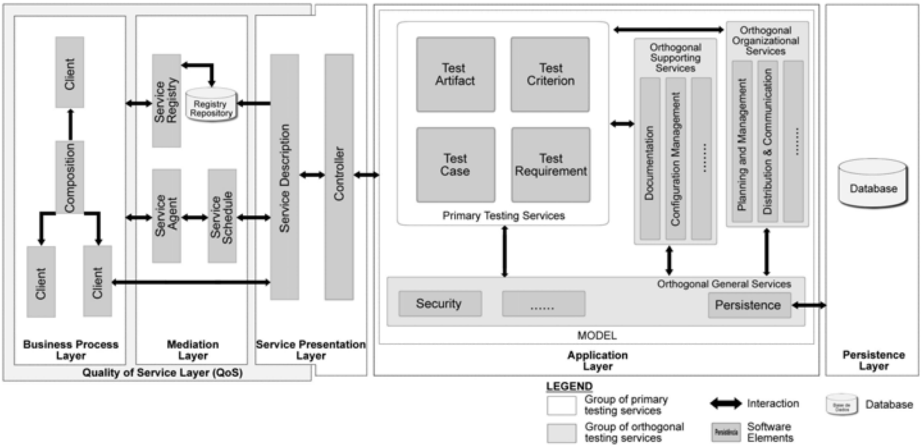


Fig. 2. General Representation of RefTEST-SOA

12207, such as documentation, and could be applied in the testing domain; (iii) `Orthogonal Organizational Services` that support software engineering activities considered organizational activities by ISO/IEC 12207, such as planning and management, and could also be applied in the testing domain; and (iv) `Orthogonal General Services` that refer to general services, such as `Persistence` and `Security`. It is important to highlight that these three last sets of services are considered orthogonal, since each service can be used by diverse services during a testing suite. For instance, `Documentation` service, responsible by documentation of a testing suite, can be required by other services, such as `Test Case` and `Test Criterion` services.

- **Persistence Layer:** It is responsible to store data produced by services that compose a testing tool. In particular, service `Persistence` acts directly in this layer;
- **Service Presentation Layer:** In stand-alone and web application, this layer processes usually user events and deals with visual presentation. However, in our reference architecture, which will underlie service-oriented systems, this layer is part of each service and contains two main elements: (i) `Service Description` that defines data format and deals with processed data that are received from clients; and (ii) `Controller` that deals with requests that are received from service clients. In order to convert data from service requests to adequate formats to be processed by `Application Layer`, `Controller` uses a service engine;
- **Mediation Layer:** In this layer, testing services are published, discovered, associated, and available. This layer is composed by three elements: (i) `Service Registry` that receives description of the testing services and enables search for them. This registry can be implemented as a *broker* or a *matchmaker* [27]; (ii) `Service Agent` that is a mediator, routing, mediating, and transporting requests from service requester to correct service provider, allowing indirect

communication among them. ESB could be used to implement this mediator; and (iii) `Service Schedule` is responsible to process service requests according to dependency among services;

- **Business Process Layer:** In this layer, business processes are defined. Thus, composed services are build based on the testing services contained in the `Application Layer`. For this, orchestration and choreography could be used. In order to orchestrate processes, standard languages, such as WS-BPEL (Web Services Business Process Execution Language), are available. Regarding use of choreography, WS-CDL (Web Services Choreography Description Language) could be used. Since that RefTEST-SOA is a reference architecture, the way that services are organized by orchestration and choreography must not be described in this abstraction level. Such details should be defined during the design of each architectural instance; and
- **Quality of Service Layer:** It supervises compliance of the quality requirements contained in other layers (`Business Process Layer`, `Mediation Layer` and part of `Service Presentation Layer`, in particular the `Service Description`). In other words, it observes other layers and signals when a non-functional requirement is not fulfilled. Thus, reliability, management, availability, scalability, and security in services could be guaranteed. This layer is very important, since different institutions could develop testing services using different infrastructures and an adequate interaction among services must be therefore guaranteed.

Besides the general representation of RefTEST-SOA, in order to adequately document this architecture, three architectural views were built: module view, runtime view, and deployment view. For the sake of space, only the module and deployment views are presented herein. The module view shows the structure of the software systems in terms of code units; packages and dependency relationship can be used to represent this view. In Figure 3, it is presented the module view of RefTEST-SOA represented in Package Diagram of UML. Four sets of testing services are proposed by RefTEST-SOA: `primaryTestingServices`, `orthogonalSupportingServices`, `orthogonalOrganizationalServices` and `orthogonalGeneralServices`. Each testing service must be independently implemented, enabling composition of a more complex testing service. Other services — `serviceSchedule`, `serviceAgent`, `serviceRegistry` and `qualityOfService` — make possible that testing services work adequately. Package `presentation` manages the interface of services, enabling communication among them. This support infrastructure is similar to those used in reference architectures for other application domains, such as those presented in [18]. Besides that, `client` is illustrated in this figure in order to show how client service can use the testing services. It is important to notice that, since RefTEST-SOA is based on SOA, packages in this view, for instance, service `testCase` and `qualityOfService`, could be physically distributed in different servers and even in different institutions.

In more details, if the client service does not know the location where the required testing service is, it must firstly search for this service in the `ServiceRegistry`, using a standard protocol, such as UDDI (Universal

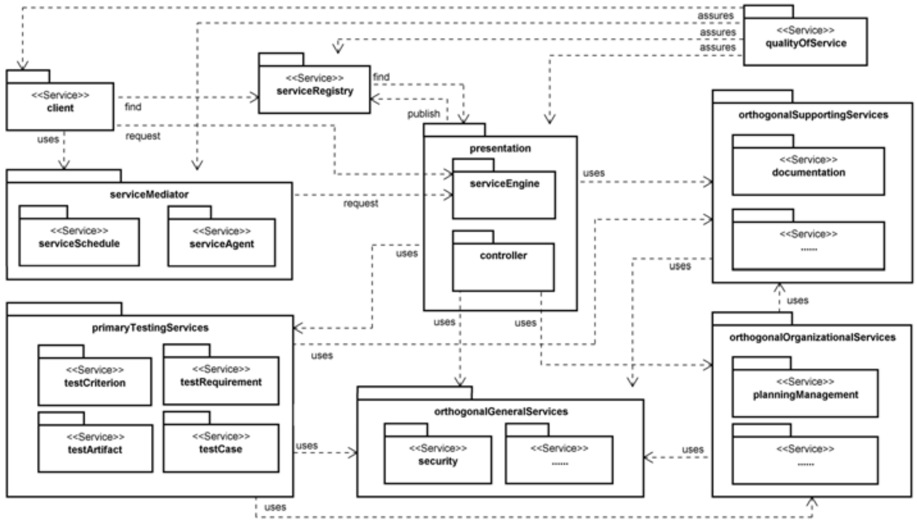


Fig. 3. Module View of RefTEST-SOA

Description, Discovery, and Integration). If the testing service is found, the `ServiceRegistry` informs the address (i.e., *endpoint*) to the client. Thus, a connection between the client and the required service is established. Following the client requests information related to the service description to the required service, aiming at an adequate communication. A description language, such as WSDL (Web Services Description Language), could be used to describe the services. From that, the communication between client and required service could be direct or through intermediate services (using the `ServiceAgent`). In both cases, the communication must be performed using a protocol and a standard language, for instance, SOAP (Simple Object Access Protocol). It is observed that a required service is any testing service proposed by RefTEST-SOA. In order to ensure quality in the interaction between the client and the required service, the `QualityOfService` monitors the communication.

Deployment view, describes the machines, software that is installed on those machines and network connections that are made available. This view is particularly useful, since different testing services, client services, and other services could be available in separated machines. Figure 4 illustrates a possible deployment view of RefTEST-SOA, represented in Deployment Diagram of UML. It is observed that this view is similar to deployment view of traditional service-oriented systems. According to this view, `Application Server` contains testing services; furthermore, a service engine, such as AXIS2[3], is available in the server. `Service Registry Server` is responsible to manage and store information about testing services and processes requests coming from client services. It uses

[3] http://ws.apache.org/axis2

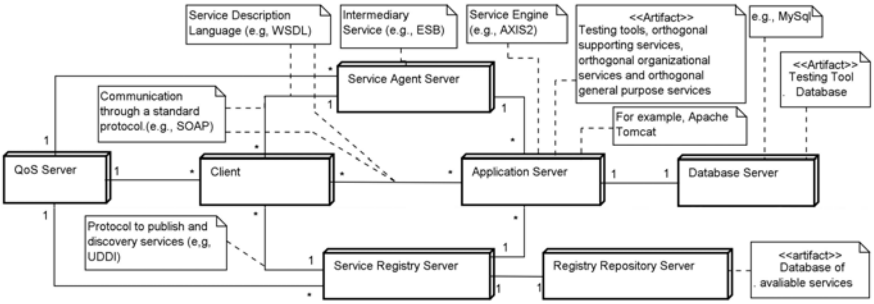

Service Description Language (e.g, WSDL)
Intermediary Service (e.g., ESB)
Service Engine (e.g., AXIS2)
<<Artifact>> Testing tools, orthogonal supporting services, orthogonal organizational services and orthogonal general purpose services
e.g., MySql
<<Artifact>> Testing Tool Database
Service Agent Server
Communication through a standard protocol.(e.g., SOAP)
For example, Apache Tomcat
QoS Server
Client
Application Server
Database Server
Protocol to publish and discovery services (e.g, UDDI)
Service Registry Server
Registry Repository Server
<<artifact>> Database of avaliable services

Fig. 4. Deployment View of RefTEST-SOA

a `Registry Repository Server` to store information about published services. Besides these servers, other servers contain an service agent (`Service Agent Server`) and a service of QoS (`QoS Server`).

3.4 Step RA-4: Reference Architecture Evaluation

Aiming at improving the quality of RefTEST-SOA, an inspection based on checklist was conducted. This checklist makes possible to look for defects related to omission, ambiguity, inconsistence and incorrect information that can be present in the architecture. Besides that, aiming at observing the viability of RefTEST-SOA, as well as its reuse and service integration capabilities, we have conducted case studies, as that presented in the next section.

