Lecture Notes in Computer Science 4758

Flavio Oquendo (Ed.)

Software Architecture

First European Conference, ECSA 2007
Madrid, Spain, September 24-26, 2007
Proceedings

Springer

Volume Editor

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

Library of Congress Control Number: 2007935168

CR Subject Classification (1998): D.2.11, D.3, H.2.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-75131-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75131-1 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12161744 06/3180 5 4 3 2 1 0

Preface

Software architecture has emerged as an important subdiscipline of software engineering encompassing a broad set of languages, styles, models, tools, and processes. The role of software architecture in the engineering of software-intensive applications has become more and more important and widespread. Indeed, component-based and service-oriented architectures have become key to the design, development, and evolution of most software systems.

The European Conference on Software Architecture (ECSA) is the premier European conference dedicated to the field of software architecture, covering all architectural features of software and service engineering. It is the follow-up of a successful series of European workshops on software architecture held in the United Kingdom in 2004 (Springer LNCS 3047), Italy in 2005 (Springer LNCS 3527), and France in 2006 (Springer LNCS 4344). Due to its success, it has evolved into a full-fledged series of European conferences whose first edition was ECSA 2007, held in Madrid, Spain September 24–26, 2007.

ECSA 2007 provided an international forum for researchers and practitioners from academia and industry to present innovative research and to discuss a wide range of topics in the area of software architecture. It focused on formalisms, technologies, and processes for describing, verifying, validating, transforming, building, and evolving software systems, in particular founded on component-based and service-oriented architectures. Covered topics included architecture modeling, architectural aspects, architecture analysis, transformation and synthesis, quality attributes, model-driven engineering, and architecture-based support for developing, adapting, reconfiguring, and evolving component-based and service-oriented systems.

The conference attracted paper submissions from 26 countries (Algeria, Argentina, Australia, Belgium, Brazil, Canada, Chile, Denmark, Egypt, Finland, France, Germany, Ireland, Italy, Japan, Korea, Mexico, New Zealand, Poland, Portugal, Spain, The Netherlands, Tunisia, UK, Uruguay, and USA). In all, 89 abstracts were submitted of which 62 papers were actually uploaded on the conference submission Web site.

All submissions were reviewed by three members of the Program Committee. Papers were selected based on originality, quality, soundness, and relevance to the conference.

The Program Committee selected papers according to four types, for presentation in paper sessions:

- Full research papers: they describe novel contributions to software architecture research and cover work that has a sound scientific/technological basis and has been fully validated.
- Emerging research papers: they present promising results from work-in-progress in a topic of software architecture research and cover work that has a sound basis, but has not yet been validated in full.

- Experience papers: they describe significant experiences related to software architecture practice and present case studies or real-life experiences of benefit to practitioners and researchers.
- Research challenge papers: they present significant research challenges in theory or practice of software architecture or the state of the art on different topics related to software architecture.

These papers are published in the conference proceedings as long papers (16 pages), short papers (8 pages), or position papers (4 pages). Full research papers always appear as long papers; research challenge papers typically as position papers; the others as long or short papers.

In addition, papers accepted as posters to be presented in a poster session are published as extended abstracts (4 pages).

The Program Committee selected 18 papers out of 62 submissions: 12 long papers (5 as full research papers, 1 as full state-of-the-art paper, 6 as emerging research papers), 4 short papers (3 as emerging research papers, 1 as experience paper), and 2 position papers (research challenge papers). This gives an acceptance rate of 10% for full papers, +10% for emerging research (long) papers, +10% for short and position papers. In addition, 16 papers were selected to be presented as posters, i.e., +25%.

Credit for the quality of the proceedings goes to all authors of papers. In addition, the conference included three keynote talks. The opening day keynote was presented by David Garlan, from Carnegie Mellon University, USA, on "Software Architectures for Task-Oriented Computing"; the second keynote was delivered by Ron Morrison, from the University of St. Andrews, UK, on "An Active Architecture Approach to Dynamic Systems Co-evolution"; the last keynote speaker was Michael P. Papazoglou, from the University of Tilburg, Netherlands, on "What's in a Service?".

We would like to thank the members of the Program Committee for providing thoughtful and knowledgeable reviews and for their substantial effort in making ECSA 2007 a successful conference.

The ECSA 2007 submission and review process was extensively supported by the Paperdyne Conference Management System. We are indebted to Thomas Richter for his timely support.

On the organizational front, we deeply acknowledge the Conference Secretary, Verónica Andrea Bollati, for her great work beyond the call of duty, and all the members of the Organizing Committee for their excellent service.

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

September 2007

Flavio Oquendo
Carlos E. Cuesta
Esperanza Marcos

Organization

Conference Co-chairs

Carlos E. Cuesta
Rey Juan Carlos University, Spain
carlos.cuesta@urjc.es

Esperanza Marcos
Rey Juan Carlos University, Spain
esperanza.marcos@urjc.es

Program Committee

Program Committee Chair

Flavio Oquendo
University of South Brittany – VALORIA, France
flavio.oquendo@univ-ubs.fr

Program Committee Members

Yamine Ait Ameur
ENSMA, France

Dharini Balasubramaniam
University of St. Andrews, UK

Thais Batista
University of Rio Grande do Norte – UFRN, Brazil

Marco Bernardo
University of Urbino, Italy

Antoine Beugnard
ENST Bretagne, France

Jan Bosch
Nokia Research Center, Finland

Pere Botella
Polytechnic University of Catalonia, Spain

Francisco Curbera
IBM Research Center, USA

Rogerio de Lemos
University of Kent, UK

Laurence Duchien
INRIA and University of Lille, France

José Luiz Fiadeiro
University of Leicester, UK

Régis Fleurquin
University of South Brittany – VALORIA, France

David Garlan
Carnegie Mellon University, USA

Carlo Ghezzi
Polytechnic of Milan, Italy

Ian Gorton
Pacific Northwest National Lab, USA

Mark Greenwood
University of Manchester, UK

Paul Grefen
Eindhoven University of Technology, The Netherlands

Volker Gruhn
University of Leipzig, Germany

Wilhelm Hasselbring
University of Oldenburg, Germany

Juan Hernández
University of Extremadura, Spain

Valérie Issarny
INRIA Rocquencourt, France

Natalia Juristo
Polytechnic University of Madrid, Spain

René Krikhaar
ICT NoviQ and Vrije Universiteit Amsterdam, The Netherlands

Philippe Kruchten
University of British Columbia, Canada

Frédéric Lang
INRIA Rhône-Alpes, France

Nicole Levy
University of Versailles St-Quentin en Yvelines – PRiSM, France

Antonia Lopes
University of Lisbon, Portugal

Jeff Magee
Imperial College London, UK

Carlo Montangero
University of Pisa, Italy

Ron Morrison
University of St. Andrews, UK

Robert L. Nord
Software Engineering Institute, USA

Henk Obbink
Philips Research Europe, The Netherlands

Mourad Oussalah
University of Nantes – LINA, France

Mike P. Papazoglou
Tilburg University, The Netherlands

Jennifer Pérez
Polytechnic University of Madrid, Spain

Dewayne E. Perry
University of Texas at Austin, USA

Mario Piattini
University of Castilla-La Mancha, Spain

Frantisek Plasil
Charles University, Czech Republic

Eltjo Poort
LogicaCMG, The Netherlands

Amar Ramdame-Cherif
University of Versailles St-Quentin en Yvelines – PRiSM, France

Isidro Ramos
Polytechnic University of Valencia, Spain

Ralf Reussner
University of Karlsruhe, Germany

Clemens Schäfer
University of Leipzig, Germany

Bradley Schmerl
Carnegie Mellon University, USA

Clemens Szyperski
Microsoft Research, USA

Richard N. Taylor
University of California at Irvine, USA

Miguel Toro
University of Sevilla, Spain

Brian Warboys
University of Manchester, UK

Eoin Woods
UBS Investment Bank, UK

Additional Reviewers

Edoardo Bontà
University of Urbino, Italy

Fabien Dagnat
ENST Bretagne, France

Anna Grimán Padua
Polytechnic University of Madrid, Spain

Marta Lopez Fernandez
Polytechnic University of Madrid, Spain

Radu Mateescu
INRIA Rhône-Alpes, France

Luca Padovani
University of Urbino, Italy

Olivier Ponsini
INRIA Rhône-Alpes, France

Maria-Teresa Segarra
ENST Bretagne, France

Wendelin Serwe
INRIA Rhône-Alpes, France

Organizing Committee

Conference Secretary

Verónica Andrea Bollati
Rey Juan Carlos University, Spain

Financial Chair

Belén Vela
Rey Juan Carlos University, Spain

Local Arrangements Chair

César J. Acuña
Rey Juan Carlos University, Spain

Publicity and Liaisons Chair

Javier Garzás
Kybele Consulting and Rey Juan Carlos University, Spain

Posters Chair

Jennifer Pérez
Polytechnic University of Madrid, Spain

Registration Chair

Valeria de Castro
Rey Juan Carlos University, Spain

Webmasters

Diana Marcela Sánchez
Rey Juan Carlos University, Spain

Juan Manuel Vara
Rey Juan Carlos University, Spain

Organizing Members

Nour Ali
Polytechnic University of Valencia, Spain

Paloma Cáceres
Rey Juan Carlos University, Spain

José María Cavero
Rey Juan Carlos University, Spain

Marcos López
Rey Juan Carlos University, Spain

Steering Committee

Flavio Oquendo
University of South Brittany – VALORIA, France

Carlos E. Cuesta and Esperanza Marcos
Rey Juan Carlos University, Spain

John Favaro
Consorzio Pisa Ricerche, Italy

Volker Gruhn
University of Leipzig, Germany

Ron Morrison
University of St. Andrews, UK

Mourad Oussalah
University of Nantes – LINA, France

Brian Warboys
University of Manchester, UK

Sponsorship

ECSA 2007 was sponsored by several public institutions under competitive grants and by private companies and organizations. They are listed in several categories, reflecting the extent of their support and funding.

Platinum Sponsorship

The Spanish Ministry of Education and Science (MEC)
The Managing Director’s Office for Universities and Research (DGUI) from the Autonomous Government of Madrid (CM)
Rey Juan Carlos University (URJC)

Gold Sponsorship

Kotasoft Soluciones Informáticas, Ltd.
Kybele Consulting, Ltd.
Open Canarias, Ltd.

Silver Sponsorship

InterSystems Corporation

Moreover, the conference had the collaboration of the Spanish Office of the World-Wide Web Consortium (W3C).

Table of Contents

Keynotes

Full Research Papers

Emerging Research Papers

Experience Papers

Research Challenge Papers

Poster Papers

Software Architectures for Task-Oriented Computing

David Garlan

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213 USA
garlan@cs.cmu.edu

Abstract. Recent advances in ubiquitous computing and artificial intelligence have led to a desire to better support user-oriented tasks by placing more intelligence in the computing infrastructure. This infrastructure helps mediate between computing resources and legacy applications on the one hand, and a user's high-level goals on the other hand. In this talk I survey recent research in developing software architectures to support these new capabilities. Key features of these architectures are the ability to interface with legacy applications, but still add considerable support for user tasks; the ability to incorporate machine learning so that the system adapts to the user over time; and the ability to cope with resource variability and user mobility. I outline some of the consequent software engineering challenges that arise in this setting.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, p. 1, 2007.

An Active Architecture Approach to Dynamic Systems Co-evolution

Ron Morrison[1], Dharini Balasubramaniam[1], Flavio Oquendo[2], Brian Warboys[3], and R. Mark Greenwood[3]

[1] University of St Andrews
St Andrews, KY16 9SX, UK
{ron,dharini}@cs.st-andrews.ac.uk
[2] University of South Brittany – Valoria
BP 573, 56017 Vannes Cedex, France
Flavio.Oquendo@univ-ubs.fr
[3] University of Manchester, Manchester, M13 9PL, UK
{brian,markg}@cs.man.ac.uk

Abstract. The term *co-evolution* describes the symbiotic relationship between dynamically changing business environments and the software that supports them. Business changes create pressures on the software to evolve, and at the same time technology changes create pressures on the business to evolve. More generally, we are concerned with systems where it is neither economically nor technologically feasible to suspend the operation of the system while it is being evolved. Typically these are long-lived systems in which *dynamic* co-evolution, whereby a system evolves as part of its own execution in reaction to both predicted and emergent events, is the only feasible option for change. Examples of such systems include continuously running business process models, sensor nets, grid applications, self-adapting/tuning systems, routing systems, control systems, autonomic systems, and pervasive computing applications.

Active architectures address both the structural and behavioural requirements of dynamic co-evolving software by modelling software architecture as part of the on-going computation, thereby allowing evolution during execution and formal checking that desired system properties are preserved through evolution. This invited paper presents results on active architectures from the Compliant System Architecture and ArchWare projects. We have designed and constructed the ArchWare-ADL, a formal, well-founded architecture description language, based on the higher-order typed π-calculus, which consists of a set of layers to address the requirements of active architectures. The ArchWare-ADL design principles, concepts and formal notations are presented together with its sophisticated reflective technologies for supporting active architectures and thereby dynamic co-evolution.

1 Introduction

Where software cannot change to keep up with the changing business goals, the business loses efficiency, and the perceived value of the software decreases [17].

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 2 – 10, 2007.

Conversely, a business must continually intercept the potential of new technology to maintain its effectiveness and remain competitive. Thus business changes create pressures on the software to evolve, and at the same time software changes create pressures on the business to evolve [32]. We use the term co-evolution to describe the concept of the business and the software evolving sympathetically, but at potentially different rates, to satisfy changing requirements.

More generally, we are concerned with what Milner termed "wide informatic systems" [21] to describe systems, assembled from components, that co-evolve with their environment. Such systems recognise that the business effects of introducing, or changing, a software system are potentially emergent, especially in terms of non-functional requirements such as safety, reliability and performance. Typically these are long-lived applications in which dynamic co-evolution, whereby a system evolves as part of its own execution, is the only feasible option for change. Examples of such systems include continuously running business process models, sensor nets, grid applications, self-adapting/tuning systems, routing systems, control systems, and pervasive computing applications.

Active architectures address the structural and behavioural requirements of dynamic co-evolving software by modelling the software architecture as part of the on-going computation, thereby allowing evolution during execution. They facilitate the design and engineering of dynamic co-evolving systems in a way that preserves the flexibility to adapt to new requirements while ensuring that important properties can be continuously verified. Fundamentally, users and developers must be able to understand a system, and its interactions with the environment, to define the appropriate evolutions. In our approach active architectures manage the co-evolutionary process using: a formal description of the system's structure and behaviour; self-monitoring capabilities to provide a view of the system's dynamic behaviour; sophisticated evolutionary mechanisms including a model of the system itself at the appropriate level of abstraction and incremental checking tools for formally verifying that desired system properties are preserved over evolution.

We have designed and constructed the ArchWare-ADL [26][27], a formal, well-founded architecture description language based on the higher-order typed π-calculus [19], which consists of a set of layers to address the requirements of active architectures. The ArchWare-ADL design principles, concepts and formal notations are presented along with sophisticated reflective technologies for supporting active architectures and thereby dynamic co-evolution.

2 Active Software Architecture

A key aspect of the design of a software system is its architecture; the fundamental organisation of the system embodied in its components, their relationships to one another and to the environment. Two viewpoints are frequently used in describing software architectures: *structural* and *behavioural* [13].

The structural viewpoint may be specified in terms of:

- components – units of computation of a system,
- connectors – interconnections among components for supporting their interactions, and
- configurations – of components and connectors.

From a structural viewpoint, an architecture description should provide a formal specification of the architecture in terms of components and connectors, how they are composed, and how the architectural structure may change.

The behavioural viewpoint may be specified in terms of:

- actions a system executes or participates in,
- relations among actions to specify behaviours,
- behaviours of components and connectors, and how they interact and change,
- the state of the active system.

We define an active architecture as dynamic in that it can evolve during execution by creating, deleting, reconfiguring and moving components, connectors and their configurations at runtime. Importantly, active architectures require the structural and behavioural viewpoints to specify static and dynamic behoviour and properties. A challenge for an Architecture Description Language (ADL) is to support active architectures by finding the abstractions and technologies that accommodate the dynamic evolution of components, connectors and configurations through their actions, relationships and behaviours.

3 The Archware ADL

The ArchWare-ADL [26] is a novel architecture description language (ADL) designed to accommodate active architectures. It is:

- a formal, theoretically well-founded language based on the higher-order typed π-calculus;
- automated by tools, i.e. a specification and verification toolset providing support for automated checking; and
- supported by a set of reflective technologies for dynamic co-evolution.

The ArchWare-ADL takes its roots in previous work on the use of π-calculus as semantic foundation for architecture description languages [4][5]. When augmented with domain-specific architectural extensions it blends Turing completeness and high architecture expressiveness with a simple formal notation.

The novelty of ArchWare lies in its holistic view of formal development, encompassing languages for formal specification, (i.e. ArchWare ADL), formal verification [18], and formal transformation [28]. As such it goes beyond existing formal methods in supporting architecture-centric model-driven engineering of software systems.

Like formal methods such as B [1], Z [6], VDM [7], and FOCUS [31], ArchWare provides full support for formal description and development of software systems. Unlike these formal methods, ArchWare is based on a process algebra, the higher-order typed π-calculus, and provides architectural support for formal architecture-centric software engineering.

The ArchWare-ADL consists of a set of layers to address the structural and behavioural requirements of active architectures. The novelty is that it models the architecture as part of the on-going computation, thereby allowing the evolution of the architecture in step with execution.

There are five layers in the ArchWare-ADL:

- The base layer defines a coordination language without data values correspondding to the monadic π-calculus.
- The first order layer adds data values and abstractions and corresponds to polyadic π-calculus.
- The higher order layer corresponds to higher-order polyadic π-calculus.
- The analysis layer enables the specification of constraints on the styles.
- The style layer allows the specification of components and connectors.

The style layer provides constructs from which the base component-and-connector style and other derived styles can be defined. Conceptually, an architectural style provides:

- a set of abstractions for architectural elements;
- a set of constraints (i.e. properties that must be satisfied) on architectural elements, including legal compositions; and,
- a set of additional analyses that can be performed on architecture descriptions constructed in the style.

Thus the ArchWare-ADL provides a framework for formalising software architecttures based on the concepts of components and connectors. Styles are used to define families of architectures that have a common structure and satisfy the same properties.

The style layer makes use of both the underlying π-calculus layers and the analysis layer. Styles in ArchWare are defined as property-guarded abstractions that may be applied to yield instances of an architecture conforming to the style. The structure and behaviour of an architecture family are specified using the ArchWare-ADL [11] with the constraints on the family being specified using the analysis layer.

The essential property of dynamic co-evolutionary systems is the ability to (partially) stop and decompose a running system into its constituent components, and compose evolved or new components to form a new system, while preserving any state or shared data if desirable. Once a co-evolving system receives an internal or external stimulus for evolution it can: (partially) suspend the component(s) to be evolved; decompose them into constituent parts; modify the components separately; recompile the components and bind them into the executing system. The support technologies for this are a decomposition operator, linguistic reflection and a system representation capable of capturing closure, called hyper-code [14].

4 Dynamic Co-evolution

A dynamic co-evolving system is constantly in a state of flux as it evolves in reaction to external and internal stimuli. To understand and manage the co-evolutionary process, we introduce the concepts of incarnation, evolutionary step and locus [24].

A dynamic co-evolving system mutates from one incarnation to the next via an evolutionary step. Within a context we can observe a sub-system through a sequence of incarnations. Each incarnation includes a period of normal execution, followed by an evolutionary step. The end of the evolutionary step defines the start of the next incarnation. Thus an incarnation can be considered as system execution between the end points of two subsequent evolutionary steps and consists of the code, data and meta-data not only for carrying out its present purpose, including architectural constraint checking, but also for a variety of possible future evolutionary steps. The change context, i.e. the set of entities that will change during an evolutionary step, is known as the locus. An evolutionary step is defined as a change to a locus and is made by an evolver internal to the locus. Thus it is the locus that evolves itself, in reaction to some external or internal stimulus. During normal execution, the locus continually monitors the incarnation to determine when an evolutionary step is necessary.

A co-evolving application may be constructed in terms of loci. The internal evolver produces new incarnations of the locus. The structuring of loci within an application is static where it is possible to predict the sets of loci and interactions that are subject to change. In such cases, an evolver produces a new incarnation of its locus and the application changes without changing the structure of the loci. Where it is not possible to predict the sets of loci and interactions that are subject to change, a new locus may be defined dynamically to undertake the desired evolutionary step. Thus the structuring of the loci is dynamic. In both cases the locus determines those factors that change and those that remain constant during the evolutionary step. The objective is to make explicit the distinction between what can change and what remains unaffected by each evolutionary step, even though this may not be completely statically defined.

We offer this initial set of intrinsic requirements for dynamic co-evolution [24]:

- a method for structuring dynamic co-evolving systems as loci;
- a technique for programming with incremental design;
- a method of monitoring the internal and external environment;
- a method of partially stopping and decomposing loci and their interactions;
- a method of dynamically reifying the state of the computation;
- a method of generating new loci using the reified state (IR6);
- and, a method of rebinding the new loci into the executing system.

5 Co-evolving Active Architecture

We have developed the following technologies that satisfy the needs of the intrinsic requirements of adaptation [3][23][33][34]

- **Decomposition:** a decompose operator to suspend execution and break up a locus into its constituent parts.
- **Reification:** a reify operator to yield a dynamic representation of the executing locus.
- **Generation:** a method of transforming the representation to create new representations.
- **Reflection:** a reflect operator to compile the resultant representations and bind them into the running computation [30].
- **Probes:** a software mechanism to provide feed forward and feedback stimuli [2].

- **A persistent** environment
- **Dynamic** property checking

Figure 1 illustrates the use of dynamic property checking to ensure that the execution and evolution of the active architecture preserves the essential properties of the system. The architectural style is expressed in terms on elements and constraints among these elements. These map on to the active architecture. The property checker monitors the execution and dynamically checks properties. When a violation occurs this is fed back to the executing architecture that then may form loci (ellipses) to evolve the architecture.

Importantly, since the property checker is part of the active system, it may itself evolve to check new properties or use new technology.

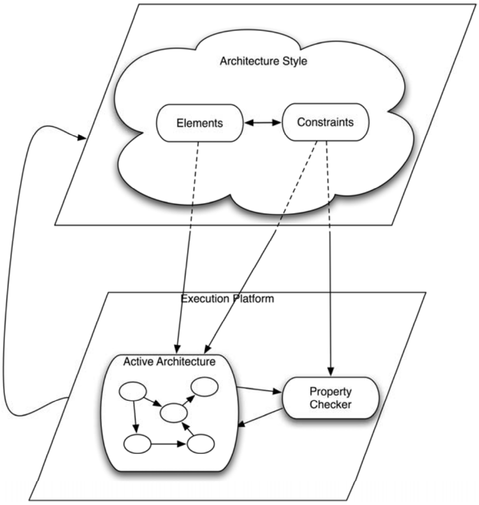


Fig. 1. A Dynamically Co-evolving System

6 Decomposition and Hyper-code

The two most unusual technologies supported by the Archware ADL are decomposition and hyper-code.

Decomposition suspends execution and breaks up a locus into its constituent parts. Unusually the semantics of decomposition are well suited for definition in the π-calculus by allowing each constituent part to run to its reduction limit, that is where it is waiting for an interaction, and then breaking the communication channel and stopping the execution.

Hyper-code is a representation of an active executing graph linking source code and existing values. It allows state and shared data to be preserved during evolution. Thus at any point during the computation the state of the execution may be inspected by viewing the hyper-code.

The importance of hyper-code is that it is rich enough to represent executing code since it captures the state of the computation. Since hyper-code can represent closure, it may be used to introspect an executing system, and thereby can be used to return the result of a decomposition operation. Together the facilities of hyper-code, decomposition, reflection and reification permit users to stop part of an executing system (while the rest of the system continues to execute), inspect its specification and state, evolve the part as necessary, and recompose the system. The generation may be performed by specifying a series of semantics preserving transformations using a programmable interface to the hyper-code graph such as supplied by the Metaprogramming Framework [19].

7 Conclusions

There are many complementary architectural approaches to system evolution [8][9] [10] [12][15][16][25][29][35]. Active architectures address the structural and behavioural requirements of dynamic co-evolving software by modelling the software architecture as part of the on-going computation, thereby allowing it to evolve during execution.

The judicious mixture of formality in the Archware-ADL and its sophisticated support technologies yield an innovative approach to implementing active architectures for dynamic co-evolution. The language can describe the system's specification, the executing software and the reflective evolutionary mechanisms within a single computational domain in which all three may evolve in tandem.

Acknowledgements

The ArchWare Project was partially funded by the Commission of the European Union under contract No. IST-2001-32360 in the IST-V Framework. The work reported here also builds on earlier EPSRC-funded work in Compliant Systems Architectures [22] (GR/M88938 & GR/M88945).

References

[1] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)

[2] Balasubramaniam, D., Morrison, R., Mickan, K., Kirby, G.N.C., Warboys, B.C., Robertson, I., Snowdon, R., Greenwood, R.M., Seet, W.: Support for feedback and change in self-adaptive systems. In: WOSS'04. Proc. ACM SIGSOFT Workshop on Self-Managing Systems, Newport Beach, CA, USA, pp. 18–22. ACM Press, New York (2004)

[3] Balasubramaniam, D., Morrison, R., Kirby, G.N.C., Mickan, K., Warboys, B.C., Robertson, I., Snowdon, B., Greenwood, R.M., Seet, W.: A software architecture approach for structuring autonomic systems. In: DEAS 2005. Proc. ICSE Workshop on the Design and Evolution of Autonomic Application Software, St Louis, MO, USA, pp. 59–65 (2005)

[4] Chaudet, C., Greenwood, M., Oquendo, F., Warboys, B.: Architecture-Driven Software Engineering: Specifying, Generating, and Evolving Component-Based Software Systems. IEE Journal: Software Engineering 147(6) (December 2000)

[5] Chaudet, C., Oquendo, F.: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems. In: ASE'00. Proceedings of the 15th IEEE International Conference on Automated Software Engineering, Grenoble, September 2000, IEEE Computer Society, Los Alamitos (2000)

[6] Davies, J., Woodcock, J.: Using Z: Specification, Refinement and Proof. Prentice Hall International Series in Computer Science (1996)

[7] Fitzgerald, J., Larsen, P.: Modelling Systems: Practical Tools and Techniques for Software Development. Cambridge University Press, Cambridge (1998)

[8] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)

[9] Godfrey, M.W., Tu, Q.: Evolution in Open Source Software: A Case Study. In: ICSM'00. Proceedings of the International Conference on Software Maintenance, Washington, DC, October 11 - 14, 2000, pp. 131–142. IEEE Computer Society, Los Alamitos (2000)

[10] Gorlick, M.M., Razouk, R.R.: Using Weaves for Software Construction and Analysis. In: Proc. 13th International Conference on Software Engineering, Austin, Texas, United States, pp. 23–34. IEEE Computer Society Press, Los Alamitos (1991)

[11] Greenwood, M., Balasubramaniam, D., Cimpan, S., Kirby, N.C., Mickan, K., Morrison, R., Oquendo, F., Robertson, I., Seet, W., Snowdon, R., Warboys, B., Zirintsis, E.: Process Support for Evolving Active Architectures. In: Oquendo, F. (ed.) EWSPT 2003. LNCS, vol. 2786, Springer, Heidelberg (2003)

[12] Groenewegen, L.P.J., de Vink, E.P.: Evolution-On-The-Fly with Paradigm. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 97–122. Springer, Heidelberg (2006)

[13] IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (October 2000)

[14] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M., Morrison, R.: Persistent hyper-programs. In: Albano, A., Morrison, R. (eds.) Persistent Object Systems, 1992. Proc. 5th International Conference on Persistent Object Systems, Italy, pp. 86–106. Springer, Heidelberg (1993)

[15] Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. on Software Engineering 16(11), 1293–1306 (1990)

[16] Kramer, J., Magee, J.: Analysing Dynamic Change in Software Architectures: A Case Study. In: CDS 98. Proc. IEEE 4th Int. Conference on Configurable Distributed Systems, Annapolis, USA, May 1998, pp. 91–100. IEEE Computer Society Press, Los Alamitos (1998)

[17] Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EWSPT 1996. LNCS, vol. 1149, pp. 108–124. Springer, Heidelberg (1996)

[18] Mateescu, R., Oquendo, F.: pi-AAL: An Architecture Analysis Language for Formally Specifying and Verifying Structural and Behavioural Properties of Software Architectures. ACM Software Engineering Notes 31(2) (March 2006)

[19] Mickan, K.: A Meta-Programming Framework for Software Evolution. Ph.D. Thesis, University of St Andrews (2006)

[20] Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge (1999)

[21] Milner, R.: Computing in space. In: 17th International Congress on Computer Assisted Radiology and Surgery (CARS2003) (2003), http://www.cl.cam.ac.uk/users/rm135/

[22] Morrison, R., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Munro, D., Warboys, B.C.: A Compliant Persistent Architecture. Software, Practice & Experience 30, 1–24 (2000)

[23] Morrison, R., Kirby, G.N.C., Balasubramaniam, D., Mickan, K., Oquendo, F., Cîmpan, S., Warboys, B.C., Snowdon, B., Greenwood, R.M.: Support for evolving software

architectures in the ArchWare ADL. In: WICSA 4. Proc. 4th Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, pp. 69–78 (2004)

[24] Morrison, R., Balasubramaniam, D., Kirby, G.N.C., Warboys, B.C., Greenwood, R.M.: A Framework for Supporting Dynamic Systems Co-evolution. Journal of Automated Software Engineering (accepted for publication, 2007)

[25] Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-Based Runtime Software Evolution. In: Proc. 20th International Conference on Software Engineering, Kyoto, Japan, pp. 177–186. IEEE Computer Society, Los Alamitos (1998)

[26] Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., Occhipinti, C.: ArchWare: Architecting Evolvable Software. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, Springer, Heidelberg (2004)

[27] Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM Software Engineering Notes 29(3) (2004)

[28] Oquendo, F.: pi-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures. ACM Software Engineering Notes 29(5) (September 2004)

[29] Schmerl, B., Garlan, D.: Exploiting architectural design knowledge to support self-repairing systems. In: SEKE ’02. Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 15-19, 2002, vol. 27, pp. 241–248. ACM Press, New York (2002)

[30] Stemple, D., Fegaras, L., Stanton, R.B., Sheard, T., Philbrow, P., Cooper, R.L., Atkinson, M.P., Morrison, R., Kirby, G.N.C., Connor, R.C.H., Alagic, S.: Type-safe linguistic reflection: a generator technology. In: Atkinson, M.P., Welland, R. (eds.) Fully Integrated Data Environments, pp. 158–188. Springer, Heidelberg (1999)

[31] Stolen, K., Broy, M.: Specification and Development of Interactive Systems. Springer, Heidelberg (2001)

[32] Warboys, B.C., Kawalek, P., Robertson, I., Greenwood, R.M.: Business Information Systems: A Process Approach. McGraw-Hill, New York (1999)

[33] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R., Munro, D.S.: Collaboration and Composition: Issues for a Second Generation Process Language. In: Nierstrasz, O., Lemoine, M. (eds.) Software Engineering - ESEC/FSE ’99. LNCS, vol. 1687, pp. 75–91. Springer, Heidelberg (1999)

[34] Warboys, B.C., Greenwood, R.M., Robertson, I., Morrison, R., Balasubramaniam, D., Kirby, G.N.C., Mickan, K.: The ArchWare Tower: The Implementation of an Active Software Engineering Environment using a π-calculus based Architecture Description Language. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 30–40. Springer, Heidelberg (2005)

[35] Zhang, J., Cheng, B.H.: Model-based development of dynamically adaptive software. In: ICSE ’06. Proc. of the 28th international Conference on Software Engineering, Shanghai, China, May 20-28, 2006, pp. 371–380. ACM Press, New York (2006)

What's in a Service?

Michael P. Papazoglou

INFOLAB, Tilburg University, Dept. of Information Systems and Management,
Tilburg 5000 LE, The Netherlands
mikep@uvt.nl

Abstract. Automated services help enterprises create new value from reuse of software and resources and achieve new levels of agility through greater flexibility and lower cost structures. As services come in many flavors and guises they have differing characteristics. In this paper we describe the most necessary aspects and features of automated services. We also focus on the interplay of SOAs and Business Process Management technologies and argue that the key enablers for Service Oriented Architectures (SOAs) should focus on four inter-related elements: engineering and planning the SOA, SOA implementation, SOA management and monitoring and SOA governance.

Keywords: automated services, Service Oriented Architecture, business processes, Enterprise Service Bus, service-oriented design and development.

1 Introduction

Software services (or simply services) are self-contained, platform-agnostic computational elements that support rapid, low-cost and easy composition of loosely coupled distributed software applications [10]. The functionality provided by a service can range from answering simple requests to executing sophisticated processes requiring peer-to-peer relationships between multiple layers of service consumers and providers. Services are described, published, discovered, and can be assembled to create complex service-based systems and applications. Services help integrate applications that were not written with the intent to be easily integrated with other applications and define architectures and techniques to build new functionality while integrating existing application functionality.

Service-based applications are developed as independent sets of interacting services offering well-defined interfaces to their potential users. This is achieved without the necessity for tight coupling of applications between transacting partners, or for pre-determined agreements to be put into place before the use of an offered service is allowed.

Services may be implemented on a single machine or on a large number and variety of devices, and be distributed on a local area network or more widely across several wide area networks (including mobile and ad hoc networks). A particularly interesting case is when the services use the Internet (as the communication medium) and open Internet-based standards. The resulting *Web services* share the characteristics of more general services, but they require special consideration as a result of using a public, insecure, low-fidelity mechanism for service-mediated interactions. Web services constitute a

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 11 – 28, 2007.

distributed computer infrastructure made up of many different interacting application modules trying to communicate over private or public networks (including the Internet and Web) to virtually form a single logical system. Web services expose their features programmatically over the Internet (or intra-net) using standard Internet languages (based on XML) and standard protocols, and are implemented via a self-describing interface based on open Internet standards.

Web services[1] can vary in function from simple requests (for example, credit checking and authorization, pricing enquiries, inventory status checking, or a weather report) to complete business applications that access and combine information from multiple sources, such as, for instance, an insurance brokering system, an insurance liability computation, an automated travel planner, or a package tracking system.

In this paper we shall examine the service concept more closely and deconstruct its meaning. First, we shall concentrate on the various facets and characteristics of services and provide a more complete view of how its various parts fit together. Subsequently, we shall provide a brief overview of Service Oriented Architecture (SOA) and examine key SOA enablers that form the core of (inter- and cross)-enterprise integration.

2 The Multiple-Aspects of a Service

Services come in many flavors and guises and thus have many aspects to them. In the following we summarize the most necessary aspects of a service.

2.1 Service Types

Topologically, services can come in two flavors: (simple) informational or complex services. Each of these two models exhibits several important distinguishing characteristics that are briefly described below.

Informational services are services of relatively simple nature. They either provide access to content interacting with an end-user by means of simple request/response sequences, or alternatively they may expose back-end business applications to other applications. Informational services can be of various kinds:

1. *Pure content services*, which give programmatic access to content such as simple financial information, stock quote information, design information, news items and so on.
2. *Seamless aggregation services*, which provide seamless aggregation of information across disparate systems and information sources including back-end systems. These give programmatic access to a business service so that the requestor can make informed decisions. Consider for example, “pure” business services, such as logistic services, where automated services are the actual front-ends to fairly complex physical organizational business processes.
3. *Information syndication services*, which are services offered by a third-party and run the whole range from commerce-enabling services, such as logistics, payment, fulfillment, and tracking services, to other value-added commerce services, such as rating services.

[1] In this paper we shall use the terms Web service and service interchangeably.

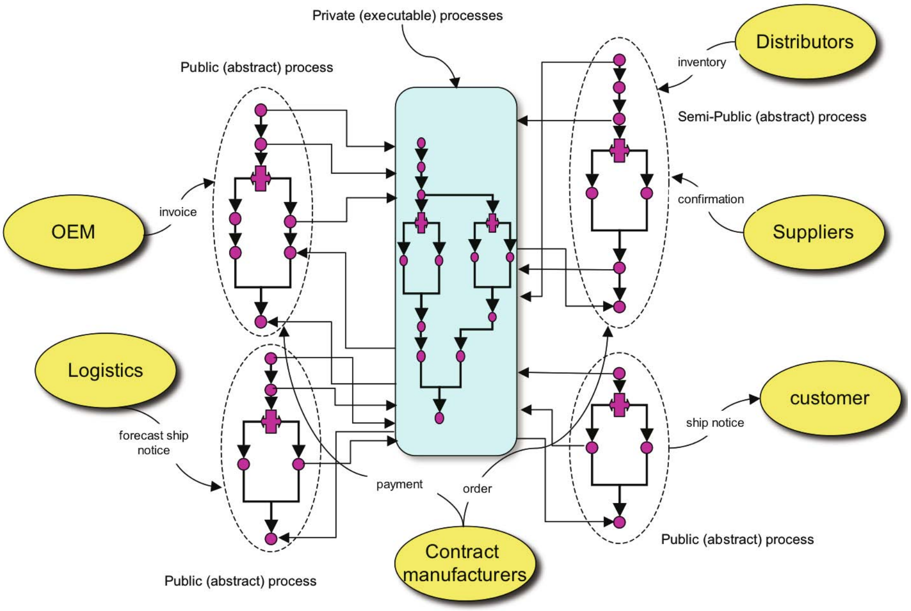

Private (executable) processes
Public (abstract) process
Distributors
inventory
Semi-Public (abstract) process
confirmation
Suppliers
OEM
invoice
Logistics
forecast ship notice
customer
ship notice
payment
order
Contract manufacturers
Public (abstract) process
Public (abstract) process

Fig. 1. A typical manufacturing business processes

An informational service is stateless in that it does not keep any memory of what happens to it between requests.

Complex services typically involve the assembly and invocation of many pre-existing services, possibly found in diverse enterprises, to complete a multi-step business interaction – called a business process. Consider for instance a supply-chain application involving an Original Equipment Manufacturer (OEM) such as the one depicted in Figure 1. The business process in Figure 1 handles order management, sourcing, inventory control, financials and logistics. The OEM in Figure 1 is shown to deal with suppliers, distributors, customers and contract manufacturers. OEMs typically outsource the manufacturing of subassemblies or entire products to contract manufacturers who perform functions including product design, sourcing, planning, production, fulfillment, and reverse logistics. Numerous document exchanges will occur in the business process illustrated in Figure 1 including requests for quotes, returned quotes, purchase order requests, purchase order confirmations, delivery information and so on.

In many circumstances complex services combine programmatic behavior with interactive behavior delivering thus business processes that combine typical business logic functionality with user interactivity whenever this is required as part of the process.

2.2 Service Facades

A service contains minimally four facade elements, which are part of its specification:

1. *Structural facade*: This focuses on defining the service types, messages, interfaces and operations. The structural façade deals mainly with the syntax of a service.

2. *Behavioral facade*: This entails understanding the effects and side effects of service operations and the semantics of input and output messages. If, for example we consider an order management service we might expect to see a service that lists "place order," "cancel order," and "update order," as available operations. The behavioral specification for this ordering service might then describe how a user cannot update or cancel an order she did not place, or that after an order has been cancelled it cannot be updated.
3. *Policy facade*: To address quality of service (QoS) considerations a service policy specification mechanism is necessary. Policy specification describes policy assertions and constraints on the service, which prescribe, limit, or specify any aspect of a business agreement that is agreed to among interacting parties. Policies may describe constraints external to constraints agreed by interacting parties in a transaction and include universal legal requirements, commercial and/or international trade and contract terms, public policy (e.g., privacy/data protection, product or service labeling, consumer protection), laws and regulations that are applicable to parts of a business service. Policy assertions usually cover non-functional characteristics including security, performance rates, transactional features, and so on. For instance, a policy specification may indicate that certain elements of the order need be encrypted, denoting the encryption techniques to be used, certificates to use, and so forth. Such non-functional requirements can be specified using appropriate constructs from the WS-Policy specification framework [1].
4. *Vocabulary and best practices façade*: This part includes definition of common business processes, e.g. sending a purchase order to achieve canonization and standardization of processes and services; definition of common data-interchange formats i.e., messages that are exchanged in the context of the above processes/transactions; and, definition of a common terminology at the level of data items and messages seeking a way to bridge varying service terminologies.

2.3 Provisioning and Charging Model

For services to create economic value it is important that service providers come up with viable business models that address factors such as business service metering, rating and billing.

Service providers can develop a service-metering model that estimates the use of a service by a client in case that the service provider requires usage-based billing. In this case the service provider needs to audit the business service (see following section for a definition) as it is used and bill for it. This could typically be done on a periodic basis and requires that a metering and accounting model for the use of the service be established. Alternatively, a pricing (rating) model could determine subscriber rates based on subscription and usage events. For example, the pricing model could calculate charges for services based on the quality and precision of the service and on individual metering events based on a service-rating scheme. Finally, service charging model may include alternatives such as payment on a per use basis, payment on a subscription basis, payment on a leasing basis, lifetime services, free services, and free services with hidden value.

2.4 Operational Categories of Services

We may discern three broad categories of services classified according to their operational characteristics: business functionality services, technical services and utility services.

A *business service* automates a generic business task with significance to the business, e.g., creating a customer record, creating an invoice or closing an open customer service ticket. It provides value to an enterprise and is part of standard business process.

A Web services environment contains a collection of *technical services*, which are coarse-grained services that provide the technical infrastructure enabling the development, delivery, maintenance and provisioning of singular business services and their integration into processes as well as capabilities that maintain QoS such as security, performance, and availability. It also contains another breed of technical services that monitor the health of SOA applications, giving insights into the health of systems and networks, and into the status and behavior patterns of applications making them thus more suitable for mission-critical computing environments (see section-6).

Finally, a Web services environment contains a collection of fine-grained *utility* (or commodity) *services*, which provide value to and are shared by business services across the organization. Examples of utility services include services implementing calculations, algorithms, directory management services and so on.

3 Service Oriented Architecture

Service orientation utilizes services as constructs to support the rapid, low-cost and easy composition of distributed applications. Key to this concept is the service-oriented architecture (SOA), which is a logical way of designing a software system to provide services to either end-user applications or to other services distributed over a network, via published and discoverable interfaces. A well-constructed SOA can empower a business environment with a flexible infrastructure and processing environment by provisioning independent, reusable automated business processes (as services) and providing a robust foundation for leveraging these services.

Business processes form the foundation for SOAs and require that multiple steps occur between physically independent yet logically dependent services. Underlying the need for flexibility in SOA is the ability to rapidly assemble new services and business processes to address business needs. An important characteristic of SOAs is that they are impacted by industry regulations. Without explicit business process definitions, flexible rules frameworks, and audit trails that provide for non-repudiation, organizations face litigation risks. Acts such as the Sarbanes-Oxley requires all organizations to review their business processes and ensure that they meet the compliance standards set forth in the legislation. This can include, but is not limited to, data acquisition and archival, document management, data security, financial accounting practices, and shareholder reporting functions. When a compliance-model, such as the Sarbanes-Oxley Act, is used to drive business processes enforces the exchange of reliable, credible information, adherence to regulations as well as visibility and transparency of business processes between interacting parties.

The following key SOA enablers form the core of (inter- and cross)-enterprise integration: engineering and planning the SOA, SOA implementation, SOA management and monitoring, and governing the SOA.

We shall examine each of these SOA enablers in turn in the following sections.

4 Engineering and Planning the SOA

When software developers are building a service-oriented application, they must rely on a service-based development methodology [8]. Such a methodology focuses on analyzing, designing and producing an SOA in a way that it aligns with business process interactions between trading partners in order to accomplish a common business goal, e.g., requisition and payment of a product, and stated functional and non-functional business requirements, e.g., performance, security, scalability, and so forth. Without a service-based development methodology, projects aiming at building SOA processes are doomed to be overtime, violate quality criteria, incorporate designs that infringe basic principles (such as loose-coupling), are hard or virtually impossible to manage, and, are likely to result in systems that are notably intricate, time-consuming and costly to maintain.

A service-oriented design and development (SoDD) methodology provides sufficient principles and guidelines to specify and construct business processes choreographed from a set of internal and external Web services [16]. Adopting a service-oriented approach to solutions development necessitates a broader review of its impact on how solutions are designed; what it means to assemble them from disparate services; and how deployed services-oriented applications can evolve and be managed. This requires addressing common concerns such as the identification, specification and realization of services, their flows and composition into processes, as well as the enterprise-scale components needed to realize them and ensure the required QoS.

4.1 SOA Layers of Abstract Functionality

SOA-based development is facilitated when we view the way that SOA operates as comprising a number of layers of abstract functionality, each with its own category of artifacts that is characterized by its own set of properties and relationships. Each layer helps organize SOA functionality at the most appropriate level of detail. This is illustrated in Figure 2, where the basic SOA model is divided into six major layers of abstraction: domains, business processes, business services, infrastructure services, service realizations and operational systems.

The logical flow employed in the layered SOA development model usually focuses on a top-down, a bottom-up or a meet-in-the middle development approach. The top-down development approach requires an holistic view of business processes and their interactions in an enterprise and emphasizes how business domains are decomposed into a collection of business processes, how business processes are decomposed into constellations of business services, and how these services are implemented in terms

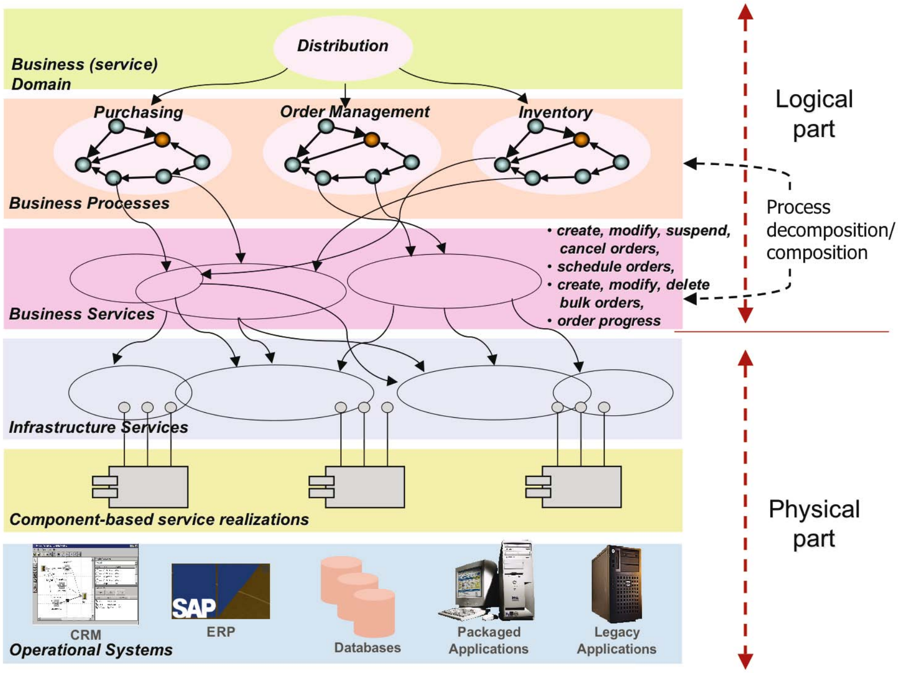


Fig. 2. SOA layers of abstract functionality

of pre-existing enterprise assets. The bottom-up approach emphasizes how existing enterprise assets (e.g., databases, enterprise information systems, legacy systems and applications) are transformed into business services and how business services are in turn composed into business processes. The bottom-up approach can originate on the departmental and group level, starting by exposing existing applications as services and building around those services. The most common approach is, however, to combine top-down and bottom-up approaches, starting at each end and meeting in the middle.

Figure 2 shows that a service domain such as distribution, which is part of the complex application in Figure 1, is subdivided into a small number of higher-level standard business processes such as purchasing, order management and inventory. The order management business process in Figure 2 typically performs order volume analysis, margin analysis, sales forecasting and demand forecasting across any region, product or period.

In Figure 2, the order management process provides business services for creating, modifying, suspending, canceling, querying orders, for creating and tracking orders for a product, a service or a resource, and capturing customer-selected service details. Business processes orchestrate the execution of several finer-grained business services to fulfill the required business functionality and are thus the units of decomposition into (top-down approach) or composition from (bottom-up approach) business services. Business services are the appropriate units of business process and transaction analysis as they identify business processes and transactions and associated business costs, and achieve reuse of resources across enterprises and business units [7].

In an SOA the business services layer provides a kind of conceptual bridge between the higher-level business-oriented layers and the lower-level technical implementation layers. We may thus collectively think of the service domain, business processes and business services sections as comprising the *logical part* of services development life cycle, see Figure 2.

Business services are supported by infrastructure, management and monitoring services such as those providing technical utility, for instance, logging, security, or authentication, and those that manage resources. These services provide the infrastructure often considered as part of the Enterprise Service Bus (see section-5).

Service domains, business processes and services are layered on a backdrop of a collection of operational functions and data available from resources such as ERP, databases, and CRM systems as well as other enterprise resources. The component realization layer in an SOA identifies and characterizes a large number of components that provide service implementations in terms of existing resources. Here, bottom-up analysis of exiting applications and systems is required.

In a similar manner to the logical part of Web services development life cycle, we may think of the infrastructure services, the component-based realizations, and the operational systems sections as comprising the *physical part* of Web services development life cycle. The physical part of the Web services development life cycle is intended to accelerate service implementation.

4.2 Phases in Service-Oriented Design and Development Methodology

The main concern of SoDD is that business goals and requirements should drive downstream design, development, and testing to transform business processes into composite applications that automate and integrate enterprises. In this way business requirements can be traced across the entire lifecycle from business goals, through software designs and code assets, to composite applications.

A SoDD methodology comprises a number of phases typically encompassing planning, analysis and design, construction and testing, provisioning, deployment, execution and monitoring. These phases are traversed iteratively and feedback is cycled to and from phases in iterative steps of refinement and the methodology may actually be built using a blend of forward- and reverse-engineering techniques or other means to facilitate the needs of the business. This approach considers multiple realization scenarios for business processes and services that take into account both technical and business concerns. In the following we briefly summarize the service analysis and design phases of SoDD. More information about the remaining phases as well as the SoDD methodology can be found in [8] and [10].

4.2.1 Service Analysis

Service analysis aims at identifying and describing the processes and services in a business problem domain, and on discovering potential overlaps and discrepancies between processes under construction and available system resources that are needed to realize singular services and business processes. Service analysis helps prioritize business processes and services where SOA can contribute to improvements and offer business value potential. It also helps center efforts on business domains within an enterprise that can be mapped to core business processes.

The analysis phase reviews the business goals and objectives of an enterprise, since these drive the development of business processes. It helps focus SOA initiatives by creating a high-level process map that identifies business domains and processes of particular interest to an enterprise. Business processes are ranked by criteria related to their value and impact, reuse and high consumption, feasibility and technical viability. From the process map, analysts can identify candidate business services that relate to these processes. Service analysis is aimed at enabling the business analyst to model their current (as-is) and new (to-be) processes using modeling environments, simulate their processes to test for effectiveness and impact, define their processes, and manage business processes proactively, feeding business-activity management information back to the analyst for ongoing optimization (see section-6.2).

4.2.2 Service Design

Designing a service-oriented application requires developers to define related, well-documented interfaces for all conceptual services identified by the analysis phase, prior to constructing them.

The design phase encompasses the steps of singular service specification, business process specification, and policy specification for both singular services and business processes. Service design is based on a twin-track design approach that provides two production lines – one along the logical part and one along the physical part of the SOA – and considers both functional and non-functional service characteristics. The purpose of logical service design is to define singular services and assemble (compose) services out of reusable singular service constellations. This calls for a business process model that forces developers to determine how services combine and interact jointly to produce higher level services. The physical design trajectory focuses on how to design component implementations that provide services at an acceptable level of granularity. Physical design is thus based on techniques for leveraging legacy applications and component-based development.

5 Implementing the SOA

A successful SOA deployment requires adopting and installing a service integration backbone that mediates differences between services, known as an Enterprise Service Bus (ESB) [3, 5, 14]. The ESB is an open, standards-based message bus designed to enable the implementation, deployment, and management of SOA-based solutions with a focus on assembling, deploying, and managing distributed SOA [12]. The ESB provides the distributed processing, standards-based integration, and enterprise-class backbone required by the extended enterprise [4].

An ESB provides a necessary layer of intermediation in SOA, insulating services’ end points from direct connection to each other, and a powerful backbone for those services to plug into. The purpose of an ESB is to enable service composition across the entire lifecycle of service delivery—from connecting heterogeneous services running on various enterprise systems (legacy, packaged, and custom applications), to dynamically mediating and transforming messages between disparate services to

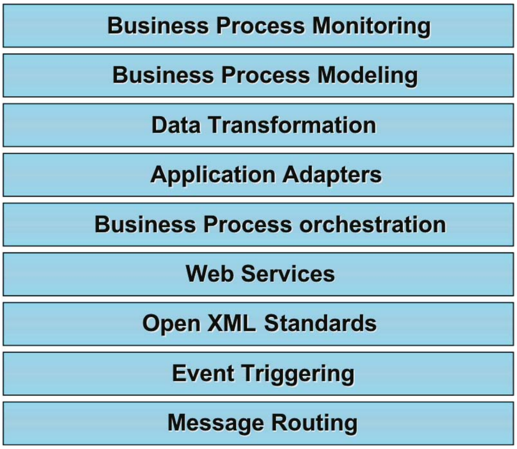


Fig. 3. ESB functionality layers

accomplish steps within a business process, to managing and monitoring service interactions to ensure QoS and Service Level Agreement (SLA) compliance. Figure 3 illustrates the key functional layers of an ESB.

The ESB becomes the foundation for further expansion of enterprise integration for SOA. It achieves this by streamlining communication between disparate services by removing the integration logic from the service end points. This capability shields services, processes, and users from service changes. As part of this streamlining, the service-integration layer extends to and bridges multiple protocols, message styles, security policies, and data formats.

The most common ESB components are:

- *Leveraging existing assets*: the objective is to leverage legacy assets and integrate them with modern technologies and applications.
- *Service communication:* service interactions must be routed through a variety of protocols, and transformed from one protocol to another where necessary. Another important aspect of an ESB implementation is the capacity to support service-messaging models consistent with the SOA interfaces, as well as the ability of transmitting the required interaction context, such as security, transaction, or message correlation information.
- *Dynamic connectivity:* Web services can connect dynamically without using a separate static API or proxy for each service. The dynamic connectivity API is the same regardless of the service implementation protocol (Web services, JMS, EJB/RMI, etc.).
- *Topic/content-based routing:* should facilitate not only topic-based and itinerary-based routing but also, more sophisticated, content-based routing (with declarative routing rules).
- *Transformation and mapping:* to ensure that messages and data received by any component is in the format it expects, thereby removing the burden to make changes. Transformation can range from simply mapping data formats of message

payloads to providing rich aggregation or decomposition of service semantics so that each side of the conversation is unaware of details of the other.

- *Service orchestration, aggregation, and process management:* fine-grained services should be aggregated into coarse-grained services. To that effect, many ESB vendors provide service orchestration tools and BPEL-based runtimes.
- *Endpoint discovery with multiple QoS:* SOAs need to discover, locate, and bind to services with different values to the business. Several scenarios make it desirable for the client to select the best endpoint at run-time, rather than hard-coding endpoints at build time.
- *Reliable messaging:* supports asynchronous store-and-forward delivery as well as guaranteed delivery capabilities. Primarily used for handling events, this capability is crucial for responding to clients in an asynchronous manner, and for a successful ESB implementation.
- *Security model:* should be provided to service consumers and should integrate with the (potentially varied) security models of service providers. Both point-to-point (e.g., SSL encryption) and end-to-end security capabilities are supported. To address these intricate security requirements trust models, WS-Security [13] and other security related standards have been developed.
- *Monitoring and management:* a crucial goal of the ESB is to manage applications that cross system and organizational boundaries, overlap, and may ultimately change over time (see following section).

6 Managing the SOA

In SOA solutions, service usage patterns, SLA criteria and metrics, and failure data should be continuously collected and analyzed. Without such information, it is often quite challenging to understand the root cause of an SOA-based application’s performance or stability problems. A Web services measurement and management infrastructure provides comprehensive ways of understanding exactly what is involved in a business process so it can cross organizational boundaries and function as an integral element in an end-to-end process chain that spans organizations and value chains.

Web services need to be managed in at least two dimensions [10]:

1. The *operational* (or infrastructure) *management* dimension, in which a systems administrator starts and stops Web services and keeps track of how many instances of a Web service are running, in which containers, and on which remote systems.
2. The *tactical* (or business) *management* dimension that provides a number of business activity monitoring and analytics capabilities that enable some human agent to watch over business operations, identifying opportunities and diagnosing problems as they occur so as to ensure that the Web services supporting a given business task are performing in accordance with service level objectives.

We shall examine these two Web services management dimensions in what follows.

6.1 Distributed Infrastructure Services Management

An orchestrated service implementation forms a logical network of services layered over the infrastructure (including software environments, servers, legacy applications, and back-end systems) and the physical connectivity network. Any such environment should be managed in order to provide the security, usability, and reliability needed in today's business environment.

Web services management is defined as the functionality required for discovering the existence, availability, performance, health, patterns of usage, extensibility, as well as the control and configuration, life-cycle support and maintenance of a Web service or business process within the context of SOAs [10]. This definition suggests a manageability model that applies to both Web services and business processes in terms of manageability topics, (identification, configuration, state, metrics, and relationships) and the aspects (properties, operations and events) used to define them [11]. A Web services management framework performs four activities as part of end-to-end Web services management [10]:

1. Measuring end-to-end levels of service for user-based units of work, e.g., individual Web services, business processes, application transactions, and so on.
2. Breaking such units of work down into identifiable, measurable components, e.g. requests for application transactions.
3. Attributing end-to-end service levels and resource consumption to such units and their components. This involves tracing and monitoring them through multi-domain, geographical, technological, application, and supplier infrastructures.
4. Identifying and predicting current problems and future requirements in user terms.

To perform the above activities a Web services management framework must take into consideration a number of interrelated Web services management aspects. The most prominent functions of service management include the following.

- *Performance indicators and metrics:* service-focused management requires measurable key performance indicators (KPIs) for applications and services to ensure quality. Generic KPIs for business applications include service availability, response time, transaction rate, service throughput, service idle time, and security. Service providers set the service KPIs and their objectives, and are able to measure, analyze, and report on the KPI values achieved against these objectives.
- *Auditing, monitoring, and troubleshooting:* Web services should be audited and these measurements can be used to compare against service level agreements. Resources should be monitored for bottlenecks and for user-defined thresholds that exceed the prescribed limits. These measurements can be recorded and the historical performance data collected and used for capacity planning.
- *Infrastructure monitoring:* distributed resources, such as servers and networks, but also critical software infrastructure, must be monitored for availability, performance and utilization so that developing problems can be quickly detected. The management infrastructure must know how many instances of a service are running, whether a service is performing adequately and when a service has stopped functioning and how to readily connect and disconnect services and their clients without breaking applications or infrastructure.