4 Case Study

We developed a testing tool organized as services that supports application of the Mutation Testing [33]. In short, Mutation Testing is a fault-based testing criterion which relies on typical mistakes programmers make during the software development [33]. This criterion relies on the Competent Programmer and the Coupling Effect hypotheses [33]. These hypotheses state that a program under test contains only small syntactic faults and that complex faults result from the combination of them. Fixing the small faults will probably solve the complex ones. Given an original program P, the criterion requires creation of a set M of mutants, consisting of slightly modified versions of P. Mutation operators encapsulate modification rules applied to P. Then, for each mutant m, $(m \in M)$, tester runs the test suite T originally designed for P. If there is a test case t, $(t \in T)$, and $m(t) \neq P(t)$, this mutant is considered dead. If not, tester should improve T with a test case that reveals the difference between m and P. If m and P are equivalent, then $P(t) = m(t)$ for all test cases. This criterion has been empirically shown to be one of the strongest testing criteria [34].

To develop our testing tool, we reused the knowledge contained in RefTEST-SOA. In particular, we adopted the overall structure of RefTEST-SOA, that is based on layers, to organize the structure and distribution of the services to be implemented. Moreover, we reused all functional requirements related to the testing domain to establish the functionalities to be available in these services; furthermore, these requirements were specialized to the Mutation Testing criterion, in order to support this technique. To organize these services, we used orchestration, where interaction is coordinated by a specific service.

4.1 Description of the Testing Services

Aiming to build the core services for the testing domain (represented as `primaryTestingServices` in Figure 3), four services was implemented:

- **Test Case:** it refers to service named TestCaseManagement (TCM) that has as main objective to manage test case sets. This service is responsible to: (i) add test cases in the set; (ii) remove test cases of the set; (iii) update test cases; (iv) provide description about test cases; and (v) list test cases that have not been used yet;
- **Test Artifact:** this service, named MuTestPascalArtifact (MTPA), is responsible to: (i) read the program to be tested; (ii) generate the syntactic tree related to the program; (iii) compile the source code; and (iv) execute the program using test cases;
- **Test Requirement:** this service — the MuTestPascalRequirement (MTPR) — is responsible to execute functionalities related to treatment of test requirements. Thus, it: (i) generates mutants using mutation operators; (ii) compiles source code of mutants; (iii) provides mutation descriptions to the client service; (iv) executes mutants using test cases; and (v) mark/unmark mutants as equivalent; and
- **Test Criteria:** this service, named MuTestCriteria (MTC), implements functionalities related to Mutation Testing criterion: (i) verification of the number of alive, dead, and equivalent mutants; (ii) mutation score calculation, what can indicate adequacy of the test case set; and (iii) generation of reports about testing execution.

The integration of these four services was also conducted, resulting in a service-oriented testing tool. Thus far, this tool enables to test programs written in Pascal. However, it is worth noting that it can be easily extended to support other programming languages, since services can be additionally developed and integrated.

4.2 Integrating Testing Services

To build the service-oriented testing tool, we have developed an orchestrator service, named RExMuTesT(*Reusable and Extensible Mutation Testing Tool*) that coordinates and solicits functionalities of the four testing services. Figure 5 presents an example of a business process used by the tool. Dashed arrows represent messages; continuous arrows are related to the sequence of activities; and each column represents a testing service previously described (MTPR, MTPA,

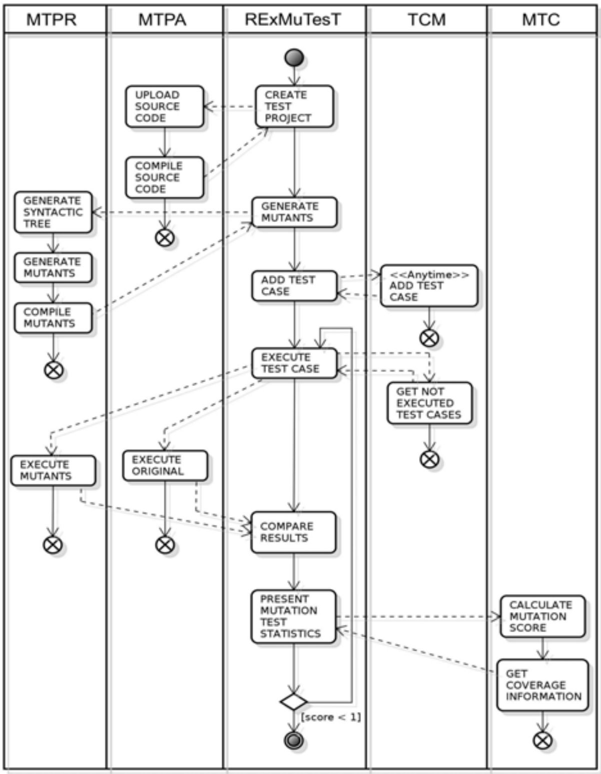


Fig. 5. Business Process of RExMuTesT

TCM, and MTC). In the central column, RExMuTesT is presented, coordinating other services.

When a client service (an user, an application or another business process) creates a test project and loads the source code of a program to be tested, RExMuTesT sends a message to test artifact service (MTPA), requiring treatment of the source code. MTPA is again solicited to generate the syntactic tree related to the source code and compiles the source code. Following, mutants are generated and posteriorly compiled by test requirement service (MTPR). Furthermore, test cases can be inserted anytime into test project using test case service (TCM). In order to test the program, a message is sent to test case service (TCM), requesting the list of test cases not executed yet. In test artifact service (MTPA), each test case is used to execute the original program to get the expected output. Following, test requirement service (MTRP) executes alive mutants using test cases. If output of a mutant is different from the expected output, this mutant is considered dead. Meanwhile, mutation score can be calculated by test criterion service (MTC); moreover, general information about test requirement coverage can be obtained. Thus, while the test case set is not considered adequate, new test cases can be added and the original program and alive mutants are executed.

In short, regarding technologies used to implement our services, we have adopted Apache Tomcat as the application server and AXIS2 as the service engine. Java was adopted as a general purpose language and JavaCC[4] as the parser generator. Design patterns were also considered in our implementation: Mediator for implementation of Controller and Facade to simplify use of functionalities provided by services.

4.3 Preliminary Analysis about Testing Service Reuse

Since business processes could also be services, we intend that RExMuTesT is easily integrated in other applications. Thus, this characteristic could make possible the development of an integrated testing environment, where different testing techniques and criteria are applied in a complementary way. Table 3 presents a preliminary analysis about reuse capability of the four testing services (MTPA, MTPR, MTC, and TCM).

Table 3. Service reuse level analysis

Reuse level	TCM	MTC	MTPA	MTPR
Similar Tool	✓	✓	✓	✓
Technique/Criterion	✓	–	–	–
Programming Language	✓	✓	–	–

Since test case service (TCM) was implemented to be independent of testing techniques/criterion (i.e., techniques/criterion-independent), as well as of language that is written the program to be tested (i.e., language-independent), it presents great reuse capability in other testing tools that support different testing techniques/criteria and languages. Test criterion service (MTC) requires and deals with information about test requirements (i.e., the mutants). Thus, this service could be reused in tools that support Mutation Testing; however, this service is language-independent; therefore, it could be reused in tools that test program in different languages, such as Java. Otherwise, test requirement service (MTPR) and test artifact service (MTPA) are directly related to the testing criterion that they support (Testing Mutation). Thus, they can be reused in similar tools, supporting Mutation Testing in program written in Pascal, however, in tools based on other service compositions or architectures. It is important to observe that this preliminary analysis refers specifically to the four services presented in this case study. Other services developed based on RefTEST-SOA will certainly present other reuse levels.

5 Conclusion and Future Work

Service-oriented reference architectures can bring a significant contribution to areas in which software systems based on SOA need to be built. For the testing

[4] https://javacc.dev.java.net/

domain, development of service-oriented testing tools is also a real need; thus, the main contribution of this paper is RefTEST-SOA, a service-oriented reference architecture for that domain. We believe that since it is possible to reuse the knowledge (structure and requirements) contained in RefTEST-SOA to develop new testing tools, reduction of efforts and time and, as a consequence, improvement in productivity could be achieved using RefTEST-SOA if compared with development from scratch. However, establishment of a service-oriented reference architecture is not a trivial task. Considerable sources of information are required to achieve a consolidated and relevant reference architecture. In this perspective, we intend that RefTEST-SOA reaches this objective, considering the systematic way that we adopted to establish this architecture. We intend also this same experience could be reproduced in other domains, aiming at establishing service-oriented reference architecture for such domains.

Our case studies have pointed out that testing tools traditionally developed as stand-alone systems can be built as a set of independent services. This observation could be considered for other domains that do not have yet service-oriented systems. Furthermore, a preliminary and qualitative observation indicates that services developed according to RefTEST-SOA present capacities of reuse, integration, and scalability provide by SOA. However, other services must be built yet, for instance, services that support orthogonal activities, such as testing documentation and testing planning. Therefore, more studies must be conducted to reach a complete and quantitative evaluation of RefTEST-SOA. For the future work, we intend to develop a set of testing services based on RefTEST-SOA, aiming an integrated testing environment and, as a consequence, contributing to the software testing activity.

Acknowledgments. This work is supported by Brazilian funding agencies (CNPq and CAPES) and Project "Process to the Establishment and Evaluation of Service-Oriented Reference Architectures" (Process N.: 2009/11135-9) supported by FAPESP.

References

1. Harrold, M.J.: Testing: A roadmap. In: ICSE 2000, pp. 61–72. ACM Press, New York (2000)
2. Myers, G.J., Sandler, C., Badgett, T., Thomas, T.M.: The Art of Software Testing. John Wiley & Sons, Inc., New Jersey (2004)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (2003)
4. Angelov, S., Grefen, P.W.P.J., Greefhorst, D.: A classification of software reference architectures: Analyzing their success and effectiveness. In: WICSA 2009, Cambridge, UK, pp. 141–150 (September 2009)
5. Eickelmann, N.S., Richardson, D.J.: An evaluation of software test environment architectures. In: ICSE 1996, Berlin, Germany (March 1996)
6. Nakagawa, E.Y., Simão, A.S., Ferrari, F., Maldonado, J.C.: Towards a reference architecture for software testing tools. In: SEKE 2007, Boston, USA, pp. 1–6 (July 2007)

7. Papazoglou, M.P., Heuvel, W.-J.: Service oriented architectures: approaches, technologies and research issues. The VLDB Journal 16(3), 389–415 (2007)
8. Bartolini, C., Bertolino, A., Marchetti, E.: Introducing service-oriented coverage testing. In: ASE 2008, L'Aquila, Italy, pp. 57–64. IEEE, Los Alamitos (2008)
9. Eler, M.M., Endo, A.T., Masiero, P.C., Delamaro, M.E., Maldonado, J.C., Vincenzi, A.M.R., Chaim, M.L., Beder, D.M.: JaBUTiService: A Web Service for Structural Testing of Java Programs. In: SEW 2009, Sweden, pp. 1–9 (2009)
10. Bayer, J., Forster, T., Ganesan, D., Girard, J.F., John, I., Knodel, J., Kolb, R., Muthig, D.: Definition of reference architectures based on existing systems. Technical Report 034.04/E, Fraunhofer IESE (2004)
11. Nakagawa, E.Y., Martins, R.M., Felizardo, K., Maldonado, J.C.: Towards a process to design aspect-oriented reference architectures. In: CLEI 2009, Brazil, pp. 1–10 (2009)
12. Schmidt, M.-T., Hutchison, B., Lambros, P., Phippen, R.: The enterprise service bus: making service-oriented architecture real. IBM Systems Journal 44(4), 781–797 (2005)
13. Peltz, C.: Web Services Orchestration and Choreography. IEEE Computer 36(10), 46–52 (2003)
14. Dai, G., Bai, X., Wang, Y., Dai, F.: Contract-based testing for web services. In: COMPSAC 2007, Washington, USA, vol. 1, pp. 517–526 (July 2007)
15. Arsanjani, A., Zhang, L.J., Ellis, M., Allam, A., Channabasavaiah, K.: S3: A service-oriented reference architecture. IT Professional 9(3), 10–17 (2007)
16. OASIS: Reference model for service oriented architecture 1.0. Technical report, OASIS Standard (October 2006)
17. OASIS: Reference architecture for service oriented architecture version 1.0. Technical report, OASIS Standard (April 2008)
18. Oliveira, L.B.R., Felizardo, K.R., Feitosa, D., Nakagawa, E.Y.: Reference models and reference architectures based on service-oriented architecture: A systematic review. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 360–367. Springer, Heidelberg (2010)
19. Costagliola, G., Ferrucci, F., Fuccella, V.: SCORM run-time environment as a service. In: ICWE 2006, New York, NY, USA, pp. 103–110 (2006)
20. Peristeras, V., Fradinho, M., Lee, D., Prinz, W., Ruland, R., Iqbal, K., Decker, S.: CERA: A collaborative environment reference architecture for interoperable CWE systems. Service Oriented Computing and Applications 3(1), 3–23 (2009)
21. Eler, M.M., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Built-in structural testing of web services. In: CBSoft 2010, Los Alamitos, CA, USA, pp. 70–79 (2010)
22. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report (2007)
23. Nakagawa, E.Y., Oliveira, L.B.R.: Using systematic review to elicit requirements of reference architectures. In: WER 2011, Rio de Janeiro, Brazil, pp. 1–12 (April 2011)
24. Dieste, O., López, M., Ramos, F.: Formalizing a systematic review process in requirements engineering. In: WER 2007, Brazil, pp. 96–103 (2007)
25. Baldamusa, M., Bengtsona, J., Ferrari, G., Raggi, R.: Web services as a new approach to distributing and coordinating semantics-based verification toolkits. In: WSFM 2004, Pisa, Italy (February 2004)
26. Ghezzi, G., Gall, H.: Towards software analysis as a service. In: ASE 2008, L'Aquila, Italy, pp. 1–10 (2008)

27. Dillon, T.S., Wu, C., Chang, E.: Reference architectural styles for service-oriented computing. In: ICNPC/IFIP 2007, Dalian, China, pp. 543–555. Springer, Heidelberg (2007)
28. Lan, J., Liu, Y., Chai, Y.: A solution model for service-oriented architecture. In: WCICA 2008, Chongqing, China, pp. 4184–4189 (June 2008)
29. Zimmermann, O., Kopp, P., Pappe, S.: Architectural knowledge in an SOA infrastructure reference architecture. In: Software Architecture Knowledge Management, pp. 217–241. Springer, Heidenberg (2009)
30. International Organization for Standardization: Information technology – software life–cycle processes. Technical report, ISO/IEC 12207 (1995)
31. Ferrari, F.C., Nakagawa, E.Y., Rashid, A., Maldonado, J.C.: Automating the mutation testing of aspect-oriented Java programs. In: AST 2010 at ICSE 2010, Cape Town, South Africa, pp. 51–58 (2010)
32. Barbosa, E.F., Nakagawa, E.Y., Maldonado, J.C.: Towards the establishment of an ontology of software testing. In: SEKE 2006, San Francisco Bay, USA (July 2006)
33. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 34–43 (1978)
34. Li, N., Praphamontripong, U., Offutt, A.J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: Mutation 2009 at ICST 2009, Denver, USA, pp. 220–229 (2009)

Decouplink: Dynamic Links for Java

Martin Rytter and Bo Nørregaard Jørgensen

The Maersk Mc-Kinney Moller Institute, University of Southern Denmark,
Campusvej 55, 5230 Odense M, Denmark
{mlrj,bnj}@mmmi.sdu.dk
http://www.sdu.dk/mmmi

Abstract. Software entities should be open for extension, but closed to modification. Unfortunately, unanticipated requirements emerging during software evolution makes it difficult to always enforce this principle. This situation poses a dilemma that is particularly important when considering component-based systems: On the one hand, violating the open/closed principle by allowing for modification compromises independent extensibility. On the other hand, trying to enforce the open/closed principle by prohibiting modification precludes unanticipated dimensions of extension. Dynamic links increase the number of dimensions of extension that can be exploited without performing modification of existing types. Thus, dynamic links make it possible to enforce the open/closed principle in situations where it would otherwise not be possible. We present Decouplink – a library-based implementation of dynamic links for Java. We also present experience with the use of dynamic links during the evolution of a component-based control system.

Keywords: Dynamic links, extensibility, object-oriented programming.

1 Introduction

The inability to close software components to modification poses a threat to the extensibility of software systems [27]. Ideally, individual software components are open for extension, but closed to modification [17,15].

The need for modification arises when a software component does not comply with its specification – i.e. due to a bug – or when there is a need to incorporate new requirements. Whereas bugs rarely pose an enduring problem, the need to incorporate new requirements does. This is so, because all non-trivial software systems are subject to uncertainty, which requires them to evolve in ways that cannot be anticipated [12,13,3]. Thus, the need for modification to accommodate extension is usually an enduring problem.

Modifications that introduce new functionality are not only enduring, they also tend to be more difficult to confine. Whereas correcting a bug can often be confined so that dependent components remain unaffected, incorporating new functionality is more likely to affect existing components.

Software evolution implies that a software system must change in order to support new requirements. However, components inside the system do not per se

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 422–437, 2011.

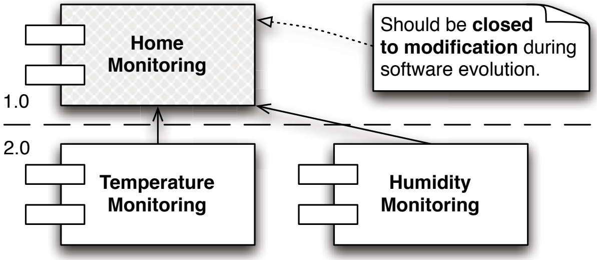


Fig. 1. Evolution of a component-based home monitoring system

need to change. In the best case, new functionality can be introduced by adding new components, while existing components remain closed to modification.

We will use the simple home monitoring system in figure 1 to discuss the open/closed principle – in figure 6 we will share experience from the evolution of a real system. To start with, the home monitoring system consists of a single component, i.e. `home monitoring`. In its next version, two new components are added to the system, i.e. `temperature monitoring` and `humidity monitoring`.

To satisfy the open/closed principle, it must be possible to add the two new components in the system without modifying the `home monitoring` component. Unfortunately, it is not always possible to anticipate those dimensions of extension – i.e. "kinds of" extension – that will be needed in the future. When this is the case, extension developers are faced with an inconvenient dilemma:

One the one hand, an extension developer – e.g. the developer of `temperature monitoring` – may choose to violate the open/closed principle by performing modification of an existing component – e.g. `home monitoring` – to facilitate introduction of the extension. Even if the required modification seems to be backwards compatible, the fact that it is made by an extension developer makes it problematic. The problem is that other extensions may require similar modifications that could potentially produce a conflict. Therefore, the composition of invasive extensions must entail a global integrity check, and thus extensions cannot be completely independent – i.e. the system fails to be independently extensible [26].

On the other hand, an extension developer may refrain from any modification of existing components – i.e. `home monitoring` remains closed to modification. This decision implies that the required extension – e.g. `temperature monitoring` – cannot be introduced. Thus, the open/closed principle is violated as the system fails to be open for extension.

In summary, enforcement of the open/closed principle relies on anticipating required dimensions of extension. The ability to do so is one of the most important skills for a software architect to master. Nevertheless, even the most skilled software architect can never anticipate everything – thus, in the ultimate case the open/closed principle cannot be enforced.

An important feature of component platforms is the ability to handle situations, where modification of existing components is unavoidable. This is traditionally done by implementing a component lifecycle management system that maintains dependencies among component versions [16]. While versioning is certainly always an option, it should be the last option. In general, it is best if modification of existing components can be avoided.

In this paper, we argue that the need for modification of existing components can be reduced. It is often the case that existing components must be modified, not to change existing functionality in any fundamental way, but to allow new components to associate new functionality with concepts managed by existing components. We will demonstrate that this form of modification can be avoided – and thus our ability to satisfy the open/closed principle can be increased.

The core of our approach is dynamic links – a new kind of link that can connect objects of unrelated types. Dynamic links promote both elements of the open/closed principle: First, dynamic links promote openness by allowing new objects to be attached to old objects in ways that were not anticipated. Second, dynamic links can connect objects without modifying their types – existing types remain closed to modification.

The paper is a continuation of preliminary work described in [23]. It provides two main contributions: We present Decouplink [1] – a library-based implementation of dynamic links for Java [2] – and we present experience with the use of dynamic links to evolve a component-based control system for greenhouse climate control.

The paper is organized as follows. We introduce dynamic links in section 2. Section 3 presents support for dynamic links in Java. In section 4, we present experience with the use of dynamic links during the evolution of a greenhouse control system. Section 5 presents related work. Section 6 concludes the paper.

2 Dynamic Links

In this section we introduce dynamic links, we discuss how they are different from traditional links, and we demonstrate that dynamic links promote the open/closed principle.