- *Resource provisioning:* when a Web services-based application is running careful monitoring of system activity can help to identify potential problems before they are manifested in overloads or failures. For instance, when a server cluster no longer has the capacity to handle the demands being made upon an application, additional servers are provisioned and configured.

In a Web services management environment a *resource manager* is the module that implements a control loop that collects details regarding a managed resource and acts accordingly. The resource manager realizes the management capabilities and functions of management solutions. To achieve this, it collects the details it needs from a Web service (or business process); analyzes these details to determine if something needs to change; it then creates a plan, or sequence of actions, that specifies the necessary changes; and finally it performs these actions. These four parts work in tandem to provide the control loop functionality.

A resource manager relies on a *monitor* module that aggregates, correlates and filters managed resource meta-data such as metrics, topology information, configuration property settings, offered capacity, throughput and so on, and organizes them into symptoms that require analysis. A resource manager uses such monitored data, for instance, during the analysis process, to generate a change request or the execute function to perform system self-optimization. Selected monitored data also could be presented to a systems administrator on a management console.

6.2 Business Process Management

Tactical Web services management provides end-to-end visibility and control over all parts of a long-lived, multi-step transaction/process that spans multiple applications and human actors in one or more enterprises. Tactical Web services management issues, e.g., management of end-to-end SLAs, metrics that ensure conformance, and so on, are usually addressed by Business Process Management (BPM).

BPM manages the lifecycle of a process starting from business goals and definition, through deployment, execution, measurement, change, and redeployment. A BPM suite offers advanced modeling and simulation tools that give the ability to not only model graphical representations of a process, but also to simulate the impacts of changes to a process to measure its effect on numerous variables and KPIs including cost, profitability, resource utilization, throughput speed, and other critical business objectives. Statistical analysis models, such as Six Sigma, can also work within the simulation environment. As processes run, the BPM engine continuously captures snapshots of instance data and aggregates them in visible performance metrics, both built-in and user-defined.

BPM is a natural complement to SOA, and a mechanism through which an organization can apply SOA to high-value business challenges. The objective is to effectively align technical initiatives with the strategic goals of the business user at every level within the organization to achieve a comprehensive approach to real business transformation.

Both SOA and BPM can each be pursued without the other, but the two approaches in concert offer reciprocal benefits. Layering BPM on top of a solid SOA allows

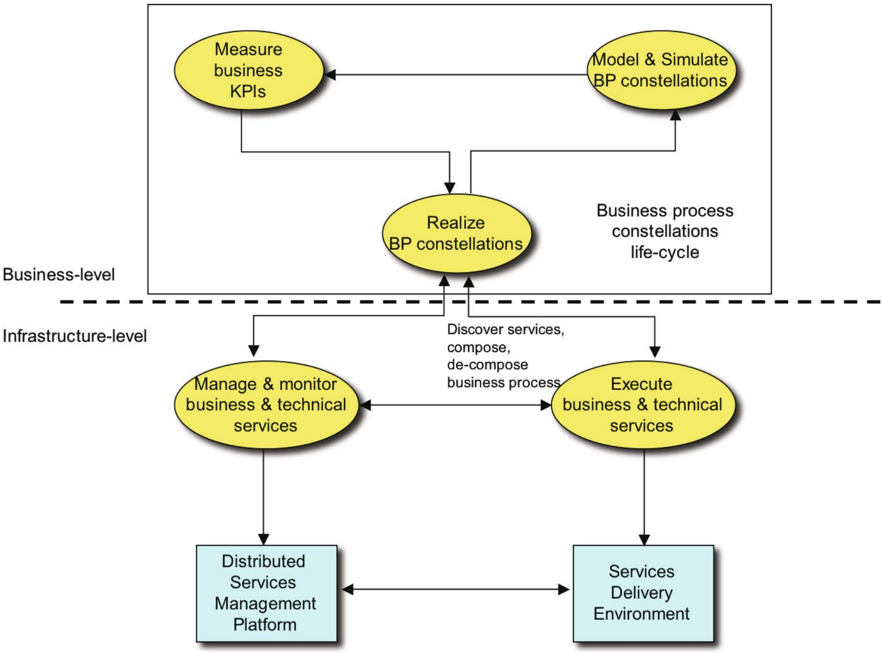

Measure business KPIs
Model & Simulate BP constellations
Realize BP constellations
Business process constellations life-cycle
Business-level
Infrastructure-level
Discover services, compose, de-compose business process
Manage & monitor business & technical services
Execute business & technical services
Distributed Services Management Platform
Services Delivery Environment

Fig. 4. The business and infrastructure levels for Web services management

actions within business processes to be exposed via automated services and SOAs. With BPM orchestration, the exposure of key business events and information to users at the appropriate times and in the appropriate contexts adds tremendous business value that might not otherwise be achieved with a conventional SOA. In addition, BPM helps deliver control over business processes, fostering standardization across a company or an end-to-end process chain and compliance with regulations, policies, and best practices. It also enables some services required by the business process to be outsourced to trading partners, and opens up brand-new business models in which the enterprise's own business processes can be exposed as services to new customers, both internal and external. Figure 4 illustrates how the SOA and BPM approaches are pursued in concert.

6.3 Connecting the Business Service and Infrastructure Channels in an SOA

Figure 5 is a further illustration of how management and business application channels can be developed in accordance with SOA principles [10]. This architecture continuously connects to the Web services application channel and directs it into the management channel. Typical management services that are common across both the business applications and management channels, include:

1. SLA and QoS management, including the measurement of performance and availability, as well as alerting services.

2. Visibility and control capabilities, including interactive monitoring, administration, and reporting.
3. Service adaptability, including versioning, routing, differentiated services, and message transformation.
4. Web-services and XML-based security mechanisms, i.e., WS-Security related standards.

An enterprise based on this architecture can extend business applications with special management capabilities or add management capabilities as required.

In the architecture depicted in Figure 5, service management involves a collection of management infrastructure services that communicate with each other — passing data or coordinating some activity — to facilitate the delivery of one or more business services. The architecture is generic and does not prescribe the use of a particular management protocol or instrumentation technology; it can rather work with existing and future technologies and standards.

In Figure 5, managed resources include physical and logical hardware and software resources. These resources expose their management capabilities as Web services that implement various management interfaces, such as those defined in the Web Services Distributed Management (WSDM) standard [15]. WSDM defines a protocol for the interoperability of management information and capabilities via Web services. To resolve distributed system management problems, WSDM focuses on two distinct tasks: management using Web services (MUWS) and management of Web services (MOWS).

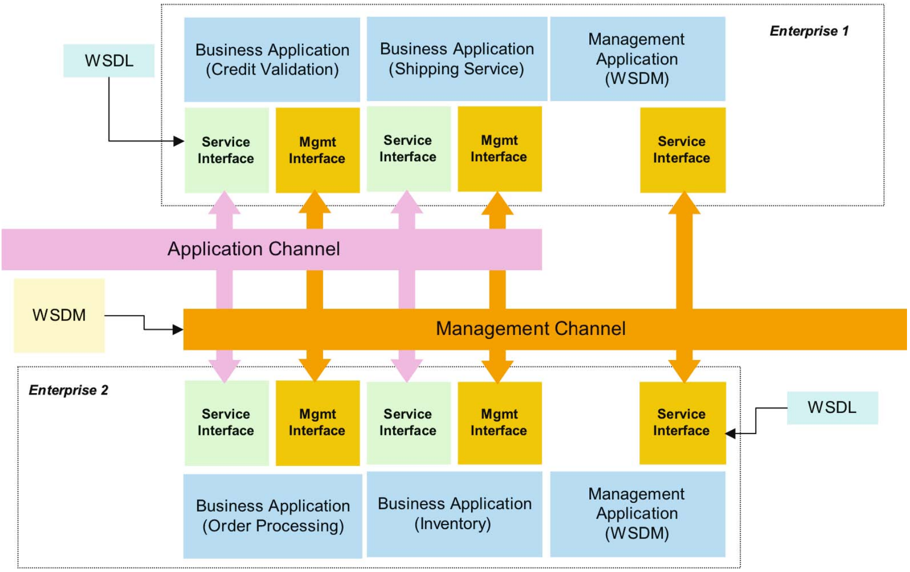


Fig. 5. Developing and managing Web services-based applications

Figure 5 shows that a business process integrates basic services such as credit validation, shipping, order processing, and inventory services originating from two collaborating enterprises. The architecture's management applications, such as performance management, capacity planning, asset protection, job control, manage resources through their management interfaces or management infrastructure services [6]. A resource's management interface is described by a WSDL document, resource properties schema, metadata documents, and (potentially) a set of management-related policies. Resource managers interact with managed resources and management infrastructure services using the Web services interfaces. In addition, service managers leverage Web services technologies, such as BPEL, to describe and execute management processes that perform a "scripted" management function [6].

7 Governing the SOA

A significant challenge to widespread SOA adoption is for SOAs to deliver value. To achieve this, there must be control in areas ranging from how a cross-organizational end-to-end business process that is composed out of variety of service fragments is built and deployed, how QoS is enforced, proven and demonstrated to service consumers, to granular items such as XSD schemas and WSDL creation. This requires efficient SOA governance.

SOA governance refers to the organization, process, policies and metrics that are required to manage an SOA successfully [7]. In particular, SOA governance is a formalization of the structured relationships, procedures and policies that ensure the IT functions in an organization support and are aligned to business functions, with a specific focus on the life cycle of services. SOA governance is primarily designed to enable enterprises to maximize business benefits of SOA such as increased process flexibility, improved responsiveness, and reduced IT maintenance costs.

Services that flow between enterprises have defined owners with established ownership and governance responsibilities, including gathering requirements, design, development, deployment, and operations management for any mission critical or revenue generating service. SOA governance introduces the notion of business domain ownership, where domains are managed sets of services sharing some business context to guarantee that services their functional and QoS objectives both within the context of a business unit and the enterprises within which they operates [2].

Two different governance models are possible [10]. These are *central governance* versus *federated governance*. With central governance, the governing body within an enterprise has representation from each business domain as well as from independent parties that do not have direct responsibility for any of the service domains. With federated governance each business unit has autonomous control over how it provides the services within its own enterprise, while a central governance committee, which has an advisory role, can provide guidelines and standards to different teams.

Figure 6 illustrates the role that the ESB plays with respect to SOA governance. As this figure illustrates, the ESB can play an important role in SOA governance by monitoring business activities and handling exceptions, by providing runtime monitoring capabilities to check policy compliance and provide audit trails, facilitating service management, resource and business process optimization.

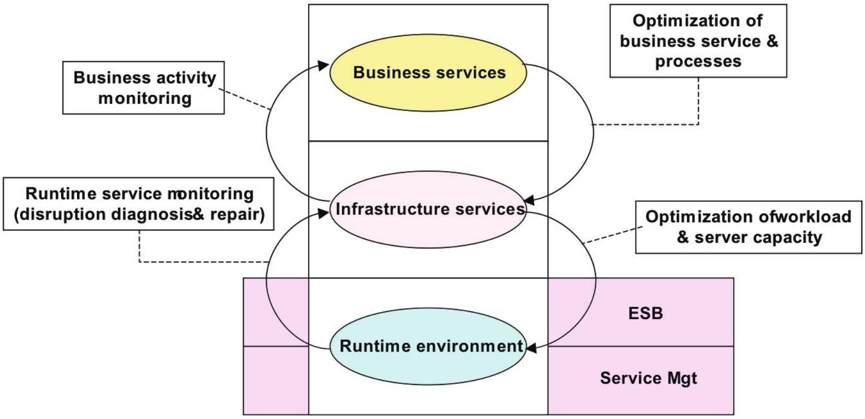

Business activity monitoring
Business services
Optimization of business service & processes
Runtime service monitoring (disruption diagnosis& repair)
Infrastructure services
Optimization ofworkload & server capacity
ESB
Runtime environment
Service Mgt

Fig. 6. Developing and managing Web services-based applications

8 Summary

Automated services help enterprises create new value from existing investments, reuse of software and resources, and achieve new levels of agility through greater flexibility and lower cost structures. As services come in many flavors and guises they have differing characteristics. We argue that the most necessary and minimal set of characteristics for services must include elements such as two service types: informational and complex services, service facades such as a structural, a behavioral, and a policy façade, a vocabulary of terms, as well as a provisioning and charging model.

Service Oriented Architectures empower a business environment with a flexible infrastructure and processing environment by provisioning independent, reusable automated business processes (as services) and providing a robust foundation for leveraging these services. Effective SOAs must rely on a set of core of core enablers, which include engineering and planning the SOA, SOA implementation, SOA management and monitoring, and governing the SOA.

References

1. Bajaj, S., et al.: Web Services Policy Framework (WS-Policy) Version 1.2 (March 2006), available at: http://xml.coverpages.org/ws-policy200603.pdf
2. Bieberstein, N., et al.: Service-Oriented Architecture (SOA) Compass. IBM Press (2006)
3. Chappell, D.: Enterprise Service Bus. O’Reilly Media, Inc. (2004)
4. Chappell, D.: ESB myth busters: Clarity of Definition for a Growing Phenomenon. Web Services Journal, 22–26 (February 2005)
5. Colan, M.: Service-Oriented Architecture Expands the Vision of Web Services, Part 2. IBM DeveloperWorks (April 2004)
6. Kreger, H.: A Little Wisdom about WSDM. IBM DeveloperWorks, available at: http://www-128.ibm.com/developerworks/library/ws-wisdom

7. Marks, E.A., Bell, M.: Service-oriented Architecture: A Planning and Implementation Guide for Business and Technology. J. Wiley, Chichester (2006)
8. Papazoglou, M.P., van den Heuvel, W.J.: Service-oriented Design and Development Methodology. Int. J. Web Engineering and Technology 2(4), 412–442 (2006)
9. Papazoglou, M.P., van den Heuvel, W.J.: Service Oriented Architectures: Approaches, Technologies and Research Issues. The VLDB J. 16(3), 389–415 (2007)
10. Papazoglou, M.: Web Services: Principles and Technology. Prentice-Hall, Englewood Cliffs (2007)
11. Potts, M., Sedukhin, I., Kreger, H.: Web Services Manageability – Concepts (WS-Manageability). IBM, Computer Associates International, Inc., Talking Blocks, Inc. (September 2003), available at: www3.ca.com/Files/SupportingPieces/ web_service_manageability_concepts.pdf
12. Ritter, T., Scott Evans, R.: SOA Research Costs and Benefits. Gartner Custom Research (2006), available at: http://www.gcrinsight.com/
13. Rosenberg, J., Remy, D.: Securing Web Services with WS-Security. Sams Publishing (2004)
14. Schulte, R.: Predicts 2003: Enterprise Service Buses Emerge. Report, Gartner (December 2002)
15. Vambenepe, W. (ed.): Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1. OASIS Standard (March 2005), available at: http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf
16. Zimmerman, O., Korgdahl, P., Gee, C.: Elements of Service-oriented Analysis and Design. IBM DeveloperWorks (June 2004), available at: http://www-106.ibm.com/developerworks/library/ws-soad1

Pattern-Based Evolution of Software Architectures

Isabelle Côté, Maritta Heisel, and Ina Wentzlaff

University Duisburg-Essen, Faculty of Engineering, Department of Computational and Cognitive Sciences - CoCoS, Working Group Software Engineering, Germany
{isabelle.cote, maritta.heisel,ina.wentzlaff}@uni-due.de

Abstract. We propose a pattern-based software development method comprising analysis (using problem frames) and design (using architectural and design patterns), of which especially evolving systems benefit. Evolution operators guide a pattern-based transformation procedure, including re-engineering tasks for adjusting a given software architecture to meet new system demands. Through application of these operators, relations between analysis and design documents are explored systematically for accomplishing desired software modifications. This allows for reusing development documents to a large extent, even when the application environment and the requirements change.

1 Motivation

Splitting the software life cycle into several, more or less independent development phases is a need to create manageable engineering activities. Patterns introduce a further enhancement, because they provide a concept for reusing software development knowledge. Hence, a vast quantity of patterns specific to and applicable in the different phases of the software life cycle can be found today.

It has been observed that the life-span of software often covers several years, in some cases even decades. During this long lifetime, it is necessary to modify and update existing software to accommodate it to new requirements or a changing environment, where it is deployed in. Modifying existing software systems to adapt them to new or changed requirements is called *software evolution*. Existing software development processes, however, are not designed to incorporate new or changing requirements into an existing system. They usually consider a system that has to be built from scratch. This is a striking fact, as experts see the fraction of maintenance/evolution at 80% of the overall effort for a software project [9]. For this reason, it is necessary to provide systematic support for the evolution task. Ideally, this support can be embedded into an existing development process. This puts new demands on the relation of software development process and pattern usage. Re-engineering techniques and reuse, as well as the traceability between the different development artifacts are crucial in this context.

Each phase of the software life cycle has different objectives. However, even if the engineering activities of the respective software development steps are independent of each other, the resulting artifacts are not. Sudden architectural change by innovation is not desirable. Instead, architectural changes are usually motivated by the need for

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 29–43, 2007.

adding new functionality or reorganizing existing functionality to cope with new environmental circumstances. Therefore, we investigate the role of application environment and requirements change for architectural evolution. Our approach presented here stresses the early development stages. We introduce evolution operators that guide a systematic software architecture adjustment by establishing profitable pattern relations.

Patterns for software development have become widely accepted, especially during the last decade. They are a means for understanding a given development problem (problem frames [7]), or structuring its solution (as architectural styles [1] or design patterns [5]). Relating these patterns for different phases of the software life cycle [2, 10, 11, 14] in a methodical manner avoids to construct throw-away models. Thus, artifacts of the respective steps that are build on patterns depend on each other, explicitly.

Guiding the transfer of artifacts of the analysis phase to artifacts of software design by a pattern-based transformation procedure takes advantage of this underlying, pattern-based linkage between artifacts and its resulting traceability. Therefore, we extend an existing development process through suitable evolution operators. These operators guide the engineering process in evolving given artifacts. They also assist in the reuse of related development artifacts in successive development phases and provide help with selecting appropriate patterns. Because of its general nature, our pattern-based development method is applicable to a variety of software development problems and assists system evolution.

Section 2 describes our general development method. To illustrate our approach, we present as an example the architectural evolution of a chat system that is introduced in Section 3. Section 4 presents the evolution operators we have defined and attached to the development method of Section 2. In Sections 5 and 6, two evolution scenarios are given, describing the accomplishment of a pattern-based evolution of the original chat software architecture. Finally, Section 7 concludes this work with a brief summary of our contribution and future prospects.

2 A Pattern-Based Software Development Method

Our pattern-based software development procedure is based on Jackson's problem frames approach [7]. It consists of the following four steps:

1. Understand the problem situation

   After investigating the problem domain and identifying its current shortcomings, relevant domain knowledge is collected, and the system mission together with the corresponding requirements are recorded, as shown in Table 1. Domain knowledge (consisting of facts `F` and assumptions `A`) and requirements `R` (describing the chat system in its environment) are collected for detailing the system mission (`SM`).

   The given problem situation is structured by a context diagram, which represents the desired system, shown in Figure 1. It represents the overall problem situation. A context diagram covers the domain knowledge and the requirements of Table 1 by corresponding domains (boxes) and their interactions (*shared phenomena* at labeled interfaces). The *machine domain* (indicated by two vertical stripes) represents the software we are going to build.

Table 1. Initial Set of Requirements and Domain Knowledge for a Chat Application

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical display.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical display.
⋮	. . .
F1	Users communicate in a local network.
A1	Users follow the course of the chat on their private graphical display.

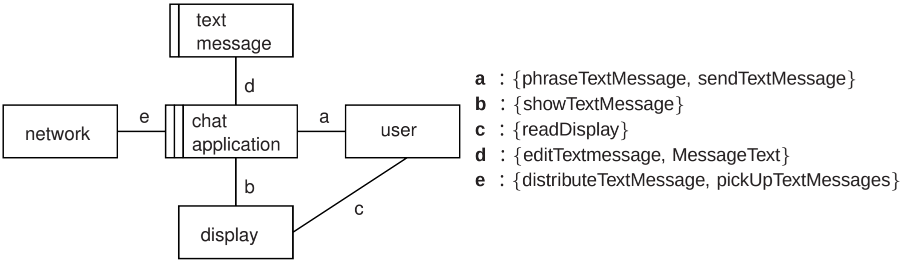

text message
d
network
e
chat application
a
user
b
c
display
a : {phraseTextMessage, sendTextMessage}
b : {showTextMessage}
c : {readDisplay}
d : {editTextmessage, MessageText}
e : {distributeTextMessage, pickUpTextMessages}

Fig. 1. Context Diagram for a Chat Application

2. Decompose overall problem into simple subproblems

   The requirements guide a knowledge-based decomposition of the overall problem that is represented by a context diagram into several simple problems. A simple problem is represented by a problem diagram, which expresses what the subproblem is about by referring to the involved domains and related shared phenomena. These subproblem representations together with the given domain knowledge suffice to derive a software *specification* describing the interface behavior of the machine. Figures 2 and 3 represent two simple subproblems for requirements R2 and R3. [1]

3. Fit subproblems to (variants of) problem frames

   When using a pattern-based development method, the subproblems are classified by instantiating suitable problem patterns, called problem frames (PF) [7]. These are patterns categorizing software development problems into problem classes during the analysis phase. Thus, each problem frame represents a problem category, which can be linked to patterns of a corresponding solution class. Via analogies, they can be related to patterns of software design, resulting in a smooth transition from requirements engineering documents into design artifacts [11].

4. Instantiate corresponding architectural and design patterns

   Finally, we make use of the *problem/solution-pattern relationship* discussed in the previous Step 3 to derive a software design appropriate for the given problem

[1] Figures 2 and 3 are explained in more detail in Section 3.

situation documented in Steps 1 and 2. As an instance of a problem frame, each derived subproblem assigns values to its related architectural styles or design patterns. As a result, we obtain a more or less coarse-grained design (see Figures 4, 6, and 7) for each subproblem. These subproblem solutions can be used as a starting point for additional solution refinement, component deployment, or coding.

Therefore, using a specific pattern in the analysis phase results in a predetermined choice of patterns in the design phase. If a subproblem fits to a problem frame (as in Figures 2 and 3), related architectural or design patterns (see Figures 4, 6, and 7) can offer a solution structure for it.

In the following, we use design patterns such as *Forwarder-Receiver* [1], which are comparable to architectural styles for developing software architectures. However, composing the overall (architectural) design out of several subproblem solutions [3] is out of scope of this paper. We consider subproblems that contribute to a common solution design in this paper.

3 Example: Developing a Chat System

The starting point of our initial software development project is the system mission in Table 1. As described in Steps 1 and 2 of our development method, domain knowledge and requirements are collected, and a context diagram is set up (see Figure 1). The problem diagrams of Figures 2 and 3 represent two of the subproblems derived for the chat application. They refer to the requirements R2 and R3.

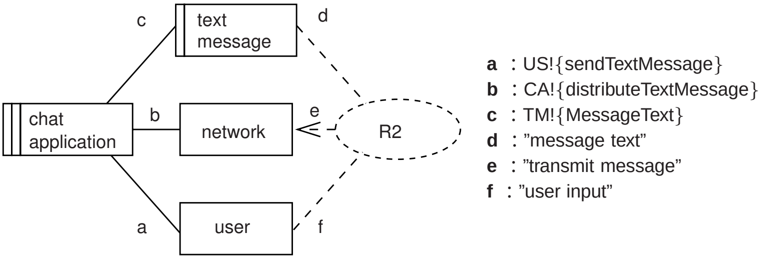


Fig. 2. ”Message Forwarder” subproblem (instance of a variant of the *commanded behaviour* PF)

If a problem requirement such as R2 and R3 can be fitted to a problem frame, corresponding domains and shared phenomena for describing it in detail become identifiable. Then, the dashed oval contains the corresponding requirements. The dashed lines demonstrate the relationships between these requirements and the different problem domains, see for example Figure 2. An arrowhead pointing at a problem domain denotes a *requirements constraint*, which stipulates the development of a machine controlling the problem domain as stated in the requirements. Then, the frame diagram supports the derivation of a *specification*, which is a technical description sufficient for developing the desired software. For this, the interfaces at the machine domain are of particular importance. They specify what services the desired software shall provide.

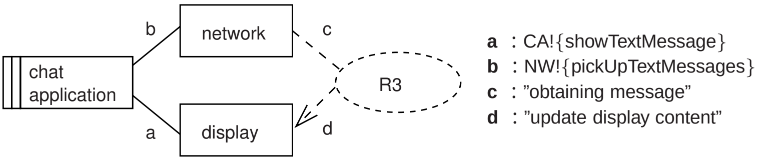


Fig. 3. "Message Receiver" subproblem (instance of *information display* PF)

In addition to the interfaces of a context diagram, in problem diagrams an abbreviation of the domain name (like US for user) is given. An exclamation mark at the labeled interfaces indicates which domain *controls* a shared phenomenon or a set of shared phenomena. An example for this is **a**: US!{sendTextMessage} in Figure 2. It means that the user initiates commands for sending chat text messages.

As recommended by Step 3 of our pattern-based software development method, both subproblems of Figures 2 and 3 are instances of problem frames [7]. In Figure 4, we illustrate how to transform a pattern for software analysis to a pattern for software design in order to accomplish Step 4.

First, we translate the subproblems into a class diagram known from Unified Modeling Language (UML) [13] (see classes chat application and network of Figure 4). This eases linking them to patterns of software design, which in general are represented in UML notation. Thus, in our approach, problem frames and problem diagrams take the role of UML use cases, which are a means for requirements elicitation and problem decomposition. In contrast to use cases, problem frames and problem diagrams refer to their respective requirements explicitly. Furthermore, they represent necessary objects and their interactions in more detail, which facilitates for a more coherent development. And in addition, they support our aim of an integrated pattern-based development procedure. Consequently, problem frames do not replace common development notations such as UML, but they extend them in a profitable way.

The upper part of the class diagram shown in Figure 4 represents the *Forwarder-Receiver* design pattern [1]. The two classes chat application and network are taken from the subproblems in Figures 2 and 3. They are related to the *Forwarder-Receiver* design pattern via analogy:

The software we are going to build, namely the chat application is related to the class Peer in Figure 4. Both have in common that they constitute the core chat system controlled by a user, who can call specific services, such as sendTextMessage. Therefore, we relate them via an inheritance relation. The network domain in Figure 2 takes the role of the class Forwarder in Figure 4, implementing its operations by distributeTextMessage. In the subproblem of Figure 3, network takes the role of a Receiver, which again is expressed by an inheritance relation in Figure 4, where network implements the Receiver operations by pickUpTextMessages. In the following, the domain text message of Figure 2 and the domain display of Figure 3 are assumed to be part of the chat application or peer. For sake of simplicity, we will not consider them any further, because they have no architectural effects in our example.

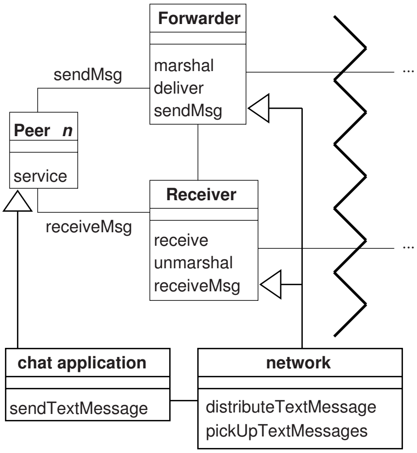


Fig. 4. Relating Problem Diagrams with Design Pattern *Forwarder-Receiver*

Figure 4 shows an initial software design, which is constructed entirely with patterns. Implementing this design results in a peer-to-peer chat system. For our example, we first finish the development process at this point.

4 Considering Evolution Through the Development Life Cycle

To support evolution, we define a corresponding evolution step for each step of the method described in Section 2. The work of O'Cinneide and Nixon [8] may seem similar to our approach. However, we do not perform refactoring to introduce design patterns into a given system. In contrast, our system is already composed of design patterns.

Our evolution method consists of several steps each providing evolution operators for the respective development phase. Changes to the original method introduced in Section 2 are indicated in bold face. The evolution operators document what the change is and how it should be carried out. This results in a corresponding set of operators for each step of our development method, e.g. the addition or deletion of a domain in the context diagram in Step 1. In the following, we concentrate on illustrating those operators which are applied to our example.

1. Understand the **new** problem situation

   Evolution takes place when a change request is present. This request has different effects, depending heavily on whether or not requirements or domain knowledge are modified. In the following, we refer to new or changed requirements as evolution requirements eR and new or additional domain knowledge as aD ($aD \equiv aF \wedge aA$). The above-mentioned modifications of requirements and/or domain knowledge make it necessary to gain an understanding of the new circumstances. This may result in a modification of the context diagram, using evolution operators. The operators relevant for this phase are:

   eAD – evolution operator *add new domain*:

   A new domain has to be added to the context diagram.

The eR and/or the aD introduce a new relevant domain. Relevant means that the domain is necessary to develop the specification for the machine. Usually, this implies that the new domain is directly connected to the machine domain. This domain has to be added to the context diagram. The new phenomena that occur have to be treated with **eAP** (*add new phenomenon*) or **eMP** (*modify existing phenomenon*) (described below).

eMD – evolution operator *modify existing domain*:

A domain contained in the context diagram has to be modified. Possible modifications are for example splitting or merging of domains.

In contrast to **eAD**, the eR and aD do not necessitate a new domain in this case. However, they make it necessary to modify a given domain in the context diagram. This may occur when the eR and/or the aD are extended or changed, resulting in a possible application of **eAP**.

eAP – evolution operator *add new phenomenon*:

A new phenomenon is added to an interface of the context diagram.

Whenever **eAD** is applied, it is also necessary to add new phenomena to the newly created interfaces between the added domain and the domains connected to it. It may also occur that a new phenomenon has to be added to an already existing interface (perhaps as a consequence of applying **eMD**).

eMP – evolution operator *modify existing phenomenon*:

An existing phenomenon has to be modified in the context diagram, e.g. by renaming.

The domains contained in the context diagram suffice to capture the new situation. The shared phenomena, however, have to be changed in order to handle the modified behavior derived from eR and/or aD.

In some cases, it may also occur that neither domains nor shared phenomena are newly introduced. Then, no changes to the context diagram are necessary in Step 1, but the new requirements/domain knowledge may require changes in later steps. A reason for this is that at this stage, only the static aspects of the system and not the dynamic aspects are taken into account. The resulting context diagram now represents the new overall problem situation.

2. Decompose overall problem into simple subproblems, **and adapt existing ones**

It is necessary to investigate the existing subproblems, applying evolution operators as necessary. The eR are the driving force behind this investigation, as they determine whether or not it is necessary to create a new subproblem or to adapt an existing one. Examples of evolution operators for this step are:

eIR – evolution operator *incorporate eR into a given subproblem*:

New domains and associated shared phenomena may be added to an existing problem diagram. This is possible if the eR references at most the same domains as the given subproblem. Conflicting requirements may occur at this point. However, resolving such conflicts will not be addressed here.

eCS – evolution operator *create new subproblem*:

Either the eR is assigned to a given subproblem, but the resulting subproblem then gets too complex. Hence, it is necessary to split the subproblem into smaller subproblems.

Or the eR cannot be assigned to a given subproblem, and a new subproblem has to be created.

For the next steps, only newly introduced and adapted subproblems have to be taken into further consideration, as only these will undergo changes. The subproblems which have not been addressed in this step can be disregarded for now. They will only become relevant again in later steps, when the solutions of the subproblems are composed to the overall solution.

3. Fit subproblems to (variants of) problem frames and **adjust problem frame instances**

   The operators for this step include:

   eFF – evolution operator *fit to a problem frame*:
   Each newly introduced subproblem is fitted into a problem frame by instantiating it according to the general procedure of Section 2.

   eCF – evolution operator *choose different problem frame*:
   For each adapted subproblem, it is checked whether its underlying problem frame is still valid, or whether another problem frame is now more appropriate. The corresponding problem frame is then instantiated accordingly.

4. **Modify** and instantiate corresponding architectural and design patterns

   Comparable to evolution Step 2 driven by eR, this step is guided by aD. Evolution operators applicable in this step are:

   eAA – evolution operator *adjust given architecture*:
   Adapted subproblems, which still fit into already instantiated problem frames, can usually be incorporated into the given architecture without difficulties.

   eCA – evolution operator *choose different architectural style or design pattern*:
   New subproblems or subproblems that fit to different problem frames than before lead to a new investigation of the solution.

   This investigation may result in a (re-)assignment of existing subproblems to new architectural styles or patterns.

   Furthermore, aD can cause a change in the *problem/solution-pattern relation*, resulting in a reallocation of subproblem elements to corresponding parts of solution patterns via new analogies. For instance, this fact distinguishes evolution scenario I from evolution scenario II (see Sections 5 and 6).

   In the subsequent sections, we illustrate the usage of these evolution operators by two evolution scenarios for our chat application.

5 Evolution Scenario I

The starting point for this first software evolution scenario is a chat system as described in Section 3 for local communication (cf. `F1` in Table 1), for example via a bluetooth device as an implementation of the network domain.

A limitation of such a chat application is that users are restricted to the range of their bluetooth devices. This limitation has to be removed now. An extended fact **`a`**`F1` about

Table 2. Changed Requirements and Domain Knowledge for Evolution Scenario I

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical displays.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical displays.
eR5	**Users want to chat via long distances. Therefore it is necessary to pass the data from the local network to the wide access network (and vice-versa).**
aF1	Users communicate in a local network**, or via a wide area access network**.
A1	Users follow the course of the chat on their private graphical display.

the application domain is introduced, see Table 2: Users communicate [...] **via a wide area access network**.

However, there is also a *constraint* restricting the evolution procedure: *The structure of the original application should be maintained*. This constraint stresses the maximal possible reuse of artifacts of the existing system. Therefore, it will be necessary to maintain the existing architecture in its original form as far as possible. We now follow the procedure described in Section 4 for evolving the given chat system:

1. Understand the new problem situation

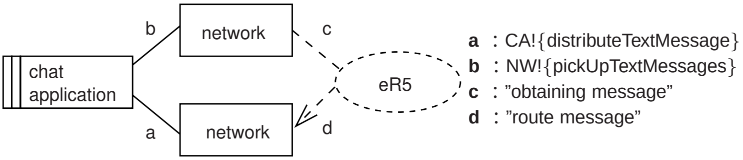
   Analyzing the change of domain knowledge results in an additional requirement, which is added to Table 2 as **eR5**.

   It is not necessary to add a new domain into the context diagram. The set of existing phenomena suffices, as well. Therefore, it is not necessary to make any changes to the existing context diagram. It still looks as shown in Figure 1.

2. Decompose overall problem into simple subproblems and adapt existing ones

   We apply the operator **eCS** (*create new subproblem*) of Section 4 and create a new subproblem. The formerly used network is not able to provide a wide area access. Taking the above constraint into consideration, as well, we obtain the following new subproblem represented in Figure 5. We need to transfer the data from our bluetooth network to another network, which will deal with passing the data to or receiving the data from the wide area access network. The other subproblems remain unchanged.

3. Fit subproblems to problem frames and adjust problem frame instances



b
network
c
chat application
eR5
a
network
d
a : CA!{distributeTextMessage}
b : NW!{pickUpTextMessages}
c : ”obtaining message”
d : ”route message”

Fig. 5. ”Message Dispatcher” subproblem (instance of a variant of the *transformation* PF)

As we have created a new subproblem, the evolution operator **eFF** (*fit to a problem frame*) described in Section 4 has to be applied. Accordingly, the new subproblem becomes an instance of a transformation problem frame variant.

4. Modify and instantiate corresponding architectural and design patterns

   By having a closer look at the new problem diagram in Figure 5, we see that what is performed by this subproblem can be characterized as a kind of dispatching. Now the evolution operators **eAA** (*adjust given architecture*) and **eCA** (*choose different architectural style or design pattern*) have to be considered. The operator **eAA** preserves the *Forwarder-Receiver* architecture for the subproblems in Figures 2 and 3. These problem diagrams stay untouched, and so does their corresponding architecture. To the newly created subproblem, we apply the operator **eCA**. As we know that we need a dispatcher, this leads us to the pattern of *Client-Dispatcher-Server* [1], because a dispatcher is responsible for establishing a (wide area) connection between two parties. Another alternative could be the design pattern *Proxy*. However, we choose the first pattern, namely *Client-Dispatcher-Server*, to illustrate the evolution in this scenario. To satisfy the accompanying evolution constraint to reuse as many development artifacts as possible, we attach the *Client-Dispatcher-Server* to the already applied *Forwarder-Receiver* pattern, resulting in a hybrid design pattern. Here, we follow in general the *pattern-oriented analysis and design* (POAD) approach [15]. As shown in Figure 6, the new solution pattern for the added subproblem in Figure 5 can therefore simply be "plugged together" on the conceptual level with the existing one in Figure 4.

   The connection between the two solution patterns namely *Forwarder-Receiver* and *Client-Dispatcher-Server* is realized through dependencies. We want to maintain the original patterns as much as possible. The class Forwarder and Client share the responsibility for sending some content or requests. Therefore, we can reuse

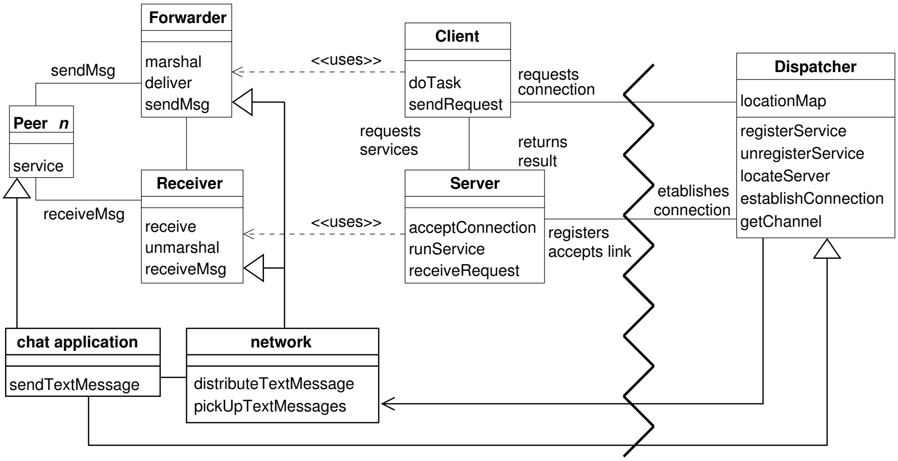


Fig. 6. Evolved Class Diagram of the Chat Application (Hybrid Style)

Forwarder for implementing sendRequest of class Client. The same holds for the class Receiver, which is responsible for realizing the Server operation receiveRequest.

The chat application and network class derived from our subproblem descriptions are related to the combined solution patterns via generalization/specialization relations. Where chat application takes the role of a peer that provides services such as sendTextMessage, and network takes the role of forwarder and receiver for handling the reception of messages by pickUpTextMessage and their delivery through distributeTextMessage. In its role as a Dispatcher, chat application uses its reference to network for controlling its message handling, respectively.

Interesting is the role of the network, because it is part of all three subproblems. For the subproblems in Figures 2 and 3, the network is responsible for providing the Forwarder/Receiver functionality. For the subproblem in Figure 5, it implements Dispatcher operations. For the *Dispatcher* the network connects the different chat peers via long distances, whereas each peer is a *Client* as well as a *Server* communicating via a *Forwarder-Receiver* mechanism.

It is clearly visible that in this first evolution scenario the modification of **aF1** and the addition of **eR5** resulted in the creation of a new subproblem. Because of this new subproblem, it was necessary to make a new design decision. Considering the given constraint, the decision leads us to an extension of the existing *Forwarder-Receiver* architecture to a hybrid *Forwarder-Receiver/Client-Dispatcher-Server* style.

6 Evolution Scenario II

The second evolution scenario is based on the results of Section 5. With the current chat application it is possible to communicate with other users via bluetooth or a network providing wide area access. The devices used so far are general purpose computers. Mobile devices such as portable phones or personal digital assistants (PDA) are not supported yet. This describes the limitation that we remove in this second evolution scenario: The usage of portable devices should be possible, as well. Once more the evolution steps described in Section 4 are applied:

1. Understand the new problem situation

   New domain knowledge is added, described by the additional fact **aF2** in Table 3: **The devices used are general purpose computers as well as portable phones and PDAs**. This new fact describes hardware constraints referring to the machine domain. No additional requirements are necessary. Therefore, the context diagram remains unchanged, even though new domain knowledge has been introduced. The reason is that **aF2** influences internal characteristics of the machine, and not its behavior. These characteristics, however, cannot be described by a context diagram.

2. Decompose overall problem into simple subproblems and adapt existing ones

   The distribution of the overall problem situation into subproblems stays unchanged, because the requirements do not change.

Table 3. Changed Requirements and Domain Knowledge for Evolution Scenario II

SM	*A text-message-based communication platform shall be developed, which allows multi-user communication via private I/O-devices.*
R1	Users can phrase text messages, which are shown on their private graphical displays.
R2	Users send their phrased text messages to participate in the chat.
R3	Sending text messages changes the chat represented on the users' graphical displays.
eR5	Users want to chat via long distances. Therefore it is necessary to pass the data from the local network to the wide access network (and vice-versa).
aF1	Users communicate in a local network, or via a wide area access network.
aF2	**The devices used are general purpose computers as well as portable phones and personal digital assistants (PDAs).**
A1	Users follow the course of the chat on their private graphical display.

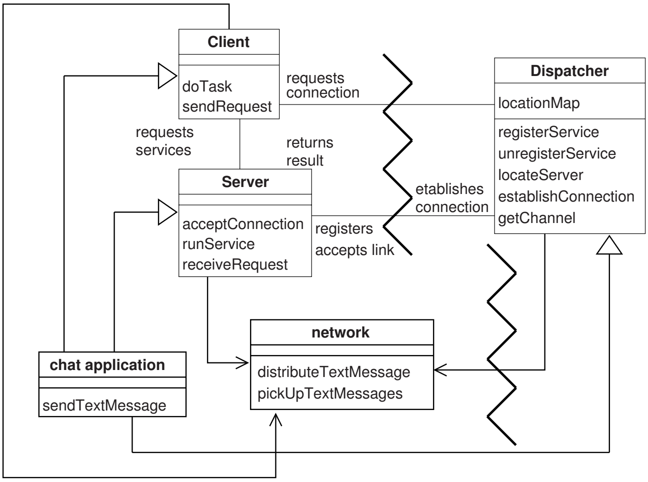


Fig. 7. Resulting architectural design based on *Client-Dispatcher-Server*

3. Fit subproblems to problem frames, and adjust problem frames instances

   No changes have to be performed in this step, because no changes were performed in the previous step.

4. Modify and instantiate corresponding architectural and design patterns

   This step is the nontrivial one in this scenario, because here the effect of the newly introduced domain knowledge becomes visible. Fact **aF2** influences the decision which pattern to select in the solution space. Mobile devices such as PDAs or mobile phones do not possess the same resources as general purpose computers do. The new fact thus imposes a constraint on the selection of design patterns and architectural styles. Here, it leads to a re-design of the present architecture by means of the evolution operator **eCA**.

> The new domain knowledge enforces a reorganization of the given subproblems, resulting in a different choice of architectural style or design pattern out of the related solution class. For our example, the three subproblems are matched with the *Client-Dispatcher-Server* pattern only (cf. Figure 7).
>
> The former forwarder and receiver components are merged with the client and server components. The chat application now consists of two parts namely, a server and a client part. The functionalities formerly represented by the peer are now partially realized by the server and client classes, respectively.

In the first evolution scenario, we had to deal with a constraint. This constraint is still present in the second scenario, however, in a weakened form: all the subproblems not involved in the data transmission/reception remain unchanged, and the development documents related to them can be reused as is. In this second scenario, adding domain knowledge in form of **aF2** results in a complete restructuring of the architecture. The reason is that the new domain knowledge leads to a different matching of subproblems to architectural patterns, resulting in a new, more appropriate and simpler architecture.

7 Conclusion and Future Work

We have introduced a pattern-based development method, incorporating evolution in each step, and driven by evolution operators. With this method, it is possible to perform software evolution systematically whenever new requirements or changes in the application environment occur.

Through a chat application example, we have shown the usage of our method. We illustrated how a system evolved from a straight peer-to-peer architecture via a hybrid architecture to a client-dispatcher-server system. This evolution is achieved by applying evolution operators. They allow for identifying those development documents, which have to undergo change. Thus, they provide guidance for performing the necessary modifications in phases to come.

This approach enables us to perform a systematic rework on the affected development documents by means of patterns. We have shown that it is possible to link the artifacts of the analysis phase to the artifacts of the design phase. Therefore, changes in the analysis help to perform an analogous procedure in the design phase.

Our method does not intend to replace existing methods or notations. It rather extends them by a goal-oriented, pattern-based approach resulting in coherent and precise specifications. The method does not postulate a dogmatic approach, always resulting in exactly one unique solution. We intend to constrain the possible solutions (design space) to provide a small, most promising set of patterns useful for solving the problem.

Also non-functional requirements can be treated with our approach: If these quality aspects are specified in the problem frames (such as *HCI*Frames [14] or using Security Problem Frames [6]), they will lead to solutions that address these non-functional issues, see evolution scenario one in Section 5. Additionally, non-functional aspects that are manifested in domain knowledge can be covered, see evolution scenario two in Section 6.

In summary, the advantages of our approach are the following:

- Our method is pattern-to-pattern, integrating evolution. Hence, it has all assets that come with the use of patterns, in particular, reuse of established analysis and design knowledge.
- Our evolution operators provide guidance concerning pattern selection and transformation. They help to adapt the development problem and to find new solutions if necessary.
- Non-functional requirements can be treated, as well.

Currently, we are working on a formalization of the problem frames. For that purpose, we are building a formal metamodel using the formal specification language Object-Z [12]. The graphical representation of the formalization is given through UML class diagrams [13]. The metamodel is equipped with integrity conditions to ensure the validity of a frame with respect to the metamodel. It is planned to create an Eclipse [4] plug-in for this metamodel. The plug-in will work as an editor to create valid frame diagrams in the context of our metamodel. The metamodel will also serve as a basis for investigating which of the evolution operators can be formalized and incorporated into the metamodel. In a further step, the plug-in will be extended to allow for an automation of the identified and formalized evolution operators presented here.

In the future, we also plan to analyze the effects of domain knowledge within the evolution process in more depth. Additionally, we plan to examine the impact of the change in the architectural structures to later documents of the software life cycle, especially considering the source code. Furthermore, we intend to put stronger emphasis on distinguishing internal and external quality aspects and how they are successfully covered by our method. It is also planed to have a detailed look at the decompositions and compositions of the subproblems with respect to the architecture, which would also contribute to investigate the scalability of problem frames. We further will investigate which problem frames should be related to which architectural design patterns or architectural styles for completing our method.

References

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester (1996)

[2] Choppy, C., Hatebur, D., Heisel, M.: Architectural Patterns for Problem Frames. IEE Proceedings - Software 152(4), 198–208 (2005)

[3] Choppy, C., Hatebur, D., Heisel, M.: Component composition through architectural patterns for problem frames. In: Proc. XIII Asia Pacific Software Engineering Conference, pp. 27–34. IEEE Computer Society Press, Los Alamitos (2006)

[4] Foundation, T.E.: Eclipse - an open development platform (2007), `http://www.eclipse.org`

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

[6] Hatebur, D., Heisel, M., Schmidt, H.: Security engineering using problem frames. In: Müller, G. (ed.) ETRICS 2006. LNCS, vol. 3995, pp. 238–253. Springer, Heidelberg (2006)

[7] Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)

[8] O'Cinneide, M., Nixon, P.: Automated Software Evolution Towards Design Patterns (2001), http://citeseer.ist.psu.edu/671812.html

[9] Pfleeger, S.L.: Software Engineering: Theory and Practice. Prentice-Hall, Englewood Cliffs (2001)

[10] Rapanotti, L., Hall, J.G., Jackson, M.A., Nuseibeh, B.: Architecture-driven Problem Decomposition. In: RE'04. Proceedings of the 12th IEEE International Requirements Engineering Conference, Kyoto, Japan, IEEE Computer Society Press, Los Alamitos (2004)

[11] Schmidt, H., Wentzlaff, I.: Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, Springer, Heidelberg (2006)

[12] Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, Dordrecht (2000)

[13] UML Revision Task Force: OMG Unified Modeling Language: Superstructure (2007), http://www.omg.org

[14] Wentzlaff, I., Specker, M.: Pattern-based Development of User-Friendly Web Applications. In: ICWE. Workshop Proceedings of the 6th International Conference on Web Engineering, ACM Press, New York (2006)

[15] Yacoub, S.M., Ammar, H.H.: Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems. Addison-Wesley, Reading (2003)

Formal Design of Structural and Dynamic Features of Publish/Subscribe Architectural Styles

Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira

University of Sfax
Research Unit ReDCAD
B.P. W.3038 Sfax, Tunisia
Imen.loulou@tunet.tn, Ahmed@fsegs.rnu.tn, Mohamed.Jmaiel@enis.rnu.tn
LAAS-CNRS
7 avenue de Colonel Roche
31007 Toulouse Cedex 4, France
Khalil@laas.fr

Abstract. This paper proposes a compositional formal approach to design correct publish/subscribe architecture styles. We provide a set of patterns and the corresponding composition rules to build architecture styles. The defined patterns and rules respect the principle of information propagation requiring that produced information have to reach all the subscribed consumers. We describe patterns as graphs and we use the Z notation to specify formally the semantic of each pattern and each rule. We prove consistency and correctness using the Z-Eves theorem prover. We show how to consider the interconnection topology between dispatchers as well as the subscription mechanism by simple refinements. We also show how to construct the Z specification of the designed architecture style based on applied rules. Moreover, we describe the dynamics of architecture via guarded graph-rewriting rules whose body describe the structural constraints and whose guards mainly describe the functional constraints of the system. We express these rules entirely with the Z notation also, obtaining a unified approach which handles both the static and the dynamic aspects.

Keywords: software architecture, publish/subscribe style, style composition, formal specification, architecture modeling, dynamic architecture, graph rewriting.

1 Introduction

An increasingly common architectural style for component-based systems is publish-subscribe. The strength of this event-based interaction style lies in the full decoupling in time, space, and synchronization between the components that have generated events, called producers, and the receivers, called consumers. This decoupling is due to the event-service which is the glue that ties together distributed components in Publish/Subscribe architecture. The event service could

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 44–59, 2007.

be centralized and implemented as a single entity called dispatcher or distributed and implemented as a network of dispatchers. This makes the publish-subscribe style relatively easy to add or remove components in a system, introduce new events, register new consumers on existing events, or modify the network of dispatchers.

The gain in flexibility is counterbalanced by the difficulty to the designer in preserving the correctness and the reliability of a designed publish-subscribe style. In fact, the design phase has to consider the principle of information propagation requiring that produced information reaches all the subscribed consumers. Another major issue is to preserve the system consistency during reconfiguration. The system must be left in a [illegible] state after reconfiguration, by maintaining the conformity of its new architecture with respect to a set of structural properties or architecture style. The complexity of designing and building these systems could be reduced by focusing on the architectural level and the ambiguity could be eliminated by using formal descriptions. Indeed, by abstracting away from implementation details, a formal architectural description makes a system design intellectually tractable and exposes the properties most crucial to its success. It is often the key technical document used to determine whether a proposed new system will meet its most critical requirements.

This paper makes a contribution in these directions by proposing a visual and formal approach which aims to help the architect to design correct publish/subscribe architecture styles. Instead of designing an architecture style from the scratch, we rather provide a reusable set of simple patterns and the corresponding composition rules to build complex event-based architecture styles. In addition, we show how to consider the interconnection topology of a distributed event service as well as the subscription mechanism by simple refinements. To validate and reason about the architecture styles designed from the building blocks, we propose the use of the formal method Z notation [1]. We elaborate a formal specification for each basic pattern, and we show how each visual operation of composition is specified in Z. We show also how we construct the Z specification of the designed architectural style using the formal specification of each operation applied by the architect and how to refine it. We use the Z-Eves theorem prover to check and reason about our specifications.

We propose an approach which benefits from the expressive power of functional notations and graph grammars to describe and to reason about architectures. We describe reconfiguration operations via a new notation based on graph rewriting rules and Z notation. This integration offers to software architects an intuitive and formal way to specify the dynamic aspect of software architecture based on visual notations. We propose also to specify the semantic of each reconfiguring operation in the same notation ( ), thus obtaining a unified approach. These rules take into account both structural and functional constraints of a system in their application conditions ensuring, in this way, its consistency during its evolution. We use Z-Eves theorem prover to check that all reconfiguring operations respect the defined architecture style.

The remainder of this paper is organized as follows: in section 2, we present a survey of the related work. In section 3, we present the main features of a publish/subscribe architecture style. In section 4, we describe our methodology to design correct styles. We also explain how to refine a designed style to consider the interconnection topology and the subscription mechanism. Section 5 presents our approach to specify reconfiguration and we explain the verification process. Section 6 presents a case study which illustrates our approach. In section 7, we give our concluding remarks and the future work perspectives.

2 Related Work

This survey will focus on research done in the area of software architecture description, on one hand. On the other hand, it will concentrate on some works that address the static as well as the dynamic aspect of publish/subscribe systems.

Considerable research has been done on the description of the static as well as the dynamic aspect of software architecture. We can classify together works describing the system architecture like Architecture Description Languages (ADLs). The most prominent among them are Rapid [2], π -Method [3], Wright [4], and ACME [5]. They provide modelling tools which help the designers to structure a system and to compose its elements. Generally, ADLs allows to describe only predefined dynamics. That is, they are interested only in systems having a finite number of configurations which must be known in advance. In addition, few of them [3,4,5] allow various architectural styles to be distinguished. It should be stressed that we don't propose an alternative to ADLs approaches. Our models can be integrated into π -Method (π -ADL) or into other ADLs. Functional languages have been also proposed. They introduce abstract notations allowing to describe dynamic software architectures in terms of properties. In [6], the authors developed a formal framework, specified in Z, to describe the dynamic configuration of software architectures. They did not address the design phase. Other works use graph grammars techniques. Graph grammars consist in using graphs for representing software architectures. They represent the most intuitive mathematical tool for modelling complex situations. In this context, [7] describes the software architecture style using a context-free graph grammar and verifies the conformity of an architecture to its style. In this work, the author addresses the client-server problem. Reconfiguration is described with a set of rewriting rules whose definition is rather simple and comprehensible. These rules explain clearly the topological changes, but they don't allow to express post-conditions of an operation to describe what must be ensured after its application, or to specify certain logical conditions which permit, for example, to reason about the instance number of a given component.

Past research addresses implementing efforts for publish/subscribe systems with scalable, distributed architectures. Several infrastructures were proposed such as SIENA [8]and JEDI [9]. Foundational research has also addressed formal reasoning for publish/subscribe systems using model checking ([10,11]). Other approaches include formal computational models for the event-based paradigm like [12] where authors define a computational model to capture connectivity in the communication topology of P2P systems. The work which is most closely related to our work is described in [10]. Authors introduce an approach to model and validate publish/subscribe systems at the architectural level. The modeling phase specifies how components react to events. However, authors consider neither the communication between dispatchers nor the interconnection topology. So, we can't reason about the event propagation inside an event service. In addition, they don't consider the subscription mechanism which is the way subscribers can express the events they are interested in. This feature allows to precise which events are produced or consumed by which components.

Recent research addresses the dynamic reconfiguration of Publish/Subscribe systems. However, the major issues which have been addressed are about notification and (un)subscription loss as well as duplication during reconfiguration [13,14]. In addition, most of them consider reconfiguration by repeatedly exchanging a single link in the network of dispatchers. In [13] for instance, presented algorithms prevent the loss and duplication of notifications and maintain the message ordering. In [14], authors present a description and analysis of a novel algorithm to deal with the reconfiguration of the dispatching infrastructure.

Our work will help the architect to design correct and elaborated publish/subscribe styles by simply composing the necessary provided patterns. Our formalisation allowed us to characterise key structural properties of the publish/subscribe architectural style formally and at a high level of abstraction. We address also the dynamic aspect by offering a new notation which is sufficiently expressive and easy to understand. To ensure that the system is evolving correctly, we elaborate a verification process which validates each rule, whose semantic is described in Z notation, using the Z-EVES theorem prover.

3 Publish/Subscribe Architecture Style

Publish/Subscribe architectures could use a centralized event service, implemented as a single entity called dispatcher, or a distributed service implemented as a network of dispatchers that provide access points to clients. Dispatchers cooperate together to route information from the producers to the subscribed entities. This interaction is governed by a principle of information propagation requiring that produced information have to reach all subscribed consumers.

In addition, several interconnection topologies have been proposed: hierarchical, acyclic peer-to-peer, and general peer-to-peer [8]. The hierarchical topology is an extension of the centralized approach in which a set of interconnected dispatchers maintain a master/client relationship with other dispatchers. In the acyclic peer-to-peer topology, dispatchers communicate with each other symmetrically as peers, adopting a protocol that allows a bidirectional flow of subscriptions, advertisements, and notifications. In this topology, any two dispatchers are connected with at most one path. Removing the constraint of acyclicity from the acyclic peer-to-peer topology, we obtain the general peer-to-peer topology.

Another characteristic is worth a particular attention: the way subscribers can express the events they are interested in (usually called subscription schema). The existing literature distinguishes among topic-based and content-based systems[15,9]. The topic-based subscription schema is based on the notion of [illegible] or [illegible]. Publishers generate events belonging to one or more topics, while subscribers express interest in some of these topics and will therefore receive notifications about events published in that particular topic. The content-based publish/subscribe variant improves on topics by introducing a subscription schema based on the actual content of the considered events.

4 A Methodology to Design Correct Pub-Sub Styles

In this section, we show how to identify the generic building patterns which can be composed to design a correct publish/subscribe style. By correctness, we mean two properties:

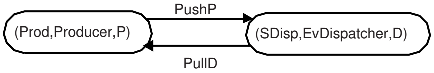
- Produced information have to reach all the subscribed consumers;
- The interaction schemas must be in conformance with the communication rules.