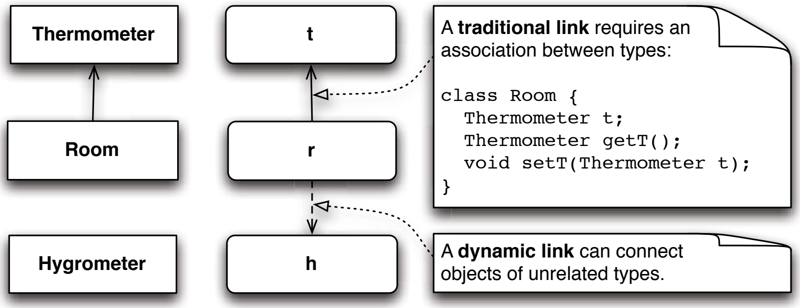
In object-oriented software, a *link* is a connection between two objects. A link usually has a direction and connects exactly two objects – a *source object* and a *destination object*. Links enable us to represent complex domain concepts as compositions of primitive objects connected by links. The use of a link between objects commonly relies on an *association* between types. The relationship between object-based links and type-based associations is emphasized in the UML specification [19]:

> "An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end."

[1] Get Decouplink from `http://decouplink.com`.

Given the definition above, a traditional link may be thought of as "an instance of" an association. It is only possible to create a link when a corresponding association exists. In Java, an association usually manifests itself as a field. E.g. it is only possible to connect a Room object and a Thermometer object when a suitable field has been declared, e.g. Room.thermometer.

A *dynamic link* can connect objects of unrelated types. Unlike a traditional link, a dynamic link does not rely on the declaration of an association. It is therefore possible to create a dynamic link between any two objects.


Thermometer
Room
Hygrometer
t
r
h
A **traditional link** requires an association between types:
A **dynamic link** can connect objects of unrelated types.

```
class Room {
  Thermometer t;
  Thermometer getT();
  void setT(Thermometer t);
}
```

Fig. 2. Comparison of traditional links and dynamic links

The difference between traditional links and dynamic links is illustrated in figure 2. The example shows three objects. The r object is an instance of the Room type – similarly, t is an instance of Thermometer, and h is an instance of Hygrometer. The figure shows two links:

First, r and t are connected by a traditional link. The link can exist only because a corresponding association exists between Room and Thermometer. In the code, the association is implemented using a field and two accessor methods.

Second, r and h is connected by a dynamic link. The dynamic link is drawn using a dashed line. The link is possible even though Room and Hygrometer do not participate in a type-based association. Thus, no methods or fields in the Room type depend on the Hygrometer type.

The primary advantage of dynamic links is that they promote closing existing code to modification. This is the case because they, unlike traditional links, can connect new objects of unanticipated types without modifying existing types.

Figure 3 illustrates how dynamic links promote closing components to modification in situations where traditional links do not. The three components are similar to those in figure 1, and the types and objects provided by each component are similar to those in figure 2.

First, the temperature monitoring component uses a traditional link to extend the system. The link connects t, an instance of the new type Thermometer, to r, an instance of Room. As we have previously seen, this link can only be created when there exists an association between Room and Thermometer. Thus, modification of the original home monitoring component is required.

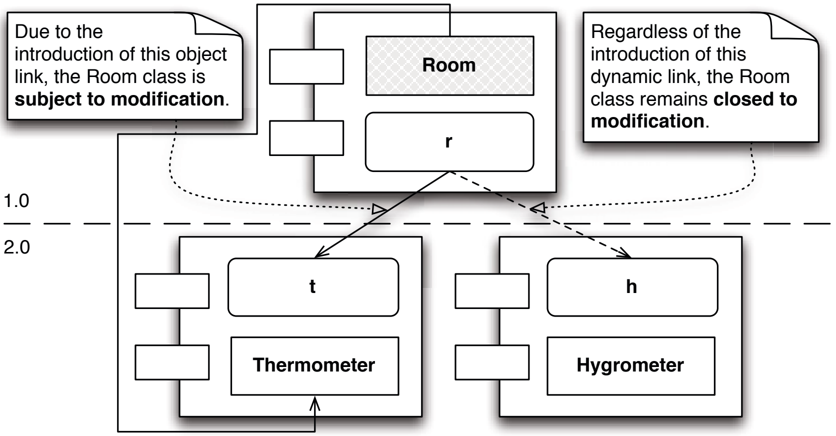

Due to the introduction of this object link, the Room class is **subject to modification**.
Room
r
Regardless of the introduction of this dynamic link, the Room class remains **closed to modification**.
1.0
2.0
t
Thermometer
h
Hygrometer

Fig. 3. Dynamic links promote closing components to modification

Second, the `humidity monitoring` component uses a dynamic link to extend the system. It connects `h`, a `Hygrometer`, to `r`, a `Room`. Since the new link requires no corresponding association, no modification of `Room` is required to perform the extension – the `home monitoring` component remains closed to modification.

The benefits and limitations of dynamic links can be emphasized by distinguishing two kinds of extension:

- *Unanticipated structural extension* is the ability to create links from original objects to new objects of unanticipated types – e.g. "add a hygrometer to a room". Unanticipated structural extension is supported by dynamic links.
- *Unanticipated behavioral extension* is the ability to wrap unanticipated behavior around original behavior – e.g. "when the light is turned on, also turn on the heat". Unanticipated behavioral extension is not supported by dynamic links.

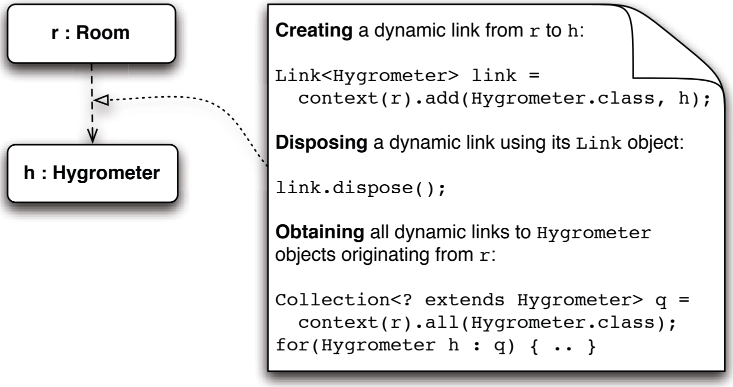
Unanticipated behavioral extension can only be achieved by modifying original types. This modification may be explicitly performed by the programmer – e.g. direct modification of source code. It may also be automated – e.g. load-time weaving of aspects [10]. Even automated modifications should be avoided to preserve independent extensibility [20].

The use of dynamic links is to some extent analogous to the way we "connect objects" in the physical world. The architect of a room is likely to create a room layout (a room type) without thinking about hygrometers (an associated type) – however, this does not prohibit a future owner of a room (an instance) from installing one. Similarly, type developers can never anticipate everything – should this prohibit object owners from creating links? As indicated above, we do not think so. However, we must stress not to use a comparison with the physical world to be an argument for or against dynamic links. We use the comparison merely to offer a familiar way of thinking about the role of dynamic links.

3 Design and Implementation

In this section we will present Decouplink – our implementation of dynamic links for Java. We show simple usage examples, we discuss the most notable design decisions, and we give an overview of how it works.

We have implemented Decoupling as a library. This choice makes the implementation accessible, as no language extension is needed.



Creating a dynamic link from r to h:

```
Link<Hygrometer> link =
  context(r).add(Hygrometer.class, h);
```

Disposing a dynamic link using its Link object:

```
link.dispose();
```

Obtaining all dynamic links to Hygrometer objects originating from r:

```
Collection<? extends Hygrometer> q =
  context(r).all(Hygrometer.class);
for(Hygrometer h : q) { .. }
```

Fig. 4. Simple usage of dynamic links

Figure 4 shows how to use our library to create, dispose, and obtain dynamic links. The `context()` method plays an important role. It is used to select an object on which to perform an operation, e.g. create a link or obtain existing links. The `context()` method is static and provided by a class in our library. By statically importing the method, it can be made available anywhere.

Since dynamic links do not rely on type-level associations, it is not possible to qualify links using accessor methods. Instead, we rely on *type-based link qualification*, i.e. we qualify links by the type of their destination object – not the name of a method or field. Consequently, instead of writing `r.addHygrometer(h)`, we write `context(r).add(Hygrometer.class, h)`. Type-based links qualification has two important consequences:

- It is always possible to add links to objects of new types without modifying existing types.
- The type of a destination object must be sufficiently specific to reveal the purpose of the data it represents. E.g. a person's first name should probably be of type `FirstName` and not merely `String`.

When programming an extension that adds new objects using dynamic links, it is often useful to be able to "protect" object links, so that other extensions cannot remove them. E.g. the `humidity monitoring` component should be able to ensure that no other extension intentionally or unintentionally disposes the link from `r` to `h`. We achieve this form of protection using *objectified link ownership*:

- Creating a dynamic link produces a Link object (see figure 4).
- A dynamic link can only be disposed through its corresponding Link object.

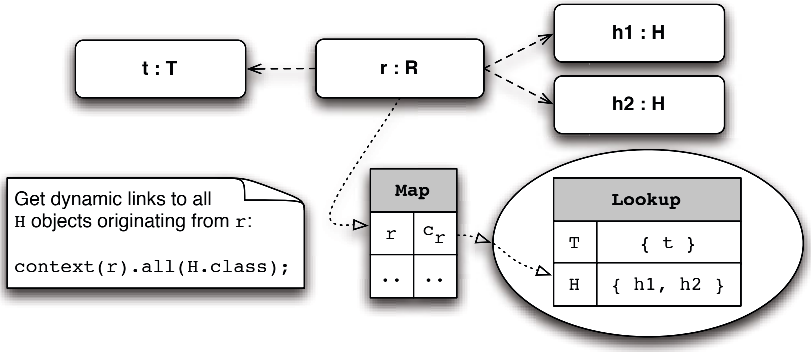
Note that a Link object represents ownership of a link – not the ownership of any particular object. Anyone with access to an object can navigate dynamic links originating from that object, but only the links' owners can dispose them.

When using traditional links, it is possible to enforce constraints on the cardinality between types. E.g. "a Room has exactly one Hygrometer". When using dynamic links, it cannot always be guaranteed that a future extension will not break such cardinality constraints. E.g. a future extension may add a second Hygrometer. In section 4 we will discuss a pattern that can enforce cardinality constraints in certain situations. However, as a general rule:

- Dynamic links are not constrained by type-level association. Therefore, design for "one-to-many" whenever it is practical.

We have already seen that the context() method is an essential part of our API. This method provides access to a simple runtime system that manages dynamic links. An overview of the runtime-system implementation is given in figure 5. The example is based on a situation where a Room has a single Thermometer and two Hygrometers. To improve readability, we have abbreviated the classnames used in previous examples – e.g. Room is abbreviated R.

The runtime system uses a systemwide map to associate each source object, e.g. r, with a corresponding *context object*, e.g. c$_r$. The context object holds information about dynamic links originating from its corresponding source object. The context objects are lazily created – i.e. c$_r$ is created when context(r) is first called. The map is a weak hash map – i.e. a context object is made eligible for garbage collection even if the global map keeps a reference to it. Consequently, developers do not have to rely on explicit link disposal – dynamic links automatically disappear when their source objects disappear.



Get dynamic links to all H objects originating from r:

```
context(r).all(H.class);
```

Fig. 5. Dynamic links runtime system

Each context object organizes dynamic links using a lookup. The lookup associates each destination object type, e.g. `H`, with a list of destination objects, e.g. {`h1, h2`}. Adding and disposing dynamic links correspond to changing the contents of the lookup. Obtaining links corresponds to accessing the lookup.

To summarize, let us consider evaluation of the `context(r).all(H.class)` statement by following the dotted lines in figure 5. First, `context(r)` corresponds to accessing the map and returning the corresponding context object, $\mathtt{c}_r$ – if no context object exists, it is lazily created. Second, `all(H.class)` returns a collection of all `H` objects in the context object's lookup.

Whereas obtaining and creating links is supported by methods invoked on the context object, link disposal is different. As discussed previously, link disposal happens exclusively through the `Link` object (not shown in figure 5). Thus, if the creator of a dynamic link does not keep a reference to the corresponding `Link` object, then the link – and the corresponding context information – can only disappear when the source object becomes eligible for garbage collection.

Before moving on, we would like to briefly mention a few features that space does not permit us to present in great detail:

First, it is often practical to manage ownership of groups of links that belong together – e.g. when a group of links must be disposed at the same time. Our library provides a small number of classes that support such management.

Second, fault tolerance is a crosscutting concern that may be difficult to maintain as component-based systems evolve. Our library allows for the creation of fault-tolerant dynamic links. A fault-tolerant dynamic link is a dynamic link that automatically tries to recover from a destination object's inability to satisfy its contract. This feature is motivated and inspired by [22].

4 Experience with Dynamic Links

The best evaluation of dynamic links available at the moment is experience gathered during the design, implementation, and evolution of a component-based control system for greenhouse climate control. An early version of the system was briefly mentioned in [23]. The system is currently composed of 19 components, 12 of which use dynamic links. The number of dynamic links in a running system depends on usage patterns. Normal usage easily generates more than 1,000 dynamic links, and those links may be obtained more that 500,000 times within a few minutes. The total size of the system is 12,919 lines of code.

The difficulty of anticipating required dimensions of extension, and thus enforcing the open/closed principle, is highly domain specific. In our experience, greenhouse climate control is a particularly challenging domain. First, the physical properties of individual greenhouses can be very different. Second, the set of sensors and actuators available vary greatly. Third, control requirements vary depending on the cultivar being produced and the grower's preferences.

Most research in the area of climate control has been focused on evaluating specific control strategies against various plant physiological criteria [31,30,28]. Attempts to integrate different control strategies into an extensible control system have turned out to be surprisingly difficult to perform [1].

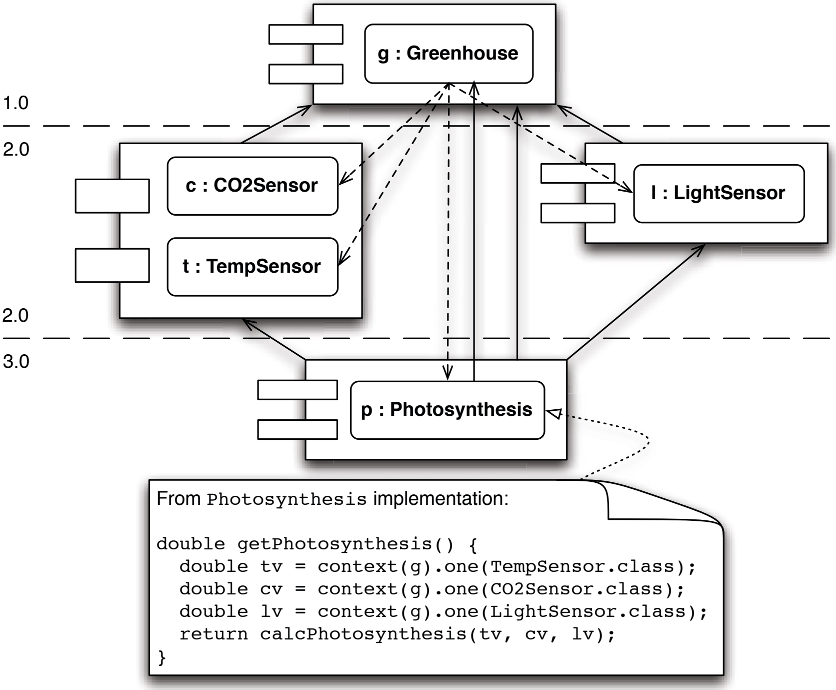
Our system is the result of a collaboration with growers, plant physiologists, and a control system vendor. We have been working on the system for two years. The concept of dynamic links has emerged during the project and plays a central role in recent versions of the system.

Figure 6 depicts selected components in the system, and some of their provided objects. For the purpose of our discussion we have organized the components in three versions – this is a simplification of the actual system's history. In the first version, a component provides `Greenhouse` objects (to improve readability only a single object is shown in the figure). In version two, two components provide CO_2, temperature, and ambient light sensors. Finally, version three adds a component that deals with photosynthesis – a measure of plant growth that can be calculated when light intensity, temperature, and the CO_2 level are known.

The figure contains two kinds of arrows: First, arrows for dependencies between components. Second, arrows for links between objects – note the difference between traditional links (normal lines) and dynamic links (dashed lines).

Based on figure 6 we will now discuss a number of concrete experiences:

- Dynamic links promote closing existing components to modification despite the presence of domain contexts, whose scopes cannot be fully anticipated.



From `Photosynthesis` implementation:

```
double getPhotosynthesis() {
  double tv = context(g).one(TempSensor.class);
  double cv = context(g).one(CO2Sensor.class);
  double lv = context(g).one(LightSensor.class);
  return calcPhotosynthesis(tv, cv, lv);
}
```

Fig. 6. Selected components in a greenhouse climate control system

In our system "a greenhouse" constitutes a domain context whose scope cannot be fully anticipated. In the broadest sense, a context is a setting in which statements may be interpreted [18]. E.g. in the context of a greenhouse we interpret statements such as "what is the temperature?" or "what is the current rate of photosynthesis?". We consider it impossible to come up with a complete list of statements that may be interpreted in the context of a greenhouse – i.e. the scope of a greenhouse context cannot be fully anticipated.

Without dynamic links, types representing domain contexts are difficult to close to modification. Addition of new context objects – e.g. `CO2Sensor` objects or `LightSensor` objects – would require modification of the `Greenhouse` type.

Note that new types of context information are not only difficult to anticipate, but can also be very different. Thus, it is difficult to extract common super types. In theory, we could use a pure tagging interface – e.g. `GreenhouseItem` – for all our unanticipated types to implement. This would actually promote closing `Greenhouse` to modification. However, since the types have very little in common, this solution would be difficult to manage for extension components, as it would often be necessary to use `instanceof` tests and typecasts to access objects using sufficiently specific interfaces.

With dynamic links, new objects can be non-invasively attached to objects of original types as the domain context they represent evolves. In figure 6, new components add links to instances of `CO2Sensor`, `TempSensor`, `LightSensor`, and `Photosynthesis`. In our system, links originating from `Greenhouse` objects refer to objects of 61 different types. Hence the `Greenhouse` type – and thus the component in which it resides – remains closed to modification.

In summary, the development style that dynamic links make possible requires developers merely to anticipate "the existence of a domain context", and "not specific dimensions of extension that must be supported by the context". Consequently, software becomes more extensible and remains closed to modification.

- Dynamic links support repeated extension, where each extension object can look up objects provided by other extensions.

It often happens that an object provided by one extension depends on objects provided by another extension. This leads to a form of repeated extension where dynamic links are used to incrementally construct a network of related objects around a common context object.

In figure 6, `p` provides the ability to calculate photosynthesis. The calculation depends on other extensions providing inputs such as temperature, `t` of type `TempSensor`, CO_2 level, `c` of type `CO2Sensor`, and light intensity, `i` of type `LightSensor`. Two things are important to note:

First, `g` represents a context in which the calculation takes place. There are many `TempSensor` objects, `CO2Sensor` objects, and `LightSensor` objects in a system. However, we need exactly those that can be found in the context of `g`.

Second, an extension can only find objects of types that are known. E.g. in order to obtain a dynamic link to `l`, it is necessary to depend on the component providing the `LightSensor` type.

The example shown in figure 6 is rather small, and thus the photosynthesis component depends on all other components being shown. A complete diagram of our system would reveal that most extensions depend only on a subset of components operating on the `Greenhouse` context – e.g. a user-interface component providing a thermometer widget needs only to know about `TempSensor` objects provided in the context of `g`. Each component may have its own incomplete view of a context, and multiple components' views may be overlapping.

In our experience, extension by attaching new objects facilitates interface segregation [14]. Dynamic links make it easy to add extension objects with "slim" interfaces, and thus clients depending on those interfaces often use all of it.

– Strive towards modeling your software so that invariants imposed by the domain do not depend on the existence of dynamic links.

The lack of class-based encapsulation makes it difficult to enforce an invariant that depends on the existence of a dynamic link. Fortunately, such invariants can almost always be avoided by taking appropriate design decisions.

In our system, we measure various values at regular intervals. The measured information is shared among components by using dynamic links originating from a `Greenhouse` object. Let us consider two different ways to implement this:

One approach is to update a measured value by replacing an object – e.g. we may dispose a dynamic link referring to an old `MeasuredCO2` object, and then create a dynamic link to a new `MeasuredCO2` object. In our experience, this implementation is often problematic, because it tends to violate invariants imposed by domain requirements. A simple invariant that may be violated is "a `Greenhouse` object must always have a `MeasuredCO2` object". Since dynamic links do not provide transaction-based creation and disposal, it is impossible to replace an object without violating the invariant. Similar problems may also emerge with more complex invariants involving more than one link.