The interactions between the components and the event channels are specified for architectures that use a centralized or a distributed event service. The interaction schemas must be in conformance with the communication rules. By ensuring this property, we ensure the propagation of events between producers and dispatchers, on the one hand and between dispatchers and consumers one the other hand. The propagation of events from producers to interested consumers is completely ensured if we guarantee the routing inside an event service which will be done later.

The communication between producers and event dispatchers can be established through the push mode if the producer is the initiator. The pull mode is used for interaction when the event dispatcher is the initiator of communication. The communication between consumers and event dispatchers can be established through the pull mode when the consumer is the initiator of communication. The push mode is used when the event dispatcher is the initiator of communication.

To characterize the generic patterns, our idea is to consider on the one hand the number of interaction links between the different components and the different event dispatchers. the first case corresponds to the situations where one link is considered between each pair of interacting entities. The second case corresponds to the situations where two links are considered. On the other hand, we distinguish the different modes of interaction. This allows us to characterize for each of the previous two cases, the different patterns of corresponding architectures.

For example, if we consider the communication between producers and dispatchers, the second case where two links of communication are considered,



$$\begin{array}{|l}
\_\, C\_P3[Producer, EvDispatcher] \_\_\_\_ \\
P : \mathbb{F}\, Producer \\
D : EvDispatcher \\
PushP : Producer \leftrightarrow EvDispatcher \\
PullD : EvDispatcher \leftrightarrow Producer \\
\hline
\forall\, p : P \bullet (p, D) \in PushP \Rightarrow (D, p) \in PullD \\
\forall\, p : P \bullet (D, p) \in PullD \Rightarrow (p, D) \in PushP \\
\hline
\end{array}$$

Fig. 1. A communication pattern between producers and a single dispatcher

involves one pattern (3) in which producers may actively produce events towards event dispatchers using the push mode and event dispatchers may actively pull information from producers using the pull mode. This pattern is depicted by figure 1. A node is described with three labels: . So, a node labeled (, ,) denotes that is a set of producer components typed . The type and variable names will be instantiated by the architect while the role is kept unchanged. Formally, this pattern is specified with the schema 3[,]. Since we are not interested now in the specification details of the components (producers and dispatchers), we introduce them as basic types as follows: [*Producer*, *EvDispatcher*].

In the declaration part of the schema 3, we declare a set of producing entities ( ), and a single event dispatcher ( ). All communication links are also declared. We specify them as relations ( ). In the predicate part, we precise the following: given a producer and an event dispatcher , then they are connected via the two links and or they are not connected.

In addition, we take into consideration the fact that an entity can behave as both a producer and a consumer (P/C). To characterize the corresponding generic patterns, we have to offer at least two links between a P/C entity and an event dispatcher: One link for the producer communication and one link for the consumer communication. So, we characterize other design patterns.

However, architectures exploiting a distributed event service need in addition the details of the interactions between dispatchers. This interaction is governed by the principle of information propagation. To guarantee this property in the case of a distributed event service, we make two constraints:

- : all event dispatchers have to be interconnected.
- : if two dispatchers communicate together then it is necessary that the information coming from the one reaches the other. So, the communication has to be bidirectional.

For example, the formal specification of the pattern involved by the second case where producers may actively produce events towards event dispatchers using the push mode and event dispatchers may actively pull information from producers using the pull mode is described with the schema 3 where the two last predicates describe the constraints 1 and 2. We have to notice also that event dispatchers communicate together using the push mode to propagate events.

$$
\begin{array}{l}
\underline{D\_P3\,[Producer, EvDispatcher]}\\
P : \mathbb{F}\, Producer\\
D : \mathbb{F}\, EvDispatcher\\
PushP : Producer \leftrightarrow EvDispatcher\\
PullD : EvDispatcher \leftrightarrow Producer\\
PushDD : EvDispatcher \leftrightarrow EvDispatcher\\
\hline
\forall\, p : P;\ d : D\\
\quad \bullet\ (p, d) \in PushP \Rightarrow (d, p) \in PullD \ \wedge\ (d, p) \in PullD \Rightarrow (p, d) \in PushP\\
\forall\, x, y : D; i : \mathbb{N}\ \mid x \neq y\\
\quad \bullet\ \exists\, T : \mathrm{seq}\, EvDispatcher\\
\qquad \bullet\ i + 1 \leq \#T\ \ \wedge i \geq 1\ \ \wedge \mathrm{ran}\, T \subseteq D\\
\qquad \wedge\, (Ti, T(i+1)) \in PushDD\ \wedge\ T1 = x\ \wedge\ T(\#T) = y\\
\forall\, x, y : D\\
\quad \bullet\ (x, y) \in PushDD \Rightarrow (y, x) \in PushDD
\end{array}
$$

4.1 How to Compose a Style?

In this section, we define the allowed operations of composition to design a correct publish/subscribe style and we show how to construct the formal specification of the designed style based on pattern specifications and according to the applied operations of composition.

- [illegible] : When the architect adds a pattern, its corresponding schema name is included in the declaration part of the style schema as follows:

  $$
  \begin{array}{l}
  \underline{Style}\\
  pattern\_name
  \end{array}
  $$

- [illegible] : The architect can rename the node labels corresponding to the formal parameters and the declaration variables. The specification is accordingly modified as follows:

  $$
  \begin{array}{l}
  \underline{Style}\\
  pattern\_name[Par_1, ..., Par_n][New/Old]
  \end{array}
  $$

  The first part corresponds to the supplied formal parameters which are optional. In the second part, we use the renaming option in Z notation where the new names of variables are given by the architect.
- [illegible] : Given two graphs, we can merge their dispatcher nodes if both of them correspond to a single dispatcher or a set of dispatchers. In addition, we distinguish two cases :
  - If we have two separate graphs, the merging is possible without any constraint.

  - If these graphs have a common node (Producer or consumer or Prod/Cons), then this node must communicate in the same manner with the two dispatcher nodes, otherwise it has no sense to merge them.

  When the merging is applied, this is described by renaming the variable ([illegible] or $\mathbb{F}$ [illegible]) so that formally the two graphs have the same variable referring to the same node.

- [illegible]: Given two graphs, we can merge their producer nodes (resp. Cons, Prod/Cons) in two cases:
  - If the nodes in question are connected to the same event dispatcher node, then they must be connected with the same types of links.
  - If the nodes in question are not connected to the same event dispatcher node, the merging is possible regardless of the types of links.

4.2 Interconnection Topologies

We provide to the architect the possibility to refine the designed styles in order to consider the interconnection topology underlying dispatchers. For this purpose, we propose to specify the hierarchical, acyclic peer-to-peer, and general peer-to-peer topology in Z notation, so that, the interested architect can choose the desired topology and the formal specification will be refined systematically by adding the schema name of the topology in the declaration part of the style schema. For example, the Z specification of acyclic P2P is as follows:

$$
\begin{array}{|l}
\hline
\mathit{acyclic\_P2P}\ [\mathit{EvDispatcher}] \\
D : \mathbb{F}\, \mathit{EvDispatcher} \\
\mathit{PushDD} : \mathit{EvDispatcher} \leftrightarrow \mathit{EvDispatcher} \\
\hline
\forall\, x, y : D;\ H, T : \mathrm{seq}\, \mathit{EvDispatcher};\ i : N \\
\bullet\, 1 + i \leq \#H \wedge 1 + i \leq \#T \wedge i \geq 1 \\
\wedge (Ti, T(i+1)) \in \mathit{PushDD} \wedge (Hi, H(i+1)) \in \mathit{PushDD} \\
\wedge T1 = x \wedge T(\#T) = y \wedge H1 = x \wedge H(\#H) = y \\
\Rightarrow T = H \\
\hline
\end{array}
$$

4.3 Subscription Mechanism

We will show in this section, how to refine a composed style to consider the underlying subscription mechanism. This characteristic allows the architect to define and check which events are produced (consumed) to (by) which components. We propose to refine the specifications of communicating entities and the event dispatchers, so that they are not considered anymore like basic types. Producers have to maintain the list of topics on which they want to publish events, the consumers have to maintain the list of topics they are interested in, the producer/consumer components have to keep the two lists and the event dispatcher has to keep the list of supported topics. So, the resulting specification of a consumer component for example is as follows:

```
┌─ Consumer ───────────────────────────────
│ topic_cons : 𝔽 TOPIC
└──────────────────────────────────────────
```

We will illustrate in section 6 how much it is interesting to specify the subscription mechanism and its impact on the system understanding .

4.4 Proving Consistency

A Z schema may be inconsistent, i.e. it includes a not satisfiable predicate or contradiction between predicates. If such a schema is elaborated to describe the state of a system, then that system could not be realized. Suppose that [illegible] describes the state of the system, and that [illegible] characterizes the initial state. If we can prove that

$$\exists \; [illegible] \bullet [illegible]$$

then this would show that an initial state exists, and hence also that the requirements upon the state components are consistent. This result is called the initialization theorem.

5 The Dynamic Aspect

The four fundamental reconfiguration operations, provided by almost all languages and systems, are creation and removal of components and connections [16]. Graph grammars showed their relevance to express these operations of reconfiguration in particular in the works described in [16] and [7]. A graph production rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones and preserving others. A graph rewriting rule expresses the structural constraints (existence/absence of a motif) via four sections:

- [illegible] : the fragment of the graph removed during the rewrite.
- [illegible] : the fragment that is created and embedded into the graph during the rewrite.
- [illegible] : the fragment that is identified but not changed during the rewrite.
- [illegible] : the fragment of the graph that must not exist for the rewrite to occur. If the subgraph matching the context and retraction can be extended to match the restriction, then the production cannot be applied to that subgraph

In publish/subscribe architecture styles, we need also to specify constraints on some attributes values related to the subscription mechanism. Furthermore, we need to define the post-conditions of an operation to describe what must be ensured after the application of an operation. Indeed, any operation defined to reconfigure the network of dispatchers, for instance, has to ensure at least the constraints [illegible] 1 and [illegible] 2 (described in section 3) after its application. In addition, sometimes it is easier to express some structural restrictions functionally (called Negative Application Condition (NAC) in the sequel) rather than visually. So, we propose a new notation which contains a structural part following the graph grammar notations, and an analytic part specified in Z notation. Given a rule [illegible], it is applied to a graph [illegible] by the following steps:

1. A subgraph isomorphic to [illegible]$_l$ = [illegible] $\cup$ [illegible] is identified in [illegible].
2. If no isomorphic subgraph exists or if the isomorphic subgraph can be extended to match [illegible]$_l$ plus the restriction (if one exists), [illegible] cannot be applied to [illegible].
3. The [illegible], the Application condition and the post-conditions parts are evaluated. They must be satisfied otherwise [illegible] cannot be applied.
4. The elements of the subgraph isomorphic to the retraction are removed from [illegible], leaving the host graph.
5. A graph isomorphic to the insertion is embedded into the host graph by the edges between the context and the insertion in [illegible].

5.1 Formal Semantic

Besides the fact that our notation fills the identified limits, the partial introduction of the formal description constitutes a preparation for a total formal semantic translation in a purpose of analysis and verification. So, we propose to describe the semantic of each reconfiguring operation via a schema operation Z as follows:

```
Rule_Name
  Δsystem_name
  in1?;in2?;...;inn?
  ─────────────────
  NAC + Restriction :
      structural and functional constraints
  Application conditions :
      functional constraints
      structural constraints
  Post − conditions
  Reconfiguration :
      (Insertion / Retraction)
```

Where:

- [illegible]$_i$? is an input parameter of the operation,
- Δ[illegible] indicates that the rule will change the system state.
- The functional constraints are about the attributes values, the number of component instances, etc.

5.2 Correctness Preservation During Evolution

In order to verify that a given reconfiguring operation makes the system evolve correctly, we propose to use the precondition theorem provided in Z notation. This theorem can identify the necessary pre- and post-conditions for an operation to preserve the properties defined by the architecture style. This theorem is described as follows:

Theorem. [illegible]
∀ [illegible] ? : [illegible] | [illegible] ([illegible] , [illegible] ?) • pre [illegible]

Where:

- [illegible] is the schema name of the designed style,
- [illegible] ? the input parameters of the operation,
- [illegible] ([illegible] , [illegible] ?) are the preconditions upon state variables and input parameters which must be satisfied beforehand,
- [illegible] is the schema operation name,
- and pre [illegible] characterizes the collection of before states and inputs for which some after state can be shown to exist.

So, the theorem allows to verify the following: what ever could be the system state, with the input parameters given in the schema operation and the defined pre- and post-conditions, a [illegible] can be shown to exist.

The first step in the verification process is to prove this theorem for each reconfiguring operation. Once proved, we can check if the preconditions identified by this theorem and those described in the schema operation are the same. If they tally, then we can say that the operation is valid.

Furthermore, we propose to use the same theorem to correct invalid operations. Indeed, we can complete or replace the pre- and post-conditions described in the schema operation by those which are identified by the theorem.

6 Case Study

To illustrate our approach, we choose a case study in the context of emergency operation activities occurring during a crisis situation [17]. The application involves structured groups of robots or military personnel which cooperate for the realization of a common mission. The activity is divided into two successive steps: an exploration step of the investigating field and an action step following the discovery of a critical situation. A team of emergency intervention is constituted of participants having different jobs: a controller and several coordinators and investigators. In the exploration step, investigators provide continuous descriptive feedbacks ([illegible]) and periodical Produced feedbacks ([illegible]), expressing the analysis of the situation, to their responsible coordinators. The coordinators complete the received analysis to send them ([illegible]) thereafter to the controller. Having discovered a critical situation, the concerned investigator keeps the same functions as in step 1 but provides feedbacks [illegible] to the other investigators. Now, the other investigators report only feedbacks [illegible] to controllers on the basis of feedbacks [illegible] transmitted by the investigator who has discovered the critical situation. The propagation of events between these participants is mediated by a network of event dispatchers called managers. Both investigators and coordinators are associated with producer/consumer components and the controller is associated with only a consumer entity.

6.1 Style Design

To design the corresponding architecture style, we apply the rule [illegible] to the pattern [illegible] 2 and to [illegible] 1 twice. Then, we apply the operation [illegible] to supply the desired parameters and to rename some declaration variables. After that, we apply the rule [illegible] twice to obtain the managers node [illegible]. So the resulting graph is depicted in figure 2. The formal specification is obtained on the basis of applied rules. To explicit the declaration and the predicate part, we propose to expand the obtained schema as follows:

$$[\mathit{Controller}, \mathit{Investigator}, \mathit{Coordinator}, \mathit{Manager}]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Emergency}} \\ \hline
\mathit{Cont} : \mathbb{F}\, \mathit{Controller} \\
M : \mathbb{F}\, \mathit{Manager} \\
\mathit{Inv} : \mathbb{F}\, \mathit{Investigator} \\
\mathit{Coord} : \mathbb{F}\, \mathit{Coordinator} \\
\mathit{Push\_M} : \mathit{Manager} \leftrightarrow \mathit{Controller} \\
\mathit{Push\_IM} : \mathit{Investigator} \leftrightarrow \mathit{Manager} \\
\mathit{Push\_MI} : \mathit{Manager} \leftrightarrow \mathit{Investigator} \\
\mathit{Push\_MC} : \mathit{Manager} \leftrightarrow \mathit{Coordinator} \\
\mathit{Push\_CM} : \mathit{Coordinator} \leftrightarrow \mathit{Manager} \\
\mathit{PushDD} : \mathit{Manager} \leftrightarrow \mathit{Manager} \\ \hline
\forall\, c : \mathit{Cont}; m : M \;\bullet\; (m, c) \in \mathit{Push\_M} \vee (m, c) \notin \mathit{Push\_M} \\
\forall\, i : \mathit{Inv}; m : M \;\bullet\; (i, m) \in \mathit{Push\_IM} \Rightarrow (m, i) \in \mathit{Push\_MI} \\
\forall\, i : \mathit{Inv}; m : M \;\bullet\; (m, i) \in \mathit{Push\_MI} \Rightarrow (i, m) \in \mathit{Push\_IM} \\
\forall\, cr : \mathit{Coord}; m : M \;\bullet\; (cr, m) \in \mathit{Push\_CM} \Rightarrow (m, cr) \in \mathit{Push\_MC} \\
\forall\, cr : \mathit{Coord}; m : M \;\bullet\; (m, cr) \in \mathit{Push\_MC} \Rightarrow (cr, m) \in \mathit{Push\_CM} \\
\forall\, m1, m2 : M \;\bullet\; (m1, m2) \in \mathit{PushDD} \Rightarrow (m2, m1) \in \mathit{PushDD} \\
\forall\, x, y : M; i : \mathbb{N} \mid x \neq y \\
\qquad \bullet\; \exists\, T : \mathrm{seq}\, \mathit{EvDispatcher} \\
\qquad\quad \bullet\; i + 1 \leq \# T \;\wedge\; i \geq 1 \;\wedge\; \mathrm{ran}\, T \subseteq M \\
\qquad\quad \wedge\; (T i, T(i + 1)) \in \mathit{PushDD} \\
\qquad\quad \wedge\; T 1 = x \;\wedge\; T(\# T) = y \\ \hline
\end{array}
$$

It should be noticed that architecturally, one can understand that the controller can consume the events coming from both investigators and coordinators, what is not in conformity with the description of the application. So, the architect

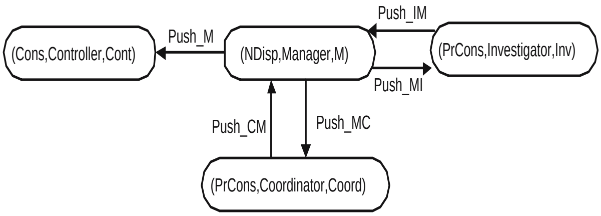


Fig. 2. Emergency Operations scenario

may decide to refine his style to describe the subscription mechanism, let's say topic-based, to specify which information is consumed (produced) by (to) which component. One solution would be to add this semantic in the graph itself using either a label or a color for every topic. The disadvantage of this solution is the overload of the graph if several topics would be concerned. We rather propose to make it possible to the user to add the necessary constraints in the predicate part of the style schema, as follows:

Emergency_refined_subscription

Emergency

$\forall\, x : \mathit{Cont} \bullet x.\mathit{topic\_cons} = \{\mathit{P\_Coord}\}$

$\forall\, y : \mathit{Inv} \bullet y.\mathit{topic\_prod} = \{\mathit{D\_Inv}, \mathit{P\_Inv}\} \wedge y.\mathit{topic\_cons} = \{\mathit{D\_Inv}\}$

$\forall\, z : \mathit{Coord} \bullet z.\mathit{topic\_prod} = \{\mathit{P\_Coord}\} \wedge z.\mathit{topic\_cons} = \{\mathit{D\_Inv}, \mathit{P\_Inv}\}$

$\forall\, w : M \bullet w.\mathit{topics} = \{\mathit{D\_Inv}, \mathit{P\_Inv}, \mathit{P\_Coord}\}$

The topic-based refinement is done as it is described in section 4.3, but with respect to the actual supplied parameters. So, the specification of controller , for example, is described by the schema [illegible] accordingly to his role consumer:

Controller

$\mathit{topic\_cons} : \mathbb{F}\ \mathit{TOPIC}$

Let's consider now the interconnection topology underlying managers. In the acyclic as well as in the hierarchical topology, the lack of redundancy in the topology constitutes a limitation in assuring connectivity, since a failure in one manager [illegible] isolates all the subnets reachable from those managers directly connected to [illegible]. Since the considered situations are critical, one may choose the P2P topology. The refinement is done by simply including the topology schema name in the declaration part of the schema [illegible].

6.2 Reconfiguration Specification

Because of the lack of space, we present the specification of only one rule (and only the formal semantic) allowing evolving our [illegible] system while taking into account its defined properties.

- ***Link exchange:*** An existing link between two managers is removed from the topology and a new link is added instead. In this case, we have to ensure that once this operation is applied the topology of managers remains interconnected according to the constraint [illegible] 1 described by the last predicate of the schema [illegible]. In addition, we have to ensure that the connection/disconnection respect the constraint [illegible] 2 described by the 6th predicate. These post-conditions are described in the analytic part of this rule which semantic is described by the schema *Link_Exchange*.

$$
\begin{array}{|l}
\textit{Link\_Exchange} \\
\hline
\Delta \textit{Emergency\_refined\_subscription} \\
m1?, m2?, m3? : \textit{Manager} \\
\hline
m1? \in M \ \land \ m2? \in M \ \land \ m3? \in M \\
(m2?, m3?) \in \textit{PushDD} \land (m3?, m2?) \in \textit{PushDD} \\
\forall\, x, y : M; i : \mathbb{N} \mid x \neq y \\
\ \bullet \exists\, T : \mathrm{seq}\, \textit{EvDispatcher} \\
\ \ \bullet\, i + 1 \leq \# T \ \ \land i \geq 1 \ \ \land \mathrm{ran}\ T \subseteq M \\
\ \ \land\, (T i, T(i+1)) \in \textit{PushDD} \setminus \{(m2?, m3?), (m3?, m2?)\} \ \cup \{(m1?, m3?), (m3?, m1?)\} \\
\ \ \land\, T1 = x \ \ \land T(\# T) = y \\
\textit{PushDD}' = \textit{PushDD} \setminus \{(m2?, m3?), (m3?, m2?)\} \cup \{(m1?, m3?), (m3?, m1?)\}
\end{array}
$$

- **Validation of *Link_Exchange*:** As it is explained in section 5.2, the first step of the verification process is to identify the necessary pre- and post-conditions of an operation. So, for the *Link_Exchange* operation, we proved the following theorem:

Theorem. [illegible]

$$
\begin{array}{l}
\forall\ \text{[illegible]};\ m1?, m2?, m3? : \text{[illegible]};\ \text{[illegible]}2? : \text{[illegible]} \\
\quad \mid m1? \in M \ \land\ m2? \in M \ \land\ m3? \in M \\
\quad \land\, (m2?, m3?) \in \textit{PushDD} \quad \land\, (m3?, m2?) \in \textit{PushDD} \\
\quad \land\, (\forall\ \text{[illegible]}1, \text{[illegible]}2 : \\
\qquad \bullet\, ((\text{[illegible]}1, \text{[illegible]}2) \in \textit{PushDD} \setminus \{(m2?, m3?), (m3?, m2?)\} \\
\qquad \cup \{(m1?, m3?), (m3?, m1?)\} \\
\qquad\quad \Rightarrow (\text{[illegible]}2, \text{[illegible]}1) \in \textit{PushDD} \setminus \{(m2?, m3?), (m3?, m2?)\} \\
\qquad \cup \{(m1?, m3?), (m3?, m1?)\})) \\
\quad \land\, (\forall\, x, y : M; i : \mathbb{N} \mid x \neq y \\
\qquad \bullet\, (\exists\, T : \mathrm{seq}\ \textit{EvDispatcher} \\
\qquad\quad \bullet\, (i + 1 \leq \# T \quad \land i \geq 1 \ \land \mathrm{ran}\ T \subseteq M \\
\qquad\quad \land\, (T i, T(i + 1)) \in \textit{PushDD} \setminus \{(m2?, m3?), (m3?, m2?)\} \\
\qquad\quad\ \cup \{(m1?, m3?), (m3?, m1?)\} \\
\qquad\quad \land\, T1 = x \quad \land\, T(\# T) = y))) \\
\bullet\ \mathrm{pre}\ \textit{Link\_Exchange}\_\text{[illegible]}
\end{array}
$$

As we can notice, the theorem requires a post-condition which is not specified in the schema operation. In fact, the principle of this theorem is to ensure the conformity of the [illegible] with respect to the architecture style. So, since the relation between managers has been changed (′), the theorem verifies all the properties related to this relation and described in the schema style . Whence the post-condition related to the constraint 2.

In this case, the second step of the verification process is to correct the schema operation by adding the identified post-condition, so that we can confirm that this reconfiguring operation leaves the system in a correct state.

7 Conclusion

In this paper, we proposed a compositional formal approach to design correct publish/subscribe architecture styles. We provided a reusable set of patterns and the corresponding composition rules to build architecture styles. We have gone further existing informal studies by proposing a logic-based approach to model the identified basic patterns and to generate new patterns by composition. So, we can verify and reason about the obtained styles. Our approach ensures that the defined patterns and operations respect the principle of information propagation and ensures that any produced information have to reach all the subscribed consumers. We also offer the possibility to refine the designed style to consider the subscription mechanism and the interconnection topology. Moreover, we proposed a formal approach for the specification of the dynamic aspect. This approach is based on an integration of graph-based semantics in the framework of the formal language . Our approach allows to formally describe the dynamic of a software architecture using graph rewriting rules. We elaborate a verification process using the / theorem prover to check that all defined operations preserve the consistency of the architecture style during its evolution.

From an experimental point of view, we are developing a plugin into Eclipse allowing software architects to design visually, using an extended UML-based notation, correct by design Architecture Styles and the corresponding reconfiguration operations. The proposed functions include also generating automatically valid and correct Z specifications.

References

1. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
2. Luckham, D.C., Kenney, J.L., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering 21(4), 336–355 (1995)
3. Oquendo, F.: π-Method: A Model-Driven Formal Method for Architecture-Centric Software Engineering. ACM SIGSOFT Software Engineering Notes 31(3), 1–13 (2006)
4. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)

5. Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of component-based systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
6. Carneiro de Paula, V.C., Ribeiro-Justo, G.R., Cunha, P.R.F.: Specifying and verifying reconfigurable software architectures. In: PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 21–31 (2000)
7. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Transactions On Software Engineering 24(7), 521–533 (1998)
8. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19(3), 332–383 (2001)
9. Cugola, G., Di Nitto, E., Fuggetta, A.: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering 27(9), 827–850 (2001)
10. Baresi, L., Ghezzi, C., Zanolin, L.: Modeling and validation of publish/subscribe architectures. In: Beydeda, S., Gruhn, V. (eds.) Testing Commercial off the shelf Components And Systems, pp. 273–292 (2005)
11. Fenkam, P., Gall, H., Jazayeri, M.: A systematic approach to the development of event based applications. In: SRDS 2003. The 22nd Symposium on Reliable Distributed Systems, Florence, Italy, October 2003, pp. 199–208 (2003)
12. Baldoni, R., Scipioni, S., Tucci-Piergiovanni, S.: Communication channel management for maintenance of strong overlay connectivity. In: ISCC '06. Proceedings of the 11th IEEE Symposium on Computers and Communications, Washington, DC, USA, pp. 63–68. IEEE Computer Society Press, Los Alamitos (2006)
13. Parzyjegla, H., Muhl, G.G., Jaeger, M.A.: Reconfiguring publish/subscribe overlay topologies. In: ICDCSW '06. Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems, Washington, DC, USA, p. 29. IEEE Computer Society Press, Los Alamitos (2006)
14. Cugola, G., Frey, D., Murphy, A.L., Picco, G.P.: Minimizing the reconfiguration overhead in content-based publish-subscribe. In: SAC 2004, pp. 1134–1140. ACM Press, New York (2004)
15. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
16. Wermelinger, M.A.: Specification of software architecture reconfiguration. PhD thesis, Université Nova de Lisbon, Septembre (1999)
17. Chassot, C., Guennoun, K., Drira, K.: Architectural adaptability management for mobile cooperative systems. In: MUE'07. International Conference on Multimedia and Ubiquitous Engineering, Seoul, Korea, pp. 1130–1135 (2007)

An Ontology-Based Approach for Modelling Architectural Styles

Claus Pahl[1], Simon Giesecke[2], and Wilhelm Hasselbring[2]

[1] Dublin City University, School of Computing, Dublin 9, Ireland
cpahl@computing.dcu.ie
[2] University of Oldenburg, Software Engineering Group, D-26111 Oldenburg, Germany
{giesecke,hasselbring}@informatik.uni-oldenburg.de

Abstract. The conceptual modelling of software architectures is of central importance for the quality of a software system. A rich modelling language is required to integrate the different aspects of architecture modelling, such as architectural styles, structural and behavioural modelling, into a coherent framework. We propose an ontological approach for architectural style modelling based on description logic as an abstract, meta-level modelling instrument. Architectural styles are often neglected in software architectures. We introduce a framework for style definition and style combination. The link between quality requirements and conceptual modelling of architectural styles is investigated. The application of the ontological framework in the form of an integration into existing architectural description notations such as ACME and UML-based approaches, and also service ontologies is illustrated.

Keywords: Software architecture modelling, architecture ontology, architectural style, description logics, quality-driven development.

1 Introduction

Architecture descriptions are used as conceptual models in the software development process, capturing central structural and behavioural properties of a system at design stage [1]. The architecture of a software system is a crucial factor for the quality of a system implementation. The architecture influences a broad variety of properties such as the maintainability, dependability or the performance of a system [2]. While architecture description languages (ADLs) exist [3], these are not always suitable to support rich conceptual modelling of architectures [12]. Only a few, such as ACME [2], support the abstraction of architectures into styles and patterns. If formally defined, these can be used to reason about architectures and their properties [7].

We present an architectural style ontology to address this problem, which serves as a modelling basis. Beyond achievements in ACME, we aim to address

– a rich and easily extensible semantic style modelling language,
– operators to combine, compare, and derive architectural styles,

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 60–75, 2007.

- an independent style language that can be applied to extend existing ADLs with style support.

For all three cases, an ontology-based approach to represent architectural knowledge – here in terms of a description logic, which is an underlying logic of ontology languages – is the ideal formal framework [14]. Our architectural style ontology focuses primarily on static, structural aspects of components and connectors. The terminological level of the ontology provides vocabulary and a type language for architectural styles. Instances of this type language are concrete architecture specifications. The structural modelling of architectures is currently adequately supported [5,4,3,1,2] and shall therefore not be addressed in this ontological framework.