Instead of continuously replacing a destination object, we prefer to change the state of the object. Instead of having a `MeasuredCO2` object that needs to be replaced when a new value has been measured, we use a `CO2Sensor` object that changes its internal state (see figure 6). The same `CO2Sensor` object is used throughout the lifetime of `g`. Since the state change takes place inside an object, we can enforce invariants using type-based encapsulation.

– The creation and disposal of dynamic links often coincide with creation and disposal of the object being extended. Therefore, the need for subscribing to creation and disposal events must often be anticipated.

Connecting two objects with a dynamic link – i.e. a structural extension – needs not to be anticipated by type developers. However, the need for adding new behavior to a control flow to create a link at a specific time – i.e. a behavioral extension – must be anticipated.

In our experience, the time at which a dynamic link must be created or disposed often – but not always – coincides with the time of creation and disposal of the object being extended. This is particularly the case when following our previous advice: When state changes take place inside referred objects – and not as creation/disposal/replacement of dynamic links – there is a tendency for referred objects to be created and disposed together with the object they extend.

In our system, the component responsible for managing `Greenhouse` objects makes it possible for extension components to be notified, when a `Greenhouse` object is created or disposed. We implement this using the observer pattern [8].

In our experience, the code needed to facilitate the required event notification is rarely subject to modification, even though it must once be anticipated. Thus, in practice our ability to close components to modification is rarely compromised.

Though it often happens, it is not always the case that dynamic links are created and disposed together with the object being extended. In some architectural styles an object may take the role of a message that is being passed around – e.g. pipes and filters [24]. In such cases each component handling a message may add new information using a dynamic link. In such designs a message may have a significantly longer lifetime than dynamic links used to extend it.

Our system reveals another exception from the general trend. The dynamic class-loading capability of Java allows our control system to have a software updating mechanism that can add components, while the system is running. When adding a new component, it is often necessary to add "new things" to greenhouses – i.e. new dynamic links are created, and they get to originate from `Greenhouse` objects that already exist. In our system, the component managing `Greenhouse` objects is responsible for organizing this.

– The lack of associations between types makes it important to document *sharing* and *co-existence* semantics when declaring types intended to be used with dynamic links.

The public part of a traditional type-based association manifests itself as type members – e.g. accessor and modifier methods. The names and documentation of these type members informally document the contract of that association. When there is no explicit association – as it is the case when using dynamic links – this form of documentation is not available. Consequently, the type of a destination object must provide documentation that is usually not needed or less important. In our experience, two aspects are particularly important to document:

First, a normal accessor method indicates whether ownership of returned objects is transferred to the caller – e.g. `Stack.pop()` – or if the returned objects are shared with the callee – e.g. `Stack.peak()`. With dynamic links, referred objects will almost always be shared. It is therefore important to document what happens when multiple independent units of code navigate the same dynamic link, and thus share access to a common destination object. E.g. consider the potential destination type `interface GreenhouseWindow { setOpen(boolean v); }`. `GreenhouseWindow` is probably not very useful to a ventilation component that wants to open the window for 30 minutes, since shared access enables another component to override the decision. Thus, developers should keep sharing in mind when designing and documenting types such as `GreenhouseWindow`.

Second, an association may document the roles of participating objects. With dynamic links, all objects of the same type have the same role. When multiple objects are referred to by dynamic links originating from a common source object, then we may say that they co-exist. It is important that the semantics of such co-existence is documented when declaring types of destination objects. E.g. when “a `Greenhouse` has multiple `TempSensors`”, then an association may assign roles to each `Thermometer` object, e.g. “near plants” or “near the ridge”. With dynamic links, co-existing `TempSensor` objects all have the same role. Thus, the `TempSensor` type must be declared, so that it makes sense to have co-existing instances. When this cannot be done, it is sometimes necessary to promote roles to types, e.g. to distinguish `PlantTempSensor` from `RidgeTempSensor`.

– *Cardinality constraints* can indirectly be achieved by limiting access to constructers of destination types. Cardinality constraints cannot be combined with abstraction.

So far, we have assumed that it is impossible to impose cardinality constraints between two types when using dynamic links. While this is to some extent true, an observation deserves to be mentioned: It is possible to indirectly impose cardinality constraints by declaring a type that cannot be instantiated directly by third-party classes or components.

Looking at figure 6, let us assume that we want to enforce that “a `Greenhouse` has exactly one `LightSensor`”. We can do this by preventing subclassing – i.e. declaring `LightSensor` to be `final` – and prohibiting other components from instantiating `LightSensor` objects – i.e. making all `LightSensor` constructors private. New `LightSensor` objects can now only be instantiated by the component providing the `LightSensor` type. Thus cardinality constraints maintained by the providing component cannot be violated by other components.

Note that this technique has an important limitation: It cannot be combined with abstraction across component boundaries. In other words, the component that enforces a cardinality constraint must also be the component that provides an implementation of the destination type.

Also note this pattern’s similarity with the singleton pattern [8] – both patterns prevent direct third-party instantiation.

In summary, it is our experience that dynamic links have the potential to promote the open/closed principle. To realize this potential it is important that programmers understand the benefits and limitations that dynamic links have to offer. We consider the experience presented here as a valuable starting point.

5 Related Work

The mechanisms by which dynamic links are created, obtained, and disposed are similar to the mechanisms by which objects are registered, discovered, and unregistered when using the lookup pattern [11]. The original motivation for the lookup pattern was the ability to discover distributed objects. Similarly, dynamic links can be used to discover objects provided by other components.

Some systems use lookups not merely to facilitate discovery, but to represent domain contexts with scopes that often change due to software evolution or software configuration. When used in this way, a system typically has many lookup instances, each representing something from the domain – e.g. a user, a company, or a greenhouse. The NetBeans Rich Client Platform was one of the first projects to use lookups successfully for this purpose [5]. It uses lookup instances to represent concepts such as folders (in file systems), projects, and nodes (in tree views). In such a system, lookups are only used to model selected domain contexts. With dynamic links, similar capabilities are available for any object in the system without any explicit introduction of the lookup pattern.

An approach to closing types to modification is to model an unanticipated association as a type in its own right – i.e. to use an association class [7]. Using this approach, "a `Room` has a `Thermometer`" can be modeled as "a `RoomThermometer-Association` has a `Room` and a `Thermometer`". While this approach is capable of avoiding modifications, it does involve quite a bit of unintuitive boilerplate code for declaring association classes and for managing association objects.

Another way to externalize associations is object-oriented support for relations [21] – a first-class concept inspired by the entity-relationship model used in database theory. Relations were not designed with the open/closed principle and independent components in mind. Therefore, no link-ownership mechanisms are discussed. This is, however, a prerequisite for independent extensibility. While relations as first-class concepts have attractive properties, we prefer a library-based approach, as it is easier to integrate with mainstream languages.

AspectJ [10], MultiJava [6], and many dynamic languages [9,29] support the addition of new fields and methods to existing types. We have previously noted that this form of modification compromises independent extensibility. A similar criticism can be found in [20] and [25].

Classboxes [4] also support the addition of new fields and methods, but their visibility is limited to a well-defined scope – i.e. a classbox. This makes it possible to introduce extensions to existing types without affecting existing code. Thus – like dynamic links – classboxes allow for the introduction of links to objects of unanticipated types without breaking clients of existing types. Classboxes is a language extension, while support for dynamic links is provided by a library.

6 Conclusion

Dynamic links can connect objects of unrelated types. This makes it possible to introduce links from objects of existing types to objects of unanticipated types without imposing any modifications.

Dynamic links promote extension that is compliant with the open/closed principle: First, software components become open towards new dimensions of extension – objects of new types can be freely attached to objects of existing types. Second, software components remain closed to modification – no introduction of fields and methods on existing types is required.

It is possible to implement dynamic links as a library for any mainstream object-oriented programming language. We have presented Decouplink for Java – no extension of the language or runtime is required to use it.

Dynamic links increase the design space for extensible software. We have used dynamic links to design and maintain a component-based system in a domain where dimensions of extension are difficult to predict – a climate control system for greenhouses. We have presented experience gained from this effort.

We believe that dynamic links have the potential to improve extensibility of a wide variety of software systems. We are, therefore, very much interested in experience from other domains. In particular, we would like to learn more about the long-term effects of evolving software using dynamic links. Finally, we would like to explore IDE-support that makes programming with dynamic links easier.

References

1. Aaslyng, J., Lund, J., Ehler, N., Rosenqvist, E.: IntelliGrow: A Greenhouse Component-Based Climate Control System. In: Environmental Modelling & Software, vol. 18(7), pp. 657–666. Elsevier, Amsterdam (2003)
2. Arnold, K., Gosling, J., Holmes, D.: Java Programming Language. Addison-Wesley Professional, Reading (2005)
3. Bennett, K.H., Rajlich, V.T.: Software Maintenance and Evolution: A Roadmap. In: Proceedings of the Conference of the Future of Software Engineering, pp. 73–87 (2000)
4. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: OOPSLA 2005 – ACM Sigplan Notices, vol. 40(10), pp. 177–189 (2005)
5. Boudreau, T., Tulach, J., Wielenga, G.: Rich Client Programming: Plugging into the NetBeans^{TM} Platform. Prentice Hall PTR, Englewood Cliffs (2007)
6. Clifton, C., Leavens, G., Chambers, C., Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: OOPSLA 2000 – Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 130–145 (2000)
7. Fowler, M.: UML Distilled. Addison-Wesley Professional, Reading (2004)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
9. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing, Amsterdam (1983)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
11. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. Wiley, Chichester (2004)
12. Lehman, M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE 68, 1060–1076 (1980)
13. Lehman, M., Ramil, J.: Software Uncertainty. In: Software 2002: Computing in an Imperfect World, pp. 477–514 (2002)
14. Martin, R.C.: The Interface Segregation Principle. C++ Report (1996)
15. Martin, R.C.: The Open-Closed Principle. C++ Report (1996)
16. Meijer, E., Szyperski, C.: Overcoming Independent Extensibility Challenges. Communications of the ACM 45(10), 41–44 (2002)
17. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1988)

18. McGregor, J.: Context. Journal of Object Technology 4(7), 35–44 (2005)
19. Object Management Group: OMG Unified Modeling Language$^{\text{TM}}$ (OMG UML), Infrastructure, http://www.omg.org/spec/UML/2.2/
20. Ostermann, K., Kniesel, G.: Independent Extensibility – An Open Challenge for AspectJ and Hyper/J. In: ECOOP 2000 – Workshop on Aspects and Dimension of Concerns (2000)
21. Rumbaugh, J.: Relations as Semantic Constructs in an Object-Oriented Language. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, pp. 466–481 (1987)
22. Rytter, M., Jørgensen, B.N.: Enhancing NetBeans with Transparent Fault Tolerance. Journal of Object Technology 9(5) (2010)
23. Rytter, M., Jørgensen, B.N.: Composing Objects in Open Contexts using Dynamic Links. In: Informatics – Software Engineering and Applications (2010)
24. Shaw, M., Garlan, D.: Software Architecture – Perspectives on an Emerging Dicipline. Prentice-Hall, Englewood Cliffs (1996)
25. Steimann, F.: The Paradoxical Success of Aspect-Oriented Programming. In: OOPSLA 2006 – Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 481–497 (2006)
26. Szyperski, C.: Independently Extensible Systems – Software Engineering Potential and Challenges. In: Proceedings of the 19th Australasian Computer Science Conference (1996)
27. Szyperski, C.: Component Software – Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Professional, Reading (2002)
28. Tantau, H., Lange, D.: Greenhouse Climate Control: An Approach for Integrated Pest Management. Computers and Electronics in Agriculture 40, 141–152 (2003)
29. Thomas, D., Hunt, A.: Programming Ruby: A Pragmatic Programmer's Guide. Addison-Wesley Professional, Reading (2000)
30. Van Pee, M., Berckmans, D.: Quality of Modelling Plant Responses for Environment Control Purposes. Computers and Electronics in Agriculture 22, 209–210 (1999)
31. van Straten, G., Challa, H., Buwalda, F.: Towards User Accepted Optimal Control of Greenhouse Climate. Computers and Electronics in Agriculture 26, 221–238 (2000)

Software Packaging Approaches —A Comparison Framework

Shouki A. Ebad[*] and Moataz Ahmed

Information and Computer Science Department,
King Fahd University of Petroleum and Minerals,
Dhaharan 31261, Saudi Arabia
{shouki,moataz}@kfupm.edu.sa

Abstract. Effective software modularity brings many benefits such as long-term cost reduction, architecture stability, design flexibility, high maintainability, and high reusability. A module could be a class, a package, a component, or a subsystem. In this paper, we are concerned with the package type of modules. There has been a number of attempts to propose approaches for automatic packaging of classes in OO development. However, there is no framework that could be used to aid practitioners in selecting appropriate approaches suitable for their particular development efforts. In this paper we present an attribute-based framework to classify and compare these approaches and provide such aid to practitioners. The framework is also meant to guide researchers interested in proposing new packaging approaches. The paper discusses a number of representative packaging approaches against the framework. Analysis of the discussion suggests open points for future research.

Keywords: automatic software packaging, software architecture, software modularization, optimization.

1 Introduction

From the object-oriented (OO) software engineering perspective, it is well recognized that good organization of classes into identifiable and collaborating modules improves the understandability, architecture stability, maintainability, testability, and reusability, all leads to more long-term cost-effective development [1]. The software architecture design effort addresses the problem of structuring the software system under development into modules; this is to include issues such as interconnections between modules, assignment of behaviors to modules, and scalability of modules to larger solutions [2]. A module could be a class, a package, a component, or a subsystem. When considering OO development, the design is a collection of related classes. If there are only few classes in the entire application, we may regard the class decomposition as the architecture. However, as the number of classes grows, such a

[*] Corresponding author. Present addresses: Information and Computer Sciences Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, P.O. Box 1594, Saudi Arabia. Tel.: +966 3 860 2464.(Off), + 966 3 860 5940 (Res). fax: +966 3 860 2174.

I. Crnkovic, V. Gruhn, and M. Book (Eds.): ECSA 2011, LNCS 6903, pp. 438–446, 2011.

set of classes can no longer suffice as the architecture because it would be too complex to comprehend at once according to the divide and conquer principle. In this case, the architecture is looked at as packages along with their interconnections; such packages are recursively divided into sub-packages and so on until a level of packages that is easily comprehendible for detailed design is achieved. At the lowest level, a package would be a collection of related classes. Terminology-wise, packages, at the highest level, which can be offered as standalone applications, are referred to as *sub-systems*. Packages, at lower levels, which are functionally cohesive enough, are referred to as *components*. Lowest levels packages are referred to as simply *packages*.

Considering OO development, it is typical for the requirements analysis and system modeling activity to produce a conceptual class model as input to the architecture design activity. In this case, the architect conducts bottom-up packaging of conceptual classes into higher level packages for later detailed design effort. The packaging process are typically guided by some objectives; for instance, it is very common for architects to design for *architecture stability* where the system is structured in a way that permits changes to a system to be made without the architecture having to change [7]. In this case, the effectiveness of the packaging process could be measured by measuring how highly cohesive (i.e., strong intra-package dependencies) and loosely coupled (i.e., weak inter-package dependencies) the resultant packages. Our literature survey, however, revealed that effective bottom-up packaging could be challenging due to two major problems: 1) the lack of effective metrics to be used in guiding the packaging process; and 2) the exponential number of possible packages to be examined for best structure. The first problem is beyond the scope of this paper. In this paper we survey the approaches proposed to address the second problem. The problem has been addressed by many researchers as a problem of NP-hard combinatorial optimization problem to determine the optimal grouping of possibly large, but finite number of classes [1][4][5][8]. There has been a number of attempts to propose packaging approaches. However, to the best of our knowledge, there is no framework that could be used to classify and compare such packaging approaches. In this paper we present a framework to facilitate such classifications and comparisons based on a set of attributes identified as a result of an intensive survey of existing approaches. The framework is meant to aid practitioners in selecting appropriate approaches suitable for their particular development efforts. Moreover, the framework can also guide researchers interested in proposing new packaging approaches. The paper discusses six representative packaging approaches against the framework.

The rest of this paper is organized as follows. Section 2 presents a set of attributes that makes up our comparison framework. A set of representative packaging approaches is discussed in Section 3. Section 4 summarizes our observations on the analysis of the approaches against the framework. Section 5 concludes the paper and offers some directions for future work.

2 Comparison Framework

Judging a packaging approach should not only be based on its effectiveness and efficiency, but also on the underlying characteristics that affect its effectiveness and efficiency. Throughout surveying existing related work on packaging approaches, we identified some attributes that can be used for classifying and comparing different packaging approaches. We expect this set of attributes to help in enhancing existing packaging approaches as well as guiding researchers trying to develop new packaging approaches. Our proposed attributes are discussed in the sequel.

Packaging Goal: In all related work, the goals of packaging come before anything else described. This is important to be able to evaluate packaging achievement. The packaging goal affects the way the packaging process is conducted and the kind of measures to be used to assess the process. Getting on more understandable system and reducing the maintainability effort are examples of the packaging goal.

Underlying Principle: To better understand a concept, it is important to understand the underlying principle upon which the work is built.

Input Artifact: This attribute determines the different inputs required by the packaging approach. Source code and class diagram are examples of the input artifact.

Internal Quality Attribute: This attribute reflects the internal design attribute that guides the packaging process. Cohesion, complexity, length, coupling, and size are examples of such internal attributes.

Search Algorithm: Because packaging could be treated as a search problem, most packaging approaches use heuristic search methods. This attribute indicates the used search algorithm (heuristic or exact) and lists, in case of heuristic, the main features of the used algorithm such as representation and parameter selection.

Fitness Function: In most of the heuristic search techniques, the packaging objective is converted into an objective function which is furthermore converted into a fitness function that is to be optimized to find a solution for the problem. Fitness functions on software packaging are nothing more than software design metrics.

Scalability: According to [4], the software system is small if the number of classes is less than 15 classes. In case of small-sized software systems, the packaging process becomes a simple problem because we can find a polynomial- time algorithm to solve it. The scalability attribute measures the capability of a packaging approach to scale up or scale out in terms of the software size. In general, rough categorization can be used: small (between 15 and 25 classes) or large (more than 25 classes).

Soundness: This attribute reflects whether the packaging approach was shown to group the classes in the way the approach claims it would.

Practicality: This attribute reflects aspects of practicality of the packaging approach. This includes amount of resources (e.g. time, memory) that the packaging approach consumes in generating the right packaging.

Supportability: To know whether the packaging approach is supported by some kind of automation and integrated with CASE tools.

3 Packaging Approaches

We present a summary discussion of six representative packaging approaches based on our set of attributes. The list of considered approaches in our study is not exhaustive, but we gave attention to those works we considered significant and more recent as regards the subject under discussion. We also discuss the shortcomings associated with the considered approaches.

3.1 Doval et al 1999 [5]

They defined the Modularization Quality (MQ) of a system as an objective function to quantify the quality of a given module dependency graph MDG They used MQ as the objective function of their Genetic Algorithm to express the trade-off between intra- and inter-connectivity attributes. MQ achieved this trade-off by subtracting the average inter-connectivity from the average intra-connectivity. Table 1 discusses this approach based on our attributes.

Table 1. Approach of Doval et. al. 1999

Attribute	Comments
Goal	To simplifying the system structure
Principle	Module Dependency Graph (MDG)
Input	Source code
Internal Att.	Intra connectivity and inter connectivity
Search Alg.	GA: numeric encoding, crossover rate = 0.8, mutation rate = 0.004, roulette wheel selection, population size = 10n where n is the number of nodes in the MDG
Fitness	Tradeoff between inter- and intra-connectivity. This trade-off is achieved by subtracting the average inter-connectivity from the average intra-connectivity. It is bound between -1 and 1
Scalable	Scale out; small-sized software system (Mini-Tunis) with 20 modules
Sound	It suffers from the module misplacement problem That makes the solution sub-optimal
Practical	No mention
Support.	The Bunch tool [9] that automatically creates a system decomposition

3.2 Liu et al 2001 [8]

A challenging problem in Email environments is optimally allocating users to servers. This paper presented a method for decomposing a large number of objects (users) into mutually exclusive groups (servers) where within-group dependencies are high and between-group dependencies are low. Their ultimate goal was to minimize network traffic. Based on our attributes, Table 2 shows our discussion of the approach.