The determination of an architectural style, based on a given set of quality requirements, should ideally be the first step in software design [11]. We use a description logic to define an ontology for the description and development of software architectures based on architectural styles that consists of

- an ontology to define architectural styles through a type constraint language,
- an operator calculus to relate and combine architectural styles.

Our aim is to present a conceptual, ontology-based modelling meta-level framework for software architectures, that allows the integration of style aspects into existing architectural description languages (ADLs) without an explicit notion of architectural styles.

We introduce the necessary ontology and description logic foundations in Section 2. We then present an ontology-based modelling approach for architectural styles in Section 3. Relating these styles is the focus of Section 4. We discuss possible extensions to deal with composition in Section 5 and relate the modelling approach to quality-driven development in Section 6. The application of the architectural style language is illustrated in Section 7, before discussing related work and ending with some conclusions.

2 Ontologies and Description Logic

Before presenting the style ontology, we introduce the core elements of the description logic language $\mathcal{ALC}$, which is an extension of the basic attributive language $\mathcal{AL}$ [14]. $\mathcal{ALC}$ provides a set of combinators and logical operators that suffices for the style ontology. Ontologies formalise knowledge about a domain (intensional knowledge) and its instances (extensional knowledge). A description logic, such as $\mathcal{ALC}$, consists of three types of basic notational elements.

- *Concepts* are the central entities. Concepts are classes of objects with the same properties. Concepts represent sets of objects.
- *Roles* are relations between concepts. Roles allow us to define a concept in terms of other concepts.
- *Individuals* are named objects.

Individuals can be thought of as constants, concepts as unary predicates, and roles as binary predicates. We can define our language through Tarski-style model semantics based on an interpretation I that maps concepts and roles to corresponding sets and relations, and individuals to set elements [16]. Properties are specified as *concept descriptions*:

- *Basic concept descriptions* are formed according to the following rules: A is an atomic concept, and if C and D are concepts, then so are $\neg C$ (negation), $C \sqcap D$ (conjunction), $C \sqcup D$ (disjunction), and $C \rightarrow D$ (implication).
- Value restriction and existential quantification, based on roles, are concept descriptions that extend the set of basic concept descriptions. A *value restriction* $\forall R.C$ restricts the value of role R to elements that satisfy concept C. An *existential quantification* $\exists R.C$ requires the existence of a role value.
- Quantified roles can be composed, e.g. $\forall R_1.\forall R_2.C$ is a concept description since $\forall R_2.C$ is one.

These combinators can be defined using their classical set-theoretic interpretations. Given a universe of values $\mathcal{S}$ of values, we define the model-based *semantics of concept descriptions* as follows[1]:

$$\begin{array}{ll}
\top^I & = \mathcal{S} \\
\bot^I & = \emptyset \\
(\neg A)^I & = \mathcal{S} \backslash A^I \\
(C \sqcap D)^I & = C^I \cap D^I \\
(\forall R.C)^I & = \{ a \in S \mid \forall b \in S.(a,b) \in R^I \rightarrow b \in C^I \} \\
(\exists R.C)^I & = \{ a \in S \mid \exists b \in S.(a,b) \in R^I \wedge b \in C^I \}
\end{array}$$

An *individual* x defined by $C(x)$ is interpreted by $x^I \in \mathcal{S}$ with $x^I \in C^I$. Structural subsumption is a relationship defined by subset inclusions for concepts and roles.

- A *subsumption* $C_1 \sqsubseteq C_2$ between two concepts C_1 and C_2 is defined through set inclusion for the interpretations $C_1^I \subseteq C_2^I$.
- A *subsumption* $R_1 \sqsubseteq R_2$ between two roles R_1 and R_2 holds, if $R_1^I \subseteq R_2^I$.

Structural subsumption (subclass) is weaker than logical subsumption (implication), see [14]. Subsumption can be further characterised by axioms such as the following for concepts C_1 and C_2: $C_1 \sqcap C_2 \sqsubseteq C_1$ or $C_2 \rightarrow C_1$ implies $C_2 \sqsubseteq C_1$. $C_1 \doteq C_2$ represents equality.

3 Modelling Architectural Styles

3.1 Basic Architectural Style Ontology

The $\mathcal{ALC}$ language shall now be used to define an architectural style ontology, providing a type and constraint language. The central concepts in this ontology are configuration, component, connector, role, and port types – all of which

[1] Combinators $\sqcap$ and $\rightarrow$ can be defined based on $\sqcup$ and $\neg$ as usual.

are derived from a general concept called an architectural type that captures all architectural notions. These are the elementary architectural types. Components and connectors are at the core of style definitions. Components encapsulate computation and connectors represent communication between the components. Components can communicate through ports. Connectors connect to other components through connectors via their ports, where each port plays a specific role in the context of a connector. Often, a provided and a required port interface is distinguished to add a direction to connectors. Configurations are compositions of components and connectors with their ports and roles.

This vocabulary consisting of five elements needs to be constrained in the ontology in order to ensure the desired semantics:

$$ArchType \sqsubseteq Component \sqcup Connector \sqcup Role \sqcup Port \sqcup Configuration$$

and

$$\begin{array}{ll} Configuration & \doteq \exists hasPart.(Component \sqcup Connector \sqcup Role \sqcup Port) \\ Component & \doteq ArchType \sqcap \exists hasInterface.Port \\ Connector & \doteq ArchType \sqcap \exists hasEndpoint.Role \end{array}$$

The roles *hasPart*, *hasEndpoint* and *hasInterface* are part of the basic vocabulary. This vocabulary of types can be extended to add further elements using the same mechanisms based on subsumption and concept descriptions.

3.2 Defining Architectural Styles

Defining architectural styles is actually done by extending the basic vocabulary of elementary architectural types. The subsumption relationship serves to introduce specific types that form an architectural style.

The Pipe-and-Filter Architectural Style. The specification of architectural styles shall be illustrated using the pipe-and-filter style. We start with an extension of the hierarchy of elementary architectural types in order to introduce style-specific components and ports:

$$\begin{array}{ll} PipeFilterComponent & \sqsubseteq Component \\ PipeFilterPort & \sqsubseteq Port \end{array}$$

These new elements shall be further detailed and restricted to express their connector semantics. Three types of pipe-filter components, *DataSource, DataSink* and *Filter*, shall be distinguished. Their respective connectivity through input and output ports is defined as follows:

$$\begin{array}{lcl} DataSource & \doteq & \leq 1 \ hasPort \sqcap \exists hasPort.Output \\ DataSink & \doteq & \leq 1 \ hasPort \sqcap \exists hasPort.Input \\ Filter & \doteq & = 2 \ hasPort \sqcap \exists hasPort.Input \sqcap \exists hasPort.Output \end{array}$$

DataSource, DataSink, and Filter are defined as components of a pipe-filter architectural style. Each of these components is characterised through the number and types of component ports using so-called predicate restrictions on a numerical domain ($\leq n$ and $= n$ are used to express $hasPort.(n|n \leq 1)$ for instance) and the usual concept descriptions. In addition to these more structural conditions that define the connections between the component types, a number of classification constraints shall be formulated that further refine the initial enumeration of pipe-filter components by describing how subtype classification is applied.

– *Disjointness* requires the individual components to be truly different:

$$DataSouce \sqcap DataSink \sqcap Filter = \bot$$

– *Completeness* requires pipe-filter components to be made up of only the three specified types:

$$PipeFilterComponent \doteq DataSource \sqcup DataSink \sqcup Filter$$

The Hub-and-Spoke Architectural Style. In addition to the well-known pipe-and-filter style [7,2], we introduce another architectural style, the hub-and-spoke style. This style abstracts a system that manages a composition from a single location, the hub, which is normally the participant initiating the composition. The composition controller (the hub) is usually remotely accessed by the participants (the spokes). This is the most popular and usually default distribution configuration for service compositions. We would specify:

$$Hub \sqsubseteq Component \quad \text{and} \quad Spoke \sqsubseteq Component$$

with suitable completeness and disjointness constraints.

$$Hub \doteq \exists hasPort.Input \quad \text{and} \quad Spoke \doteq \exists hasPort.Output$$

explains that hubs receive incoming requests from spokes. Further constraints would limit the number of hubs to one, whereas spokes can be instantiated in any number.

3.3 Architectural Styles and Architecture Modelling

So far, we have addressed specifications of architectural properties at the architectural type level. These specifications are constraints that apply to concrete architecture descriptions formulated using the defined architectural types. The question is how these type-level specifications are applied to act as architectural styles. An instantiation of these type-level properties, i.e. an architecture, could be described by instantiating the elementary types only, fully ignoring any style-specific constraints. Thus, a specification of architectural properties is not what we would commonly see as an architectural style. The configuration type matches what an architectural style needs to express. It defines a specific vocabulary of components and other elements and their constraints. Therefore, we define an architectural style to be a subtype (subsumption) of the configuration type.

$$\begin{aligned} PipeFilterStyle &\sqsubseteq Configuration \\ PipeFilterStyle &\doteq \exists hasPart.(PipeFilterComponent \sqcup PipeFilterConnector \\ &\quad\ \sqcup Role \sqcup Port) \end{aligned}$$

is, together with related concept descriptions, a style definition. What clearly identifies a style is the configuration subtype that acts as a root of the style definition. An architecture description conforming to an architectural style is a subtype of the defined style configuration, e.g. $PipeFilterStyle$. All elements linked to the style (or its subtypes) directly or transitively through $hasPart$ and the other predefined roles can be used to describe an architecture.

A distinguishing property of our approach is that the basic architecture vocabulary with notions like component or connector is defined with the same mechanism at the same layer as the architectural styles. The basic architectural style ontology itself is consequently an architectural style, albeit an abstract and unconstraining one – with the trivial equality as the required subsumption.

The ontology and the styles defined based on the ontology aim to provide a type language for architecture definitions. Components in an architecture definition are instances of the elements of an architectural style. In terms of description logics, the architecture elements are instances of the concepts that define an architectural style. The style constrains the use of the architecture elements. This architecture layer – the instances layer in terms of our ontology – shall not be addressed in terms of our framework. Instead we will demonstrate how this framework is independent of specific ADLs and can be applied to them as a style sublanguage in Section 7. Our aim is not to define yet another ADL.

4 Relating Architectural Styles

Each architectural style is defined by a separate specification as an extension of the basic ontology of elementary architecture elements. In order to reuse architectural styles as specification artefacts, these styles are often related to each other, e.g. to be compared to each other or to be derived from another [21]. Different styles can be related based on ontology relationships. We give an overview of the central operators restriction, union, intersection and refinement and define the semantics of this operator calculus. Instead of general ontology mappings, we introduce a notion of style specification and define style comparison and development operators on it.

4.1 Style Syntax and Semantics

Before defining the operators, the notions of architecture specification and styles and their semantics need to be made more precise. We assume a style to be a specification $Style = \langle \Sigma, \Phi \rangle$ based on the elementary type ontology with

- a signature $\Sigma = \langle C, R \rangle$ consisting of concepts C and roles R,
- concept descriptions $\phi \in \Phi$ based on Σ.

$Style$ is interpreted by a set of models M. The model notion [16] refers to algebraic structures that satisfy all concept descriptions ϕ in Φ. The set M contains algebraic structures $m \in M$ with

- sets of objects C^I for each concept C and

- relations $R^I \subseteq C_i^I \times C_j^I$ for all roles $R : C_i \rightarrow C_j$

such that m satisfies the concept description. This satisfaction relation is as usual defined inductively over the connectors of the description logic $\mathcal{ALC}$.

The combination of two styles should be conflict-free, i.e. semantically, no contradictions should occur. A *consistency* condition can be verified by ensuring that the set-theoretic interpretations of two styles S_1 and S_2 are not disjoint, $S_1^I \cap S_2^I \neq \emptyset$, i.e. their combination is satisfiable and no contradictions occur.

Note, that this calculus of operators is not strictly an algebra in terms of styles – only in terms of specifications. A resulting specifications can be defined as a style by identifying a new root configuration.

4.2 Restriction

While often architectural styles are used as-is in combinations and relationships, it is sometimes desirable to focus on specific parts, before for instance refining an architectural style. Restriction is an operator that allows architectural style combinations to be customised and undesired elements (and their properties) to be removed. A *restriction*, i.e. a projection or view, can be expressed using the restriction operator $\langle \Sigma, \Phi \rangle_{|\Sigma'}$ for a specification, defined by

$$\langle \Sigma, \Phi \rangle_{|\Sigma'} \overset{\text{def}}{=} \langle \Sigma \cap \Sigma', \{ \phi \in \Phi \mid rls(\phi) \in rls(\Sigma \cap \Sigma') \wedge cpts(\phi) \in cpts(\Sigma \cap \Sigma') \} \rangle$$

with the usual definition of role and concept projections $rls(\Sigma) = R$ and $cpts(\Sigma) = C$ on a signature $\Sigma = \langle C, R \rangle$. Restriction preserves consistency as constraints are, if necessary, removed.

4.3 Intersection and Union

Two architectural styles $S_1 = \langle \Sigma_1, \Phi_1 \rangle$ and $S_2 = \langle \Sigma_2, \Phi_2 \rangle$ shall be assumed.

- The *intersection* of S_1 and S_2, expressed by $S_1 \cap S_2$, is defined by

$$S_1 * S_2 \overset{\text{def}}{=} \langle \Sigma_1 \cap \Sigma_2, (\Phi_1 \cup \Phi_2)|_{\Sigma_1 \cap \Sigma_2} \rangle$$

  Intersection is semantically defined based on an intersection of style interpretations, achieved through projection onto common signature elements.
- The *union* of S_1 and S_2, expressed by $S_1 \cup S_2$, is defined by

$$S_1 + S_2 \overset{\text{def}}{=} \langle \Sigma_1 \cup \Sigma_2, \Phi_1 \cup \Phi_2 \rangle$$

  Union is semantically defined based on a union of style interpretations.

In the case of fully different architectural styles, their intersection results in the elementary architecture types and their properties. Both operations can result in consistency conflicts.

4.4 Refinement

Consistency is a generic requirement that should apply to all combinations of architecture ontologies. A typical situation is the derivation of a new architectural styles from an existing one [9]. The *refinement* operator that we are going to introduce is a consistent derivation. Refinement can be linked to the subsumption relation and semantically constrained by an inclusion of interpretations, i.e. the models that interpret a style. Refinement carries the connotation of preserving existing properties, for instance the satisfiability of the original style specification. In this terminology, the pipe-and-filter style is actually a refinement of the basic architectural type vocabulary. As the original types are not further constrained, the extension is consistent.

An explicit consistency-preserving refinement operator shall be introduced to provide a constructive subsumption variant that allows

- new subconcepts and new subrelationships to be added,
- new constraints to be added if these apply consistently to the new elements.

Assume a style $S = \langle \Sigma, \Phi \rangle$. For any specification $\langle \Sigma', \Phi' \rangle$ with $\Sigma \cap \Sigma' = \emptyset$, we define a *refinement* of S by $\langle \Sigma', \Phi' \rangle$ through

$$S \oplus \langle \Sigma', \Phi' \rangle \overset{\text{def}}{=} \langle \Sigma + \Sigma', \Phi + \Phi' \rangle$$

The precondition $\Sigma \cap \Sigma' = \emptyset$ implies $\Phi \sqcap \Phi' = \bot$, i.e. consistency is preserved. In this situation, existing properties of $S = \langle \Sigma, \Phi \rangle$ would be inherited by $S \oplus \langle \Sigma', \Phi' \rangle$. Existing relationships can in principle be refined as long as consistency is maintained – which might require manual proof in specific situations that go beyond the operator-based application.

4.5 Architectural Style Development

The main aim of these operators is to support the development of architectural styles. We imagine a catalogue of styles that is used by the software architect to describe architectures.

- The operator calculus allows individual styles from the catalogue to be compared. For instance, two styles can be united to test if the set of concepts they describe overlap. The consistency condition is used for this test.
- An existing style can be adapted. Refinement allows to add further elements and constraints, making the style more specific. Styles can also be made more general by removing constructs and properties through restriction.

This catalogue could be implemented as a repository.

The hub-and-spoke style shall be extended using the refinement operator. The idea is to add a broker component, which spokes would initially contact and which would assign a hub to them.

$$BrokeredHubSpokeStyle \doteq HubSpokeStyle \oplus \langle \Sigma, \Phi \rangle$$

where the signature Σ is defined by

$$\langle \; \{ \; BrokerComponent, BrokerSpokeConnector, BrokerHubConnector, \\ HubRegistrationRole, SpokeAllocationRole \; \} \; , \; \{ \; \} \; \rangle$$

and the properties Φ are defined by

$$\begin{array}{ll} BrokerComponent & \doteq HubSpokeComponent \sqcap \exists hasInterface.Port \\ BrokerSpokeConnector & \doteq HubSpokeConnector \sqcap \\ & \quad \exists hasEndpoint.SpokeAllocationRole \\ BrokerHubConnector & \doteq HubSpokeConnector \sqcap \\ & \quad \exists hasEndpoint.HubRegistrationRole \end{array}$$

We would automatically get $BrokeredHubSpokeStyle \sqsubseteq HubSpokeStyle$ as a consequence of the application of the refinement.

5 Composite Elements in Architectural Styles

An explicit support for composition is an important element of conceptual modelling languages. Composition is also central for software architectures. As an extension, we introduce two types of composite elements for architectural style specifications.

5.1 Components

Component hierarchies shall consist of unordered subcomponents, expressed using a component composition operator "$\models$", which adds another dimension to the subsumption-based subtype relationship. An example is $Configuration \models Port$, meaning that a *Configuration* consists of *Port*s as parts. This is actually a reformulation of the previously used *hasPart* relationship. In order to provide this with an adequate semantics, interpretations of configurations would have to be seen as tuple-structured elements.

5.2 Connectors

Connectors can be *process assemblies* that consist of ordered process elements, expressed using a set of process composition operators sequence ";", iteration "!", and choice "+". An example is $C \doteq D; E$, meaning that connector C is actually a process sequence of connectors D and E. This sequence can be semantically defined by requiring $D.in \doteq C.in$, $E.in \doteq E.out$, and $C.out \doteq E.out$ in order to express sequencing dependencies.

5.3 Discussion

Note, that these operators are specific to the respective architecture element. While the structural composition is often sufficient, full process specifications with interaction and data flow elements, however, cannot be expressed in the notational format introduced here. Ontological support for, for instance, the process combinators exists in description logics [14]. While this aspect of composition could not have been investigated here in detail, we felt it important to briefly discuss the benefits and also the potential of ontologies and description logics to provide adequate language support.

6 Quality-Driven Architecture

The use of styles in architecture design implies certain properties of software systems, as these styles are abstractions of successfully implemented systems that are usually easy to understand, to manage, or to maintain [11,12]. While of course functional properties of components are vital, non-functional quality aspects ranging from availability, performance, and maintainability guarantees to costs are equally important and need to be captured to clearly state the quality-of-service (QoS) requirements. The reliability of a system, the availability of services, and the individual component and overall system performance are often crucial. Links exist between architecture models, that based on the component and connector view allocate function to structure, and QoS properties of these systems [8,10]. A mere statement of required QoS properties is therefore often not sufficient to actually guarantee these properties. We look at architectural styles to illustrate this point.

6.1 Style-Based Quality Description

A catalogue of *architectural styles* or *patterns* [15], consisting of styles such as pipe-and-filter and hub-and-spoke, may be utilised by software architects to build architectures that exhibit some desired quality properties. Each of the styles in the catalogue is associated with certain QoS characteristics, that would be exhibited during the deployment and execution of system compositions. The ISO 9126 standard for software product quality to support the evaluation of software can serve as a starting point here that defines quality attributes and metrics [19,20].

We illustrate this using an architectural style. Some of the advantages of the hub-and-spoke architectural style in terms of QoS aspects are:

- Composition is easily *maintainable*, as composition logic is all contained at a single participant, the central hub.
- *Low deployment overhead* as only the hub manages the composition.
- Composition can include externally controlled participants. Web service technologies, for instance, would enable the *reuse* of existing service components.

The main disadvantages of this architectural style are:

- A single point of failure at the hub provides *poor reliability* and *availability*.
- A communication bottleneck at the hub results in *restricted scalability*. SOAP messages have considerable overhead for message deserialisation and serialisation.
- The high number of messages between hub and spokes is sub-optimal.

The style ontology can be extended by a quality ontology to capture a vocabulary of quality attributes and corresponding metrics using quality-specific properties.

$$\begin{aligned} HubSpokeStyle \doteq\ & \exists hasAdvQual.(Maintainable \sqcup LowOverhead \sqcup Reusable) \sqcap \\ & \exists hasDisadvQual.(\neg Reliable \sqcup \neg Scalable \sqcup \neg Performant) \end{aligned}$$

Some of these quality concepts are based ISO 9126. Further formalised descriptions such as the association of metrics, for instance in the format $Performant \doteq \exists hasMetric.ResponseTime$, are possible.

6.2 Quality Evaluation

Quality-driven development requires quality attributes to be evaluated and confirmed. The qualities of newly derived styles cannot always be taken for granted. Only through empirical evaluations can these expected qualities be confirmed.

A Goal-Question-Metric (GQM) approach to quality goal evaluation [18], a method which allows metric to be derived from abstract quality criteria, can support this quality evaluation endeavour. Implemented systems can be evaluated using the metrics derived from the quality goals via GQM.

7 Integration with Architecture Description Languages

Our aim is not to define yet another ADL. Our aim is to define a versatile architectural style language that can be combined with existing ADLs for a variety of reasons:

- to semantically define an existing style language and to allow reasoning within this semantic framework,
- to provide an ADL-independent style language that can be added to ADLs that do not have an explicit notion of styles,
- to provide a generic terminological framework into which quality aspects can be integrated.

We will look at ACME to illustrate the first point, at UML to illustrate the second point, and at service ontologies like WSMO to illustrate the third point. The architectural style ontology could be used in the first case to formally define the ACME style language. In the second case, the style ontology could be mapped to MOF, giving it an abstract syntactical definition through MOF. Equally, an integration with a service ontology such as WSMO or OWL-S is another application of our approach.

We will not fully formalise these mappings for the three applications here – our aim is solely the motivation of these possibilities and the benefits from them. The focus of this paper is only on the definition of the style ontology.

7.1 ACME

ACME is an ADL that supports the component and connector view on architectures [2]. For that purpose, a basic set of architecture elements is introduced.

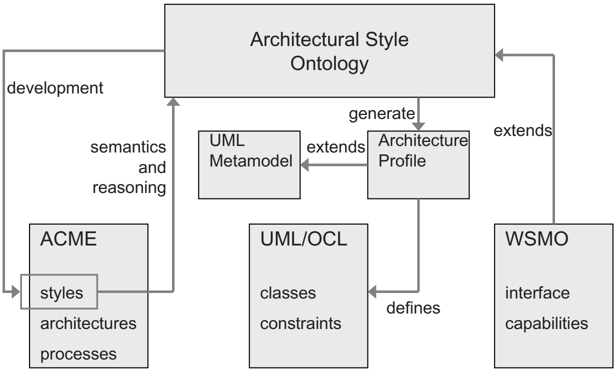


Fig. 1. Application of the Architectural Style Ontology to ACME, UML and WSMO

These include the same terms that we have defined. ACME provides specific support to define architectural styles. The basic architecture elements are supported by the corresponding types. A style, called a family in ACME, is then a collection of constrained type definitions. Invariants can be expressed using a constraint language based on properties. Properties in ACME are name-value pairs. ACME does not provide native support for the interpretation these properties.

Our architectural style ontology can provide a standard semantics for properties. Due to the syntactic equality of the elementary types, a mapping from ACME into our ontological framework can easily be defined. The intended semantics of ACME types matches the formal semantics we have introduced here. This has the following benefits for ACME:

- The ACME type language is formally defined through the architectural style ontology.
- A framework for the analysis and reasoning about styles and their properties is introduced.
- The operator calculus enriches the mechanisms to develop architectural styles effectively and consistently.

7.2 UML and OCL

UML is often used to describe software architectures [22]. Class diagrams define components and connections between components through classes and associations. Additional constraints can be added using the Object Constraint Language OCL [23].

In terms of UML, architectural styles are MOF meta-level models, i.e. architectural style definitions correspond to the M2 level. Description logic can be translated to MOF easily, thanks to the Ontology Defintion Metamodel (ODM) [29], which defines a number of MOF-based metamodels including description logics and UML and a range to central transformations between them. This reference framework can be used to translate a given architectural style into a MOF-compliant metamodel. The difficulty here is only that this MOF metamodel is not UML-metamodel compliant. This means that compliance can only be achieved by adapting the standard transformation to define a suitable UML profile. The problem is similar to the need to clearly identify a style and to guarantee its correct application. The profile needs to provide UML-compliant model elements that must only be used in a style-conformant way.

7.3 WSMO

WSMO [24] is, like OWL-S [24], an ontology-based approach to describing services. In the traditional understanding, these two are not ADL [3]. Their aim is to provide a vocabulary that allows the description on functional and non-functional attributes of services and their operations in terms of pre- and post-conditions or quality attributes. Nonetheless, looking at service ontologies helps us to understand how quality attributes, possibly ISO 9126-compliant, can be integrated into an architectural style-driven ADL. This is also one of the reasons for us to use an ontological approach in the first place. Services and their operations are the concepts in WSMO (or OWL-S). Functionality information and quality attributes in WSMO are categorised into interface (syntax) and capability (semantics, quality) attributes and are described in terms of properties in the ontology.

8 Related Work

Formalising architectural styles is the first step of understanding their properties and the resulting impact on architectures and software systems. A seminal paper in this context is [7]. A formal framework based on the model-theoretic specification language Z is given. Abowd et al. introduce the detailed formal specification of architectural styles, e.g. for the pipe-and-filter style. This work has started the integration of semantics into architectural descriptions. The description logic we have used here provides the same expressive power to formulate structural architectural properties (we discuss the behavioural properties addressed by Abowd et al. below). The reason for choosing an ontological approach in our case are pragmatic. An ontological framework for this approach is an ideal choice since extension through subsumption is a natural choice to develop a catalogue of styles. The existence of meta-level frameworks such as the Ontology Definition Metamodel ODM with its predefined transformations makes ontologies and their dynamic logic foundations suitable as an interoperable notation that can be integrated with existing ADLs.

An ontology-based approach is also taken by work addressing service and process ontologies. OWL-S [24] and WSMO [24] are examples for service ontologies, which we have already discussed. WSPO [27,26] and SWSF [28] are ontological frameworks with a stronger focus on service processes. These are of interest from an architectural perspective as they address service orchestration and choreography as two forms of architectural configuration in the form of component interaction. While we have not addressed this aspect and have rather limited our discussion to more structural properties, an integration of an architectural style ontology with these service ontologies is promising [17,14].

Around the notion of an architectural style, similar concepts have emerged [6]. In [13], a notion of an architectural scenario is used to aid analyses in the design of architectures. Direct and indirect scenarios are used to view software systems as information processing software artefacts or to view these artefacts as subjects in a change and evolution process, respectively. The dynamic nature of software architectures is emphasised in contrast to the more static view of architectural styles and their application. A similar argumentation is followed by [30]. Associating a system to a single architectural style is often not sufficient. The notion of a mode, similar to a scenario, is introduced. Modes can be changed through structural and evolution constraints, which aims to support the self-organisation of service-based systems.

9 Conclusions

In addition to structural and behavioural properties of software architectures, meta-level constructs such as architectural styles, scenarios, or modes have recently received much interest in the software architecture community. Architectural styles have emerged as architecture abstractions that strongly influence the quality of architectures and their implementations. Our discussion of quality-of-service attributes reflects this observation. Architectural styles are often also linked to platforms; middleware platforms often support only specific styles. In this context, architectural styles help to determine essential aspects of software systems.

Using an ontological, description-logic-based setting for software architecture has a number of benefits, such as a concise and precise notation with formal semantics [7], an extensible type language based on subsumption and constraints [14], and a style combination algebra based on ontology technologies. The tractability of reasoning is a central issue for description logics. The logic $\mathcal{ALC}$ that we have used for this architectural style ontology is decidable, i.e. provides the basis for termination and reliable tool support.

Overall, ontology mechanisms provide an ideal conceptual modelling support, using a classical ontology approach. The notation is adequate, as the examples have demonstrated, to model architectural styles. While the notation is suited to formulate and relate architectural styles focusing on structural aspects, the introduction of composite element has demonstrated the lack of process modelling capabilities in the notation introduced here. Concepts are not meant to model the details of structured behaviour; using concepts to express structured processes is therefore not an adequate solution. While an integration with service or process ontologies is desirable, the seamless integration requires further investigations.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. SEI Series in Software Engineering. Addison-Wesley, Reading (2003)
2. Garlan, D., Schmerl, B.: Architecture-driven modelling and analysis. In: Cant, T. (ed.) SCS'06. Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems. Conferences in Research and Practice in Information Technology, vol. 69 (2006)
3. Medvidovic, N., Taylor, R.N.: A Classification and Comparison framework for Software Architecture Description Languages. In: Jazayeri, M. (ed.) ESEC 1997 and ESEC-FSE 1997. LNCS, vol. 1301, pp. 60–76. Springer, Heidelberg (1997)
4. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
5. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
6. Cuesta, C.E., del Pilar Romay, M., de la Fuente, P., Barrio-Solorzano, M.: Architectural Aspects of Architectural Aspects. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, Springer, Heidelberg (2005)
7. Abowd, G., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology 4(4), 319–364 (1995)
8. Spitznagel, B., Garlan, D.: Architecture-based performance analysis. In: SEKE'98. Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (June 1998)
9. Baresi, L., Heckel, R., Thöne, S., Varro, D.: Style-based refinement of dynamic software architectures. In: WICSA4. Proc. 4th Working IEEE/IFIP Conference on Software Architecture, pp. 155–164. IEEE, Los Alamitos (2004)
10. Cortellessa, V., Di Marco, A., Inverardi, P.: Software performance model-driven architecture. In: SAC '06. Proceedings of the 2006 ACM symposium on Applied computing, pp. 1218–1223. ACM Press, New York (2006)
11. Giesecke, S.: A Method for Integrating Enterprise Information Systems based on Middleware Styles. In: ICEIS'06. International Conference on Enterprise Information Systems, Doctoral Symposium, pp. 24–37. INSTICC Press (2006)
12. Giesecke, S., Bornhold, J., Hasselbring, W.: Middleware-induced Architectural Style Modelling for Architecture Exploration. In: Proc. Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press, Los Alamitos (2007)
13. Kazman, R., Carriere, S.J., Woods, S.G.: Toward a Discipline of Scenario-based Architectural Evolution. Annals of Software Engineering 9(1-4), 5–33 (2000)
14. Baader, F., McGuiness, D., Nardi, D., Schneider, P.P. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
15. Barrett, R., Patcas, L.M., Murphy, J., Pahl, C.: Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In: ICWE'06. International Conference on Web Engineering, Palo Alto, US, ACM Press, New York (2006)
16. Kozen, D., Tiuryn, J.: Logics of programs. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 789–840. Elsevier, Amsterdam (1990)
17. Schild, K.: A Correspondence Theory for Terminological Logics: Preliminary Report. In: Proc. 12th Int. Joint Conference on Artificial Intelligence, Sydney, Australia (1991)

18. Basili, V., Caldiera, G., Rombach, D.: The Goal/Question/Metric approach. In: Encyclopedia of Software Engineering, vol. I, pp. 528–532. Wiley, Chichester (1994)
19. Jung, H.-W., Kim, S.-G., Chung, C.-S.: Measuring software product quality: A survey of ISO/IEC 9126. IEEE Software 21(5), 88–92 (2004)
20. ISO/IEC: Software engineering – Product quality – Part 1: Quality model. Published standard (June 2001)
21. Canal, C., Pimentel, E., Troya, J.M.: Compatibility and inheritance in software architectures. Science of Computer Programming 41, 105–138 (2001)
22. Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, J., Nord, R., Stafford, J.: Documenting Software Architecture: Documenting Behavior. Technical Report CMU/SEI-2002-TN-001. SEI, Carnegie Mellon University (2002)
23. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language – Precise Modeling With UML, 2nd edn. Addison-Wesley, Reading (2003)
24. Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontology 1(1), 77–106 (2005)
25. DAML-S Coalition: DAML-S: Web Services Description for the Semantic Web. Horrocks, I., Hendler, J. (eds.): ISWC 2002. LNCS, vol. 2342, pp. 279–291. Springer, Heidelberg (2002)
26. Pahl, C.: An Ontology for Software Component Matching. International Journal on Software Tools for Technology Transfer (STTT), Special Edition on Component-based Systems Engineering 7 (in press, 2007)
27. Pahl, C., Casey, M.: Ontology Support for Web Service Processes. In: ESEC/FSE'03. Proc. European Software Engineering Conference and Foundations of Software Engineering, ACM Press, New York (2003)
28. Semantic Web Services Language (SWSL) Committee: Semantic Web Services Framework (SWSF) (2006), http://www.daml.org/services/swsf/1.0/
29. Object Management Group: Ontology Definition Metamodel - Submission (OMG Document: ad/2006-05-01). OMG (2006)
30. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for Software Architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, Springer, Heidelberg (2006)

FIESTA: A Generic Framework for Integrating New Functionalities into Software Architectures

Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien

LIFL, INRIA team ADAM
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq, France
{waignier,lemeur,duchien}@lifl.fr

Abstract. When an application must evolve to cope with new context and user requirements, integrating new functionalities into its software architecture is necessary. The architect has thus to manually modify the architecture description, which is often tedious and error prone.

In this paper, we propose FIESTA, a generic framework for automatically integrating new functionalities into an architecture description. Our approach is inspired by TranSAT, an integration framework. However, TranSAT is dedicated to a specific architecture description language (ADL) while our approach is ADL-independent. We have performed a domain analysis, studying for many ADLs how to integrate new functionalities. Based on this analysis, we have defined a generic ADL model to manipulate and reason about architectural elements that are involved in integration. Furthermore, we have developed a generic integration engine.

1 Introduction

A software architecture is an abstract specification of a system structure. It enables the architect to identify the various components of the system, their interfaces and to reason about their assembly, without having to consider implementation details. Many architecture description languages (ADLs) have been proposed to represent various system properties [18].

An architecture description facilitates the comprehension, the analysis and the prototyping of a system, as well as being a good basis for its evolution. A system may need to evolve to take into account new functionalities in order to better address the application environment and user requirements. Nevertheless, adding a new functionality implies that the architect has to manually modify the architecture description. These modification operations are often low-level, tedious and prone to error, particularly in the case of cross-cutting functionalities that require invasively modifying specifications in several files.

The TranSAT framework proposes an approach to automatically integrating new functionalities into a software architecture description [6,7]. This approach,

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 76–91, 2007.

inspired by Aspect Oriented Programming [16], relies on both the definition of *architecture integration patterns*[1] and the use of a *weaver* to automatically integrate patterns into the target architecture. An architecture integration pattern makes explicit and reusable all the knowledge and expertise required to integrate a given functionality into a software architecture. A pattern consists of three parts: a new plan, a join point mask and a set of transformation rules. The new plan is a component assembly corresponding to a given functionality. The join point mask expresses properties that the target architecture must satisfy for the integration to be possible. Finally, the set of transformation rules specifies the operations that the weaver has to perform to integrate the new plan into the target architecture. TranSAT is however dedicated to a single ADL, SafArchie [5], which limits its applicability.

In this paper we present FIESTA, a Framework for Incremental Evolution of SofTware Architectures. FIESTA generalizes the TranSAT approach in order to be independent of any specific ADL, keeping only the general ideas of defining architecture integration patterns and using a weaver to perform the integration. To generalize the TranSAT approach, we have performed a domain analysis to understand the integration process in architectures described in various ADLs. This analysis has allowed us to identify the common architecture elements that are involved in integration, leading to the definition of a generic ADL model. This analysis has also enabled us to define more abstract expressions to specify a join point mask and the transformation rules. Overall, the FIESTA framework assists the architect during the integration process, only requiring him to specify his design choices. Furthermore, our generic framework can be extended to support new ADLs by specifying ADL-specific properties in functions dedicated to the configuration of our framework.

The rest of this paper is organized as follows. Section 2 describes our domain analysis and Section 3 presents our framework, FIESTA, including our definition of architecture integration patterns, our generic ADL model and our weaver. Section 4 describes some related work. Section 5 concludes and provides some future work.

2 Domain Analysis

In our domain analysis we have focused on the problem of adding new functionalities to the Comanche Web server. Comanche is provided as an example of the use of Julia, the Java implementation of the Fractal component model [8]. We have manually integrated several new functionalities into this application, such as caching, logging, encryption/decryption and authentication. Furthermore, to understand the integration process for different ADLs, we have performed these integrations not only in the context of Fractal ADL [9] but also in the context of CCM [19], Olan [4], SOFA [15] and SafArchie [5]. This domain analysis has led us to identify, for each of these ADLs, the elements involved in integration.

[1] Originally called software architecture patterns.

We have also studied other ADLs, such as Unicon [21], Darwin [17], Wright [2], AADL [3], Acme [12] and UML2 [20], for a total of 15 ADLs. From our domain analysis, we have established a common vocabulary to abstract architectural elements that are involved in an integration process. Furthermore, we have determined that adding a new functionality can be performed through two categories of integration.

2.1 Common Vocabulary Across ADLs

We have identified six architectural elements that the software architect needs to reason about when integrating new functionality into a component-based architecture description: component, connector, configuration, communication point, communication element and role. We define them as follows:

- A *component* is a computational element or a data store of a system;
- A *connector* describes the interactions between components;
- A *configuration* represents an assembly of components and connectors, which corresponds to the structure of the application;
- A *communication point* is an element of a component that enables the component to communicate with its environment, *i.e.,* the other components;
- A *communication element* corresponds to what is exchanged through connectors between components;
- A *role* is a participant of the interaction represented by the connector.

Furthermore, we distinguish primitive components from composite components, as well as direct communication points from delegated communication points. A primitive component is a basic computational element and a composite defines a given assembly of primitive and composite components. A direct communication point is the source or the sink of communication. A delegated communication point propagates communication elements toward another communication point.

Table 1 illustrates how these six elements map to the specific elements of Fractal ADL, CCM, SafArchie, SOFA, Unicon, AADL and UML2. We have chosen these ADLs because they are representative of various characteristics that can be found in many other ADLs.

As shown in Table 1, the concepts of primitive and composite components exist in numerous ADLs but some, such as CCM, only provide primitive components. Connectors are often simple bindings, but they can also be dedicated software entities, as in SOFA and Unicon. For example in SOFA there are three predefined connectors (CSProcCall, EventDelivery, and DataStream) and the user may also define new connector types. The concept of configuration may be explicit as in AADL and CCM, or implicit, *i.e.,* the global structure of the application corresponds to the most outwards (*i.e.,* higher-level) defined composite.

A communication point is often called an interface or port, depending on the ADL. Furthermore, if the ADL is service-oriented, an interface or a port may be qualified as client or server, which is equivalent to declaring them as required or provided, respectively. In the case of a data-oriented ADL, the direction of the dataflow may be indicated, *e.g.,* in AADL, in (resp. out) specifies that the

Table 1. Common architecture elements

ADL	Component	Connector	Configuration	Communication point	Role	Communication element
Fractal	primitive, composite	binding	composite	client/server interface	client/server	synchronous method call
CCM	primitive	binding	component assembly descriptor	port (Facet, receptacles, event source, event sink)	client/server	asyn/syn method call, syn event
SafArchie	primitive, composite	binding	composite	port	endpoints	synchronous method call
SOFA	primitive, composite	software entity	composite	required/provided interface	required/ provided	asyn/syn method call
Unicon	primitive, composite	software entity	composite	player	caller, definer, participant, load, *etc.*	syn method call, asyn event, syn typed flow
AADL	primitive, composite	binding	system	in/out data port, event port, event data port	input/output	synchronous typed flow
UML2	primitive, composite	link (binding)	composite	required/provided interface, required/provided/ complex port	connectorEnd	asyn/syn operation

dataflow is coming in (resp. out) of the communication point. Other kinds of communication points can be found as in Unicon where communication points are known as players, 14 player types are defined in total.

A role corresponds to the access point of a connector. For example, in Fractal and CCM, there are two roles, one at each endpoint of the connector, one endpoint being client and the other one server. There is also one role at each endpoint of SafArchie connectors, but they have no name. There may be more than two roles as illustrated by Unicon, which provides 11 predefined roles.

Finally, communication elements are contained within a communication point. Communication elements differ across ADLs. In the Java implementation of the Fractal component model, communication is performed through calls to methods that are defined in interfaces. Method calls may be synchronous or asynchronous. In SafArchie, methods are further declared as required or provided, indicating whether the component requires or provides the operation to function. Consequently, in SafArchie a communication port is not declared as required (resp. client) or provided (resp. server) as it may contain both required and provided methods. Communication elements may also be for example events or typed flows.

2.2 Two Categories of Integration

When performing our domain analysis we have identified two categories of integration. An integration may correspond to (i) adding a new connector or (ii) modifying an existing connector, or both. To illustrate these two categories, we consider the manual integration of a logging and a caching functionality into the Comanche Web server application. The examples are specified in Fractal ADL.

A high-level representation of the structure of the Comanche application is shown in Figure 1. When the `Request receiver` receives a URL request from

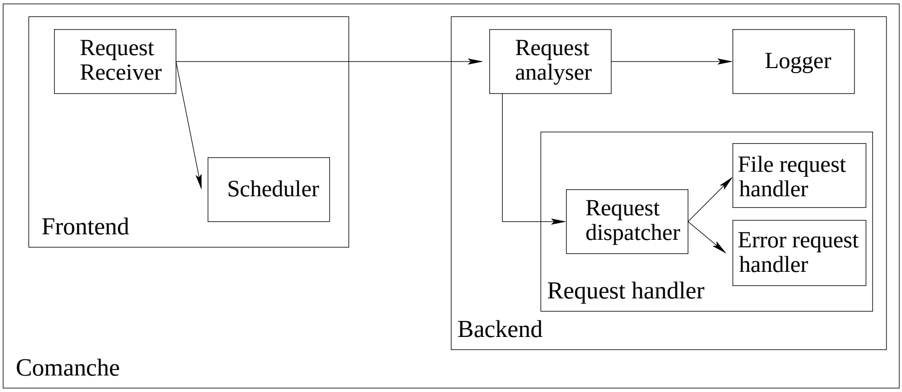
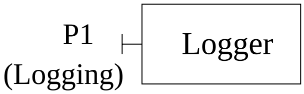


Fig. 1. Structure of the Comanche Web server



```
public interface Logging {
 public void log(String msg);
}

<interface name="P1"
          role="server"
          signature="Logging"/>
```

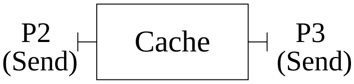
Fig. 2. Definition of the `Logger` component

a client, it triggers the `Scheduler` to create a new thread to handle the client. The request is first forwarded to the `Request analyzer`, then to the `Request dispatcher` and finally to the `File request handler`. If the file specified by the URL is found on the filesystem then the `Request dispatcher` sends file back to the client, otherwise it calls the `Error request handler`.

Adding a Connector: The Logging Example. We propose to add a `Logger` component that will be connected to the `Request analyzer` component in order to log all the requests sent to the Web server. The `Logger` component is shown in Figure 2. It has a communication point `P1`, which provides the `log` method through the `Logging` interface.

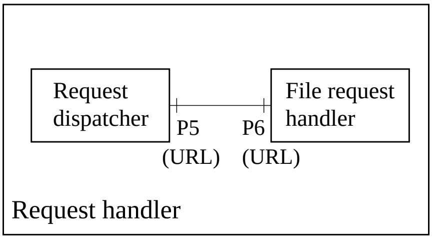
To integrate the `Logger` component, the architect has to add a new communication point, say `P8`, on the `Request analyzer` component. The communication point `P8` should be created so that it is *compatible* with `P1`. As our example is specified in Fractal ADL, compatibility implies that `P8` contains the `Logging` interface and is declared as client. To complete the integration, the `Logger` component should be placed in the `Backend` composite, and `P1` and `P8` connected using a connector.

Modifying a Connector: the Caching Example. We now consider adding a `Cache` component between the `Request dispatcher` and the `File request`



```
public interface Send {
   public Object send(Object data);
}
<interface name="P2"
           role="server"
           signature="Send"/>
<interface name="P3"
           role="client"
           signature="Send"/>
```

Fig. 3. Definition of the `Cache` component



```
public interface URL {
   public String getPage(URL u);
}
```

Fig. 4. Excerpt definition of the `Request handler` composite

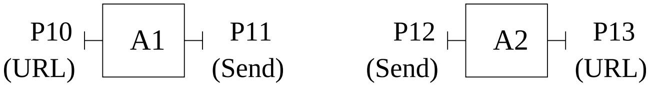


Fig. 5. Adapters

`handler` components to reduce the number of disk accesses performed by `File request handler`. This integration requires the modification of a connector.

The `Cache` component has two communication points `P2` and `P3`, as shown in Figure 3. Data is received through the communication point `P2`, which offers the `Send` interface, containing the method `send`. In Fractal ADL, `P2` is a server interface. If the data passed as the argument of the `send` method is already in cache, then the cached data is returned. Otherwise, the request is forwarded through the method `send` of the communication point `P3`, which returns the requested data. This data is cached for later requests and then sent back to the caller of `P2`'s `send` method.

To integrate the `Cache` component, the architect has to remove the connector between the `Request dispatcher` and `File request handler` components illustrated in Figure 4. The `Cache` component is placed into the `Request handler` composite. The communication points `P5` and `P6` are transformed so that they are *compatible* with `P2` and `P3`, respectively. To do so, adapter components are

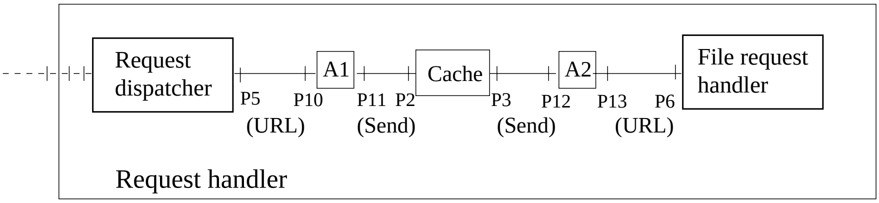


Fig. 6. Result of the integration of the `Cache` component

used to perform type conversion. The adapters `A1` and `A2`, shown in Figure 5, provide this feature. In Fractal, this amounts to creating two components `A1` and `A2`, such that `A1` has a server interface `URL`, named for example `P10`, and a client interface `Send`, `P11`, and `A2` has a server interface `Send`, `P12` and a client interface `URL`, `P13`. Finally, the adapters are placed in the `Request handler` composite and connectors are created between `P5` and `P10`, `P11` and `P2`, `P3` and `P12`, and `P13` and `P6`. Figure 6 illustrates the result of the integration of the `Cache` component.

3 The FIESTA Framework

Figure 7 provides an overview of the FIESTA framework. Our framework is directly inspired by TranSAT. Consequently, FIESTA relies on the creation of architecture integration patterns and a weaver. The weaver both determines where a pattern can be applied in a target architecture and performs the integration.

While TranSAT is dedicated to the SafArchie ADL, our approach is decoupled from any specific ADL. Being ADL-independent has required defining new means to express architecture integration patterns, as well as developing a generic integration engine. In this section, we present how to specify architecture integration patterns in our approach, as well as our generic architecture internal representation model and weaver, which are the key parts of our generic integration engine.

3.1 Architecture Integration Patterns

An architecture integration pattern describes a given functionality and contains all the information needed to integrate this functionality into an architecture. It may be developed independently of any specific architecture by an architecture integration pattern developer, and reused on different architectures by a software architect. In the FIESTA approach, an architecture integration pattern is composed of a new plan, a join point mask and a set of integration rules.

New Plan. A new plan is an assembly of components describing a functionality. For example, the `Logger` component (Figure 2) and the `Cache` component (Figure 3) can be considered as new plans. Some communication points of the assembly may not be connected yet. These points will be the points at which connectors will be attached to integrate the new plan into the base architecture, *i.e.,* the target architecture.

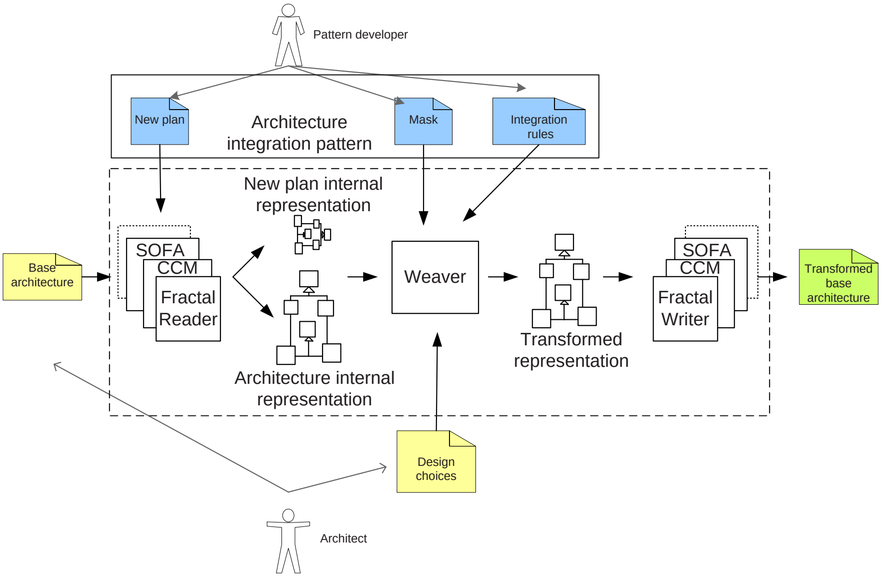

Pattern developer
New plan
Architecture integration pattern
Mask
Integration rules
New plan internal representation
Base architecture
SOFA
CCM
Fractal Reader
Weaver
SOFA
CCM
Fractal Writer
Transformed base architecture
Architecture internal representation
Transformed representation
Design choices
Architect

Fig. 7. Overview of the FIESTA framework

A new plan is described in a given ADL. It could be SafArchie, Fractal ADL or SOFA, *etc.* We have chosen not to specify the new plan with a generic ADL in order to keep all the expressiveness and the specific characteristics of each ADL, and to also not force the pattern developer to learn another ADL. The new plan will be automatically transformed into our generic architecture internal representation for use during the integration process. All the information described in the new plan will still remain in the architecture resulting from the integration.

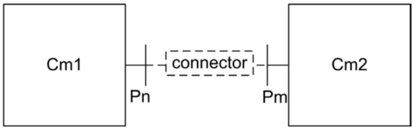
Join Point Mask. In our approach, a join point mask specifies a valid integration site as an abstract component assembly description. More precisely, it expresses structural properties that the base architecture must satisfy for the integration to be possible.

A join point mask is composed of component masks, connector masks and communication point masks. These mask elements abstract the general concepts of component, connector and communication point and thus are not specific to any ADL. Furthermore, communication point masks are constrained to be *linkable* with specific communication points of the new plan. Since the new plan is expressed in a given ADL, the resolution of the linkable property will be specific to this ADL and determined at integration time.

Figure 8 illustrates the mask to associate with the `Logger` component. This mask can match any existing component of the base architecture since the logging functionality may be added anywhere. At integration time, the architect will specify which component should be concerned by the integration of the logging functionality.

```
ComponentMask Cm1;
```

Fig. 8. Join point mask for the logging functionality



```
ComponentMask Cm1, Cm2;
CommunicationPointMask Pn on Cm1
          linkable with Cache.P2;
CommunicationPointMask Pm on Cm2
          linkable with Cache.P3;
ConnectorMask{Pn,Pm};
```

Fig. 9. Join point mask for the caching functionality

Figure 9 depicts the mask for the caching functionality. This mask specifies that in order to add caching the base architecture must contain two components connected through a connector. Implicitly, the connector may go through delegated communication points if the components are not within the same composite. The communication point masks `Pn` and `Pm` are declared to be *linkable* with the communication points `P2` and `P3` (Fig. 3), respectively.

At integration time, the weaver searches all the integration sites satisfying the mask and the architect specifies which of these sites to affect. Each abstract name associated to a mask element is then unified with the actual name contained in the architecture description.

Integration Rules. Integration rules are specified using integration primitives. FIESTA provides two integration primitives, one for each category of integration that we have identified during our domain analysis. We present these two primitives via the logging and caching examples presented in Section 2.

Adding of a new connector in the logging example is expressed using the primitive `addConnector` as follows:

```
addConnector Logger.P1 on Cm1;
```

This rule specifies that the communication point `P1` of the `Logger` component of the new plan must be connected to the component associated with the component mask `Cm1`. This rule implies that a compatible new communication point will have to be created on `Cm1` and a connector will have to be added between this communication point and `P1`. These operations will be automatically performed by the weaver. Design choices, such as choosing the name of the new communication point will have to be provided by the architect, he will be prompted by the weaver at integration time.

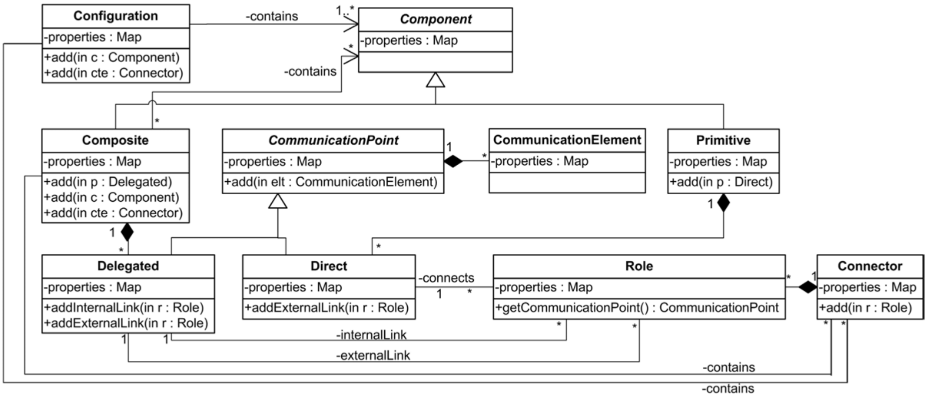


Fig. 10. Architecture internal representation model

Modifying an existing connector in the case of the cache example is expressed using the primitive `modifyConnector` as follows:

```
modifyConnector from Cm1.Pn to Cache.P2
            and from Cache.P3 to Cm2.Pm;
```

This rule indicates that the connector between `Pn` and `Pm` must be removed and replaced by a connector between `Pn` and `P2` and another connector between `P3` and `Pm`. Design choices, such as using adapters or directly modifying the communication points `P2` and `P3` to make the types compatible, are postponed until integration time.

3.2 The FIESTA Integration Process

Our domain analysis has shown that integrating a new functionality requires manipulating elements that are common across ADLs. Accordingly, we propose to build a generic integration framework rather than one framework for each ADL. To do so, our framework relies on both a generic architecture internal representation model to express a component assembly, and a generic weaver.

The FIESTA integration process is illustrated in Figure 7. A base architecture and a new plan both expressed in, say Fractal ADL, are transformed by an ADL-specific reader (in our case a Fractal ADL reader), each into an internal representation, according to our generic ADL model (Figure 10). The weaver transforms the architecture internal representation following the information contained by the architecture integration pattern. Finally, the transformed representation is translated by a ADL-specific writer back into a Fractal ADL specification.

Generic Architecture Internal Representation Model. Our internal representation, which results from our domain analysis, is described in Figure 10. To handle commonalities and variations between ADLs, our model relies on two sets

```
Test :                                                Introduction :
 bool isCompatible(CPoint p1, CPoint p2);              CPoint createCompatible(CPoint p);
 bool isAvailable (CPoint p);                          CPoint createComplement(CPoint p, List<CElt> elts);
 bool isConnectable(CElt e1, CElt e2);                 void add(CPoint p, Component c);
 bool isAddable(CPoint p, Component c);                void remove(CPoint p);

Reconfiguration :
 Connector getConnector(CPoint p1, CPoint p2);
 void makeConnection (CPoint p1, CPoint p2);
 void destroyConnection(List<CPoint> pts);
 Adapter createAdapter(CPoint p1, CPoint p2, List<CElt> elts);
```

Fig. 11. Low-level ADL-specific functions

of information: common structural information and ADL-specific non-structural information.

Since integrating a new functionality requires transforming the instance of the system structure at a given time, our model captures only the static structure of an architecture. The structural elements of the model are the ones identified in Section 2. Accordingly, a configuration is an assembly of components and connectors. A component may be a primitive component with direct communication points, or a composite component with delegated communication points. Communication points contain communication elements and are connected to connectors through roles. A composite component is formed of primitive and/or composite components, and connectors.

Our model takes into account the specific characteristics of each ADL and stores all the information contained in the original ADL description in order to be able to recreate it at the end. ADL-specific variabilities are stored into sets of properties, one set per structural element. These properties are represented as associations (key, value). For example, in the case of the communication point `P2` of the Cache component, a property ‘signature’ is associated with the value ‘Send’ and a property ‘role’ with the value ‘server’. Other information, such as the name of the file where an architectural element is specified in the original ADL description, is also stored.

Weaver. The weaver carries out the integration of a pattern with some input from the architect. First, it determines from the internal representation of the architecture the set of integration sites with respect to the join point mask of the pattern, *i.e.,* the set of component sub-assemblies of the architecture that match the structural requirements expressed by the join point mask. Then the architect chooses the sites to affect. For each selected site, the weaver first instantiates the mask elements specified in the integration rules with the base architecture elements forming a valid integration site, and finally executes the instantiated rules to integrate the new plan.

The integration site search, as well as each integration primitive, correspond to algorithms that are generic and thus independent of the ADL used to represented the base architecture. These algorithms rely on low-level operations. These operations are ADL-specific and have been provided once to configure our framework to support a given ADL.

```
// Fractal ADL
bool isCompatible(CPoint p1, CPoint p2) {
 return
  p1.getProperty(’signature’).equals(
                p2.getProperty(’signature’))
  &&
  ((p1.getProperty(’role’).equals(’client’) &&
     p2.getProperty(’role’).equals(’server’))
   ||
   (p1.getProperty(’role’).equals(’server’) &&
     p2.getProperty(’role’).equals(’client’))
   );
}
```

```
// CCM
bool isCompatible(CPoint p1, CPoint p2) {
 return
  p1.getProperty(’interface’).equals(
                   p2.getProperty(’interface’))
  &&
  ((p1.getProperty(’type’).equals(’facet’) &&
    p2.getProperty(’type’).equals(’receptacle’))
   ||
   (p1.getProperty(’type’).equals(’receptacle’) &&
    p2.getProperty(’type’).equals(’facet’))
   ||
   (p1.getProperty(’type’).equals(’event source’) &&
    p2.getProperty(’type’).equals(’event sink’))
   ||
   (p1.getProperty(’type’).equals(’event sink’) &&
    p2.getProperty(’type’).equals(’event source’))
  );
}
```

Fig. 12. Definitions of the function `isCompatible` for Fractal ADL and CCM

Low-level ADL-specific functions. The behavior of the algorithms related to the integration can be configured through the definition of 12 low-level operations. There are three kinds of operations: test, introduction and reconfiguration. The functions corresponding to these operations are shown in Figure 11.