Table 2. Approach of Liu et. al. 2001

Attribute	Comments
Goal	To find arrangement of objects on some groups to minimize network traffic
Principle	Graph represented by frequency matrix
Input	Simulated data that represents the messages being sent amongst objects
Internal Att.	Within-group dependencies and between-group dependencies
Search Alg.	Group GA: numeric encoding, random selection, fixed and variable the crossover rate based on the number of function calls
Fitness	The fitness function reflects the work objective by rewarding groups where there is a lot of communication between members
Scalable	Scale up; large-sized system; 250 objects (users) and 5 groups (servers).
Sound	Evolutionary Algorithm is the best compared to clustering (PAM and PAM-M), and Hill Climbing. When a small modification is made to the crossover rate, HC catches EA up.
Practical.	For n objects, n(n+1)/2 elements need to be stored
Support.	Not supported

3.3 Chiricota et al 2003 [3]

They presented a method for finding relatively good clustering of software systems. Their method exploits a metric based clustering of graphs. To evaluate the resultant structure, they used the MQ metric (Approach of Doval et. al. 1999). Table 3 presents a summary discussion of this approach according to our attributes.

Table 3. Approach of Chiricota et. al. 2003

Attribute	Comments
Goal	To achieve this principle “cohesive subsystems with loosely interconnected“
Principle	Small-world graph (undirected)
Input	Source code
Internal Att.	Coupling
Search Alg.	They used the min-cut algorithm consisting in finding a clustering made of several distinct subsets or blocks $c_1, \dots,$ cp such that the number of edges connecting nodes of distinct blocks is kept to a minimum
Fitness	Edges between subsystems are weak edges if their value falls below a given threshold. Once those edges have been deleted, the connected components of the induced graph correspond exactly to the required cluster structure
Scalable	Scale up; three large-sized systems: ResynAssistant, MacOS9, and MFC
Sound	A large number of nodes at the three applications is left isolated
Practical	Short computing time
Support.	Not supported

3.4 Bauer and Trifu 2004 [2]

Because the recovered software architecture is not always meaningful to a human software engineer, this paper proposed an approach that combines clustering with pattern-matching techniques to recover meaningful decompositions. Table 4 discusses this approach based on our attributes.

Table 4. Approach of Bauer and Trifu 2004

Attribute	Comments
Goal	To bring recovered subsystem decompositions closer to what an expert would produce manually
Principle	Un-clustered graph
Input	Source code
Internal Att.	Accuracy (meaningful) and optimality (cohesion and coupling)
Search Alg.	A two-pass MMST (modified minimum spanning tree)
Fitness	It is based on two criteria: accuracy (primary) and optimality (secondary). Decomposition is accurate if it is “meaningful” to a software engineer. This includes: 1) the subsystems contain only semantically related components 2) all semantically related components should be in a single subsystem. Decomposition is optimal if the subsystem: high cohesion and low coupling..
Scalable	Scale up; a large-sized system (Java AWT) with 482 classes
Sound	In terms of optimality, it does not significantly improve the decomposition
Practical	Some phases in this approach are time and memory consuming
Support.	It is not supported

3.5 Seng et al 2005 [10]

They expressed the task of improving a subsystem decomposition as a search problem. Software metrics and design heuristics are combined into a fitness function which is used to measure the quality of subsystem decompositions. Table 5 summarizes this approach based on our attributes.

Table 5. Approach of Seng et. al. 2005

Attribute	Comments
Goal	To determine a decomposition with fewer violations of design principles
Principle	Directed graph
Input	Source code
Internal Att.	cohesion, coupling, complexity, cycles, and bottleneck
Search Alg.	Group GA: an adapted crossover, mutation are split & join, elimination, and adoption, tournament selection, the initial decomposition is the existing one
Fitness	A multi modal function (cohesion, coupling, complexity, cycles, bottleneck).
Scalable	Scale up; a large-sized system (JHotDraw) with 207 classes
Sound	It does not improve the cohesion and coupling
Practical	The execution is fast because of using the efficient tournament selection
Support.	Not supported

3.6 Abdeen et al 2009 [1]

They addressed the problem of optimizing existing modularizations by reducing the inter-package connectivity; this reduction is inspired from well-known package cohesion and coupling principles. Compared to the other approaches, this approach allows maintainers to define some constraints such as package size and the limit of modifications on the original modularization Discussion of the approach based on our attributes is described in Table 6.

Table 6. Approach of Abdeen et. al. al. 2009

Attribute	Comments
Goal	To optimize the decomposition of system into packages so that the resulting organization reduces connectivity between packages
Principle	Directed graph
Input	Existing modularization
Internal Att.	Inter-package connections/cyclic dependencies
Search Alg.	Simulated Annealing, CoolingSchd.(T)=0.9975×T, p>e^(-Tcurrent/Tstart)
Fitness	The average of dependency quality (which relies on CCP and ADP package principles) and connection quality (which relies on CRP and ACP package principles)
Scalable	Scale up; four large-sized systems JEdit , ArgoUML, Jboss, and Azureus with 802, 1671, 3094, and 4212 classes respectively
Sound	It results packages having no classes i.e., empty packages. The percentage of empty packages exceeds 25% of some application classes. In addition, this approach does not allow to remove empty packages from the system
Practical	No mention
Support.	Not supported

4 Observations

Based on the above analysis of each packaging approach, the primary observations of this study can be summarized as follows:

- Most packaging approaches deal with packaging as an optimization problem.
- Most packaging approaches consider maximizing intra-package cohesion and minimizing inter-package coupling as the optimization objective function.
- Most packaging approaches use Genetic Algorithms as a heuristic search method.
- All packaging approaches are graph based.
- Most packaging approaches use the source code as their input artifact. This reveals that there is a lack of approaches that would help in packaging early during the architectural design phase; this is the time when packaging might be needed the most for better architectural and component design. This remains a research area in need of more efforts where only conceptual models are available.
- Selection of parameters of the heuristic search method such as GA or SA is very crucial. Different parameters settings may result in completely different structures; due to being trapped in local optima as in the approach of Liu et. al.
- Most packaging approaches are scalable.
- The soundness of discussed packaging approaches seems to be questionable. For examples, Doval et al may result in module misplacement, Chiricota et al 2003 may result in isolated nodes, Bauer and Trifu may result in no significant improvement, and Abdeen et al. may result in empty packages.
- Except Doval et al, all packaging approaches are not supported by CASE tools.

Table 7 summarizes the packaging approaches surveyed based on our attributes.

Table 7. Summary of the existing packaging approaches based on our attributes

Study	Goal	Input	Principle	Internal Att.
Doval et. al. 1999	To simplify the system structure	source code	Graph	Intra connectivity and inter connectivity
Liu et. al. 2001	To minimize network traffic	simulated data	Graph	Within-group dependencies and between-group dependencies
Chiricota et. al. 2003	To achieve a design principle (Cohesion & Coupling)	source code	Graph	Coupling
Bauer & Trifu 2004	To produce meaningful decomposition	source code	Graph	Accuracy (meaningful) & optimality (cohesion & coupling
Seng et. al. 2005	To decompose the system with fewer violations of design principles	source code	Graph	Cohesion, coupling, complexity, cycles, and bottleneck
Abdeen et. al. al. 2009	To achieve a design principle (Cohesion & Coupling)	existing modularization	Graph	Inter-package connections/cyclic dependencies

Table 7. (*continued*)

Search Alg.	Fitness	Scalable	Sound	Practical	Support
GA	Tradeoff between inter- and intra-connectivity	No	No	N/A	Yes
GA	Penalizing groups where there is no communication	Yes	No	No	No
Min-cut	Deleting the weak edges	Yes	No	Yes	No
MMST	High internal cohesion & low external coupling	Yes	No	No	No
GA	A multi modal fitness function	Yes	No	Yes	No
SA	Averaging of some quality metrics	Yes	No	N/A	No

5 Conclusion and Future Work

In this paper we presented an attribute-based framework to allow classifying and comparing approaches for packaging classes during software development. The paper also provides an analysis of a representative set of packaging approaches in light of the framework. The results of the study provides practitioners with an overview of prominent work in the literature and offer help with regard to making decisions as which approach would be appropriate for their particular development efforts. Moreover, this analysis is meant to serve as guide for researchers interested in developing new packaging approaches. The analysis indentifies some open issues for future work.

Clearly, package-level metrics play crucial role in guiding the packaging process. Unfortunately, package level metrics did not get that much attention of researches as class level metrics did [6]. As a follow-up to the work presented in this paper, the authors are currently working on analyzing the metrics used in the packaging approaches. The authors are working on developing package-level metrics that can guide packaging process towards the development of highly stable architectures.

References

1. Abdeen, H., Ducasse, S., Sahraouiy, H., Alloui, I.: Automatic Package Coupling and Cycle Minimization. In: WCRE 2009, pp. 103–122. IEEE CNF, Los Alamitos (2009)
2. Bauer, M., Trifu, M.: Architecture-aware adaptive clustering of OO systems. In: CSMR 2004, pp. 3–14. IEEE Computer Society, Los Alamitos (2004)
3. Chiricota, Y., Jourdan, F., Melancon, G.: Software components capture using graph clustering. In: The 11th IEEE Int'l Workshop on Program Comprehension, IWPC 2003 (2003)
4. Clarke, J., Dolado, J., Harman, M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., Shepperd, M.: Reformulating software engineering as a search problem. IEEE Proceedings on Software 150(3), 161–175 (2003)
5. Doval, D., Mancoridis, S., Mitchell, B.: Automatic clustering of software systems using a genetic algorithm. In: STEP 1999, IEEE Computer Society, Los Alamitos (1999)
6. Genero, M., Piattini, M., Calero, C.: A Survey of Metrics for UML Class Diagrams. Journal of Object Technology JOT 4(9) (November/December 2005)

7. Lethbridge, T., Laganière, R.: Object-Oriented Software Engineering: Practical Software Development using UML and Java. McGraw-Hill, New York (2005)
8. Liu, X., Swift, S., Tucker, A.: Using evolutionary algorithms to tackle large scale grouping problems. In: GECCO 2001 (2001)
9. Mancoridis, S., Mitchell, B., Chen, Y., Gansner, E.: Bunch: A clustering tool for the recovery and maintenance of software system structures. In: ICSM 1999, IEEE Computer Society Press, Los Alamitos (1999)
10. Seng, O., Bauer, M., Biehl, M., Pache, G.: Search-based Improvement of Subsystem Decompositions. In: Proc. of the GECCO 2005 (2005)

Author Index