Test functions are used to evaluate a priori whether a transformation implied by an architecture integration pattern can be performed without breaking the ADL semantics. For example, the function `isCompatible(p1,p2)` determines if the communication points `p1` and `p2` can be connected with each other without using an adapter. Specifically, in Fractal ADL, two communication points are compatible if they have the same signature and if one is declared as client and the other as server. In CCM, the notion of compatibility amounts to checking that both communication points have the same interface and that one is a facet (resp. event source) and the other a receptacle (resp. event sink). These two definitions of the operation `isCompatible` are shown in Figure 12.

The three other test functions, `isAvailable`, `isConnectable` and `isAddable`, are respectively used to check if a given communication point can accept another connection; if two communication elements `e1` and e2 can be connected so that there is no information loss between the two communication elements; and if a communication point `p` can be attached to a component `c`.

Introduction functions modify the interface of the components. The function `add` attaches a given communication point to a component, the function `remove` detaches a communication point from its parent component, the function `createCompatible` generates a communication point that is *compatible* with a given communication point, and the function `createComplement` clones a given communication point from which some communication elements have been removed.

Reconfiguration functions modify the interactions between components. They provide the capacities to obtain an appropriate connector, to connect two communication points, to destroy the connection between two communication points, and to create a specific adapter.

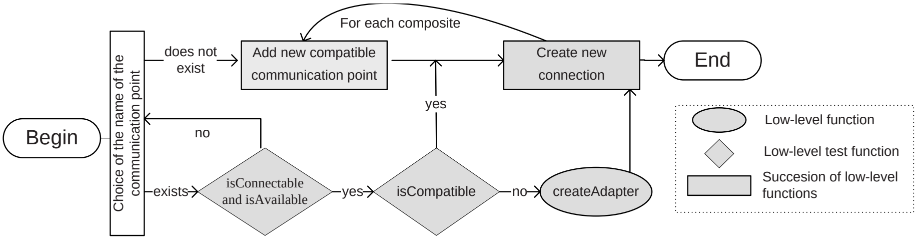

For each composite
Begin
Choice of the name of the communication point
does not exist
Add new compatible communication point
Create new connection
End
no
yes
exists
isConnectable and isAvailable
yes
isCompatible
no
createAdapter
Low-level function
Low-level test function
Succesion of low-level functions

Fig. 13. Generic algorithm for the integration primitive `addConnector`

Generic algorithms. Most of the code of the integration tool is generic and thus ADL-independent. The behavior of the algorithms related to integration only vary where the ADL-specific functions are called. These functions are used in both the algorithms to search integration sites and the ones to perform the integration. For example, the function `isConnectable` is used to resolve the *linkable* property that may be declared in a join point mask, enabling the identification of the integration sites. It is also used in the algorithms associated with the two integration primitives.

During the execution of the generic algorithms, some design choices have to be made by the architect. Consequently, the weaver may ask the architect to name new communication points, to decide whether adapters should be created or communication points directly modified, to choose the type of the connector, or also to decide in which composite a new component should be placed. The number of design choices depends directly on the ADL used, as some ADLs offer more possible actions than others. Nevertheless, the architect can also specify default design choices in a file so that he will not be prompted during the integration process.

As an example, Figure 13 gives an overview of the generic integration algorithm associated with the integration primitive `addConnector P on C`. If the name of the communication point provided by the architect exists on the component `C` then tests are performed to detect whether a connection can be created between this communication point and `P`. If needed, an adapter is introduced and all the connections are created, delegated communication points being recursively added as composites are crossed. Otherwise, if the name does not exist, then an appropriate communication point is created, then connected to `P`.

Implementation Status. The current implementation of our framework has been developed in Fractal and fully supports integrating functionalities for architectures expressed in Fractal ADL and SOFA. FIESTA can be extended to support a new ADL by defining the 12 ADL-specific functions shown in Figure 11. We have expressed these functions for Fractal ADL, SOFA, Wright, CCM and Unicon, and found that most of these functions are very similar to specify. Furthermore, a reader and a writer for the new ADL must also be developed, however, they can be most of the time adapted from the existing tools associated with this ADL.

4 Related Work

The TranSAT approach [6,7] introduced the concept of architecture integration pattern to modularize a given functionality and specify under which conditions the functionality can be integrated as well as the associated transformation rules to perform the integration. Our approach is built on the same concept but is more generic since our join point mask specification and integration rules are decoupled from any specific ADL. Furthermore, we provide higher-level abstractions to express how to integrate a new functionality than the TranSAT transformation rules, which remain low-level. Our approach reduces the risk for errors on the part of the pattern developer and also simplifies the verification of the coherence of the integration pattern.

Some works, such as Acme [12] or xADL [10], have proposed a generic ADL model to represent different ADLs. Though these two models have commonalities with ours, they do not provide enough details about communication between components. For example, they do not offer the notion of communication elements and direct or delegated communication points, which are required for our purpose. Furthermore, it is not possible to configure these generic ADLs to add a given semantics, consequently manipulating the generic representation requires the tool to know about the underlying ADL semantics, which complicates the support of new ADLs. For example, the verification of compatibility between two communication points must be done by external tools specific to each ADL.

Some ADLs enable the architect to extend their architecture description with architectural constraints, but to do so they impose to use a given formalism, such as invariants in ACME/Armani [13]. Our model can also be extended to support architectural constraints but does not impose any given formalism. Indeed, contraints are stored in our model within the component properties. Thus adding support for a new formalism, such as invariants or OCL, does not require any modification of our model. The associated verifications only have to be specified once and for all in the low-level functions, to be performed at each modification of the architecture.

Some approaches to dynamic adaptation enable the modification of an architecture [1,11,14]. However they only rely on low-level functions that are ADL-dependent. The architect must thus specify all the required steps to add a new functionality. For example, in [11] the architect has to write each time the adaptation operations that must be performed, depending on the rules that govern how the architectural elements may be composed. Our work is at a higher level of abstraction since we provide two transformation rules that are ADL-independent and that are automaticaly transformed into the appropriate sequence of low-level function calls.

5 Conclusion and Future Work

We have presented FIESTA, a generic framework that enables the integration of new functionalities into an architecture description. Our approach is built on a domain analysis that has led to the identification of the architectural elements involved in the integration process, independently of the ADL used, which has enabled the definition of a generic ADL model. Our approach is thus decoupled from any given ADL. Furthermore, we have provided higher-level abstractions to describe the join point mask and transformation rules of an architecture integration pattern than the TranSAT approach, simplifying the task of the pattern developer. Our weaver carries out the integration of new functionalities, making easier the task of the architect. The architect only has to decide which integration sites to affect and to provide information related to design choices.

Our integration engine is generic and can be configured to support new ADL through the specification of 12 low-level functions. Currently, we have defined these low-level functions for several of the ADLs we have studied and have fully implemented our framework for Fractal ADL and SOFA. In the near future, we plan to handle more ADLs, such as SafArchie. This will require extending our model and rules in order to capture and manipulate SafArchie behavioral information.

Adding support for a new ADL requires also to develop a reader and a writer. As the writer is in charge of converting the generic internal architecture representation corresponding to an instance of the architecture back into its original ADL, it may have to rebuild the types associated with the element instances if this is required by the ADL. To ease this task, we plan to work on the design of a generic type model to represent the types of the elements of our generic ADL model.

Furthermore, we would like to apply our approach to the development of more applications to better evaluate our proposition. Finally, we are interested in investigating the possibilities to adapt our approach in order to provide functionality integration capabilities at runtime.

References

1. Abdelmadjid, K., Noureddine, B.: Open framework for the dynamic reconfiguration of component-based software. In: SERP, pp. 948–951 (2004)
2. Allen, R.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon, School of Computer Science, CMU-CS-97-144 (January 1997)
3. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis & Design Language (AADL). SAE Standards nAS5506 (November 2004)
4. Balter, R., Bellissard, L., Boyer, F., Riveill, M., Vion-Dury, J.-Y.: Architecturing and configuring distributed application with Olan. In: Proceedings of the 1st IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, The Lake District, UK, pp. 241–256. Springer, Heidelberg (September 1998)
5. Barais, O., Duchien, L.: SafArchie studio: An ArgoUML extension to build safe architectures. In: Architecture Description Languages, pp. 85–100. Springer, Heidelberg (2005)
6. Barais, O., Duchien, L., Le Meur, A.-F.: A framework to specify incremental software architecture transformations. In: EUROMICRO-SEAA'05. Proceedings of the

31st EUROMICRO Conference on Software Engineering and Advanced Applications, Porto, Portugal, pp. 62–69. IEEE Computer Society, Los Alamitos (September 2005)

7. Barais, O., Lawall, J., Le Meur, A.-F., Duchien, L.: Safe integration of new concerns in a software architecture. In: ECBS'06. Proceedings of the 13th Annual IEEE International Conference on Engineering of Computer Based Systems, Potsdam, Germany, pp. 52–64. IEEE Computer Society, Los Alamitos (March 2006)
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: An open component model and its support in Java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
9. Bruneton, E., Coupaye, T., Stefani, J.-B.: The Fractal Component Model (February 2004), online documentation `http://fractal.objectweb.org/specification/`
10. Dashofy, E.M., der Hoek, A.V., Taylor, R.N.: A highly-extensible, XML-based architecture description language. In: WICSA'01. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, p. 103. IEEE Computer Society, Washington, DC, USA (2001)
11. Garlan, D., Cheng, S.-W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. Architecting Dependable Systems (2003)
12. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: CASCON'97. Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research Processing, Toronto, Ontario, Canada, pp. 169–183 (November 1997)
13. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
14. Hillman, J., Warren, I.: An open framework for dynamic reconfiguration. In: ICSE, pp. 594–603 (2004)
15. Kalibera, T., Tůma, P.: Distributed component system based on architecture description: The SOFA experience. In: Meersman, R., Tari, Z., et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, pp. 981–994. Springer, Heidelberg (2002)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
17. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, Springer, Heidelberg (1995)
18. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
19. Object Management Group: CORBA Component Model, v3.0, formal/2002-06-65 (June 2002)
20. OMG: Unified Modeling Language (UML): Superstructure. 2.0 edn. Needham, MA, USA (August 2005)
21. Zelesnik, G.: The UniCon Language Reference Manual. School of Computer Science Carnegie Mellon, Pittsburgh, Pennsylvania (May 1996)

Beyond ATAM: Architecture Analysis in the Development of Large Scale Software Systems

Andrzej Zalewski

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

Abstract. Large scale software developments require substantial investment and are exposed to high level of risk. Architectural decisions taken at early stages of the development can substantially influence the entire level of technical risk. In this paper architectural decisions are divided into two basic groups: early – top level system organization decision establishing system organization patterns (the notion introduced in this paper) and detailed ones. However as it was shown on notable examples of large scale developments carried out in recent ten years in Poland, wrong decisions concerning system organization pattern can trigger severe risks that can lead to the development crisis. These risks are frequently connected with the complexity explosion syndrome – sudden, undetected growth of design complexity that exceeds the capabiblity of the development team and time budget. To manage these risks properly appropriate architecture analysis method has been introduced. On the contrary to the traditional scenario-based architecture analysis methods, like ATAM, it was based on GQM approach. A complete assessement framework have been defined comprising three goals: complexity control, organizational adequacy and satisfactory perforormance and reliability; a set of questions related to these goals, as well as metrics for the qualities expressed by these questions. The conclusion contains *ex post* analysis of exemplary large scale systems showing that the proposed framework provides adequate assessement of design risk. It has also been indicated that the critical risks identified during the evalution of the system organization pattern should be carefully managed.

1 Introduction

Architecture Trade-off Analysis Methods (ATAM) [4], being a successor of early architecture analysis methods like SAAM [8], SAMCS, ESAAMI, SAAMER (a complete survey – [1]), represents the state-of-the-art in architecture analysis with quite a broad record of applications e.g. [14], [15], [16], [17], [19]. ATAM is based on specifying quality requirements in terms of quality scenarios. In large scale developments early architectural decisions are fundamental to the success of the whole project and should be assessed as early as it is possible. However, as we show in the course of this paper, scenario-oriented architecture analysis methods can hardly be applied to the early architectural design of large

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 92–105, 2007.

scale developments. Therefore, the paper is aimed at supplementing existing approaches so that early architectural design could be effectively assessed.

The paper starts from the introduction and description of the following notions: *large scale software system*, *system organization pattern* (result of early architectural decisions) and phenomenon of *complexity explosion syndrome.* On that basis the limitations of ATAM as a technique for analysing system organization pattern have been discussed. As a solution for these shortcomings a novel approach for the evaluation of system organization pattern has been proposed. The architecture evaluation is performed by the use of adopted and specialized GQM technique. The adequacy of proposed assessement approach has been verified on the sample of large scale software system developments.

2 Large Scale Software Systems

The class of large scale software systems can be distinguished by the following technical features:

- *Strong geographical distribution* – large scale software is typically built on the commission of country- or world-wide public or private organizations.
- *Large volumes of data being stored and processed.*
- *Large number of concurrent users being served* – the number of users usually reflects the number of organization's employees and typically exceed 1000 users.
- *Distribution of computational and data resources* – results from the geographical distribution feature; the main concern here is the allocation of storage and processing to the locations as well as organization's units.
- *Managing large distributed data resources* – distribution of data resources (distributed databases, local databases in thick-client architecture) usually require to resolve issues of data synchronization, transactions and data exchange between system units.
- *Long and short distance communication solutions* assuring data flow between system's units – communication mechnisms, protocols, message passing schemes, etc.

3 System Organization Pattern

Architecture development may be perceived as a series of architectural decisions (compare: [20], [21], [22]) concerning different level of scope and obviously different level of detail: from the very top enterprise level to a very detailed software components level [23]. In this paper we focus on the early architectural decisions, which play particularly important role in the development of large scale software systems. These top level architectural decisions have been referred to as *system organization pattern* and comprise the following basic elements:

- *Decomposition into a set of subsystems/applications* – defines a coarse grained functional decomposition. Subsystems are here meant as single applications or suites of the applications accompanied by a common infrastructure (e.g. databases, application servers), prepared for a single functional domain. Each application or a subsystem encompasses a portion of system's functionality. The allocation of functionality is a crucial aspect of the decomposition onto a set of subsystems/applications.
- *Geographical and organizational allocation of subsystems/applications* – the structure of the system has to reflect the structure and geographical— disctribution of a taget organization. System organization pattern provides information on the organization's units for which certain applications are provided as well as information on the geographical situation of these units and corresponding subsystems/applications.
- *Organization of data input* – concerns defining and allocating points, where external data is entered into the system. This includes emplacement of the system remote access points, data entry (typing) points, scanning devices etc.
- *Organization of data storage and processing*:
  - *Data storage distribution* – defines the distribution of permanent data resources (typically databases),
  - *Data processing organization* – is basically about choosing data processing architecture (client-server, multi-tier, thin/thick client).
  - *Distributed data storage management* – concerns the means of synchronization of distributed databases, transmitting data onto remote systems, upload of data from local to main data store.
  - *Transaction management* – regards selection of solutions assuring transactionality. This involves both short and long transactions, nevertheless, short ACID [13] transactions are offered by the database engines, which is not sufficient for long ones, which require specialized advanced solutions like transaction monitors.
- *Communication framework* – defines the communication mechanisms between equivalent system entities (e.g. processes, applications), communication protocols used to transmit data between system entities.

To ilustrate the concepts described above let us consider parcel/mail movement tracking system discussed further in p. 5. The system has been suited for five regional mail/parcel distribution centers of national post located in five largest cities of Poland. The centers are identical in terms of their organizational structure. The draft of system organization pattern is presented below:

- *Decomposition into a set of subsystems/applications* – subsystems handling automatic mail sorting machinery as well as users applications have been defined.
- *Geographical and organizational allocation of the subsystems/applications* – structurally the same subsystems and applications have been deployed at each of the five distribution centers.

- *Organization of data input* – data inputs are: sorting machines and users terminals – all located at the same premises.
- *Organization of data storage and processing*:
  - *Data storage distribution* – relational database management systems have been provisioned for each center.
  - *Data processing organization* – applications/subsystems follow the client-server pattern.
  - *Distributed data storage management* – data is transmitted to remote locations by a proven, commercial message queueing system.
  - *Transaction management* – there is only necessity to handle short transactions only.
- *Communication framework* – the messege queueing system communicates over IP network.

The decisions listed above are to smaller or greater extent present in every software development. In large scale software developments, all of them are vital and not trivial. The notion of 'pattern' has been applied as the elements of system organization pattern are multiplicated in the system's structure, e.g. if it's been decided to have local databases servicing local client software for each company's location this architectural pattern will be multiplicated in all the locations.

4 Complexity Explosion Syndrome

In recent ten years there have been a number of large scale software systems developments in Poland. These projects have been carried out on the commission of governmental and private financial institutions as well as of large industrial enterprises. Some of these developments encountered serious problems that lead to crisis. These situations can be attributed to the negligence of various classes. To the most severe ones belongs the lack of appropriate architecture evaluation at early design phases together with inefficient risk management.

Faulty system organization pattern causes unexpected, sudden, often undetected explosion of software complexity, which cannot be managed by the development team. This phenomenon has been referred to as *complexity explosion syndrome*. It usually affects critical elements of the whole design and can lead to general development crisis if it is not managed properly. The inability to cope with the design complexity is typically a result of a mismatch between the software construction problems that have to be solved and the experience and competences of the development team as well as time available for project completion.

Let us consider two notable examples of complexity explosion syndrome – these are software crisis stories.

PR.1. Election support system – the system consisted of central subsystem situated at the headquarters' premises and local systems located in the middle-level local electoral authorities. Low-level electoral authorities are involved in direct vote collection and summation providing with the paper protocols, while middle-level ones aggregate data provided by the low-level authorities. It was provisioned for an extensive communication between middle-level and headquarters's subsystems for the purpose of the synchronization of fragments of databases. The following architectural decisions turned out to be wrong:

- Synchronous communication between local and central systems – this caused local systems to hang waiting for the reply from overloaded headquarters.
- Decision to implement synchronization mechanism almost from scratch, while the development team had rather little experience in such a field.
- The low cohesion of a local application – its operation depended on the successful upload of data from the headquarters.
- High coupling between local and central subsystems – proper operation of each depended on the proper operation of the other system.

PR.2. Pipeline passportization system was commissioned by the operator of the gas pipeline system. The system consisted of a central subsystem located at company's headquarters and autonomous local systems. In this case the following architectural decisions turned out to be misleading and lead to the complexity explosion:

- Data processing organization – thick local client solution was chosen. This introduced processing of long transactions.
- Transaction management – it was implemented using conventional framework provided by the database engine for short transactions – this caused other users applications to hang on blocked rows of the database, these in turn imposed their blocks on other parts of the database, users restarted hung applications leaving the blocks on the database – the system was perceived as unable to operate properly.

What was missing in the above examples, was:

- Architecture analysis of system organization pattern and further architectural design models.
- Risk analysis based on the findings of early architectural design as well as further risk monitoring and control techniques supplied with the information resulting from the architecture analysis.

The phenomenon described above has both rich history and supposedly bright future as the developments in software design paradigms, methods and tools provide with newer and newer risks – e.g. SOA, being currently a buzzword in the commercial IT world, provide with a number of pitfalls causing high development risk like missing long transaction solution that has to be supplemented by the developers (for more see: [7]).

5 Other Examples of the Impact of System Organization Pattern on the Development Process

Complexity Explosion Syndrome is the most severe consequence of wrong early architectural decisions. These may lead to less serious problems (see below: problem story) than the latter ones. However, right early architectural decision will provide a sound foundation for further development (see below: success story).

Problem story: PR.3. Medical services registration – the system was developed on the commission of national health care found, its architecture was similar to the architecture in the crisis examples – the system consisted of local and central subsystems, data (information on the executed medical services) was to be uploaded from local data storage (systems placed in hospitals, surgeries, ambulatories, etc.) to central subsystem. It was chosen that communication would be based on SMTP protocol. The solution was aimed at providing data queuing without introduction of specialized software (like IBM MQ Series). It turned out in the course of exploitation that the communication is unreliable as well as the development team could not fully cope with SMTP (Simple Mail Transfer Protocol) services properly. This resulted in replacing self-designed SMTP transfer services with manual solutions: the data was transferred onto CD-W, sent via conventional post to the headquarters, and uploaded there manually from the provided CD.

Success story: PR.4. Parcel/mail movement tracking – the system has been designed for the mail distribution centers of the Polish Post (Poczta Polska). It was aimed at transferring data connected with the movement of mail and parcels between the distribution centers. The data communication had bursty characteristics – large data loads had to be transferred when transport of the mail was leaving for or arriving at another distribution center. The transport of the mail or parcel was to be accompanied by the transfer of the appropriate information to the destinate distribution center – to provide information on the expected mail transport – as well as delivery confirmation was expected to be sent back to the source distribution center. Identical subsystems have been placed in every distribution center, the information transfer was based on asynchronous queuing solutions from IBM: MQSeries (now: IBM WebSphere MQ). The system turned out to operate successfully – the data was timely and consistently transferred between the distribution centers.

6 Limitations of Scenario-Oriented Architecture Analysis Methods

Architecture analysis methods started to evolve in the middle nineties – SAAM [8] appeared in 1994. Even in 2000 it was said in [1] that ATAM [4], which now seems to be the most mature architecture analysis method, was still under research. This sheds light on the reasons of the lack of architectural analysis in the developments carried out during last ten years in Poland.

Although architecture analysis methods have developed considerably since the beginning of 21^{st} century, there is still much to do about analysing architecture at early design phases. In ATAM software architecture is analysed from the software quality attributes point of view [4], [5]. The quality requirements are expressed by ‘quality scenarios’. A quality scenario consists of a chain of: source – stimulus

– artifact – response – response measure. Quality scenario patterns have been proposed in [4] for the following quality attributes: availability, modifiability, performance, security, testability, usability.

However, system organization pattern is hardly amenable to scenario-oriented analysis due to the following reasons:

1. In the ATAM software architecture is analysed from the point of view of software quality attributes like those defined in ISO 9126 [3]. However, system organization pattern can hardly be related to the final software properties making accurate and meaningful evaluation hard to achieve.
2. The expression power of quality scenarios is that quality attributes are expressed in terms of concrete expectations of project stakeholders. However, these can be successfuly expressed and evaluated when the requirements and design is sufficiently detailed, which is usually not the case of system organization pattern of a large scale software system.
3. Although there exist numerous case studies on ATAM applications – at least 18 case studies carried out by SEI of Carnegie Mellon University mentioned in [18] (e.g. [14], [15], [16], [17]) and other evaluations reported in conference papers like [19] – there is rather a sparse record of using ATAM to assess early architectural desing. These analyses use usually 4+1 views of software architecture (or its subset). This confirms indirectly the premises given above; compare also a survey [1].

A novel approach based on the features distinct from the final software quality attributes is need to cope with system organization pattern analysis. Thus, the task is to:

– define attributes applicable for the assessment of system organization pattern,
– provide evaluation technique,
– define the whole assessment framework consisting of evaluation techniques, attributes being assessed, definition of stakeholders involvement, architecture models used as a basis for the evaluation.

7 A Framework for the Assessment of System Organization Pattern

Apart from the software quality attributes, the qualities of architecture itself are also studied in literature. In [5] there have been mentioned conceptual integrity, correctness, completeness and buildability as features of an architecture itself. These features are generally difficult to define in a quantitive manner. Except for buildability, it is also difficult to define them qualitatively.

Basing on the large scale software systems listed in p. 4 and 5 it was concluded that early architectural design expressed in the system organization pattern can affect the ability to achieve the following goals:

- *Complexity control* – software architecture is one of the means of managing software complexity. Therefore, the goal of complexity control reflects the ability of a given architectural design to keep system's complexity under control avoiding unexpected complexity growth or explosion. It is notable that the more detailed architectural decisions are, the smaller parts of the system they affect – therefore, detailed architectural decisions cannot lead to complexity explosion, while early ones can.
- *Organizational adequacy* – provides assurance that system architecture is in line with the internal structure and geographical distribution of a given organization.
- *Satisfactory performance and reliability* – are meant traditionally reflecting the ability of assuring timely processing of the external requests (user, data loaded onto the system, etc.) as well as proper and uninterrupted system operation.

Goal, Question, Metric (GQM) [9] approach has been adopted for the evaluation of system organization pattern.

7.1 Evaluation Method

The evaluation methods was based on GQM and comprises of:

- *goals* – expected attributes of the software architecture,
- *questions* – characterizing software architecture in the context of expected attributes,
- *metrics* – expressing quantitatively attributes connected with questions.

Below the adaptation of GQM framework to the assessment of system organization pattern has been presented. It is organized as follows: for each of the three goals there have been defined set of questions, for each of these questions one or more measures have been indicated. Scales for the measures have also been provided if it was necessary: the metric values have been ranked ascending (from the worst to the best). Critical values for each of the metrics have also been defined – they indicate existance of a critical risk.

GOAL: Complexity Control

- Decomposition into a set of subsystems/applications [10]
  - **DSA.01.** Are the specified subsystems/applications functionally consistent?
    Metric: level of cohesion.
    Scale: Coincidental, Logical, Temporal, Procedural, Communicational, Sequentional, Functional.
    Critical values: Coincidental, Logical.
  - **DSA.02.** Are the specified subsystems/applications sufficiently independent?
    Metric: level of coupling.
    Scale: Content, Common, Control, Stamp, Data.

  Critical values: Content, Common, Control.
  Comment: the metrics of coupling and cohesion have been proposed in the 70s in Stevens et al. [10] and Yourdon and Constantine [11]. The scales given below stem from [10] and are equally useful when subsystems or applications comprising a single large scale system are considered as software modules.

- Data storage distribution
  - **DSD.01.** Does the data storage distribution together with functional decomposition imply the necessity of communicating data between databases?
    Metric: level of database dependence;
    Scale: High – there are data storages dependent of each other – synchronization or upload/download is necessary; Low – databases are independent or there is a single database, thus synchronization between databases is not needed.
    Critical values: High.
- Data processing organization.
  - **DPO.01.** Is advanced local data storage needed for client application?
    Metric: kind of client application.
    Scale: Thick, Hybrid, Thin.
    Critical values: Thick.
    Comment: only thick client contains advanced local data storage (e.g. relational database), which can give rise to the problems of long transactions and/or data synchronization between local and central storages.
  - **DPO.02.** What is the duration of transactions connected with managing data storage of a client application?
    Metric: Length of transactions.
    Scale: Unknown, Long, Short.
    Critical values: Unknown, Long.
    Comment: this question encompasses the case of client storage in a multi-tier application.
  - **DPO.03.** Will 'central' data be uploaded/downloaded from/onto local applications/subsystems?
    Metric: existence of the need for local data upload/download.
    Scale: Exists / Not existing
    Critical values: Exists.
- Management of distributed data storage.
  - **DSM.01.** What is the confidence in the chosen database synchronization solutions?
    Metric: level of confidence.
    Scale: Low – the synchronization will be implemented entirely by the development team; High – the synchronization will base on proven commercial or proven open source solution.
    Critical values: Low.
- Transaction management framework.

    - **TMF.01.** What is the confidence in the selected long transaction management framework?
      Metric: level of confidence.
      Scale: none – the problem of long transactions has not been noticed, low – the long transaction management will be designed by the development team, high – proven solution will be used (e.g. commercial transaction monitor)
      Critical values: none, low.
- Communication framework.
    - **CF.01.** Does the development technologies of choice provide with uniform communication framework assuring data exchange between equivalent software units (e.g. processes, distributed objects, etc.)?
      Metric: Existence of uniform communication mechanisms for the whole software.
      Scale: No, Yes.
      Critical values: No.
    - **CF.02.** Is the communication between system entities uniform from the software developer point of view?
      Metric: number of protocols that developer has to be aware of?
      Critical values: greater than one.

GOAL: Organizational Adequacy

- Geographical and organizational allocation of the subsystems/applications.
    - **GOA.01.** Is the the allocation of subsystems/applications to organization's units adequate to the organization structure?
      Metrics: a) Number of organizational units, where necessary functions are unavailable; b) number of organizational units to which superfluous applications/subsystems have been allocated.
      Critical value: a) any higher than 0; b) not specified.
    - **GOA.02.** Is the geographical allocation of subsystems/applications adequate to the geographical distribution of a given organization?
      Metrics: a) Number of locations, where necessary functions are unavailable; b) number of locations where superfluous applications/subsystems have been allocated.
      Critical value: a) any higher than 0; b) not specified.
- Organisation of data input
    - **ODA.01.** Are the physical data input points (scanning devices, typing stations) placed at the points nearest to the sources of entered documents?
      Metrics: Time need to move the documents from source to the data input point, distance between source and data input points.
      Critical value: to be established individually.
    - **ODA.02.** Is the electronic data input accessible to the users or systems entering data?
      Metrics: share of users or external systems that cannot input data electronically being adequately equipped.
      Critical value: to be established individually.

GOAL: Satisfactory Performance and Reliability

- The questions below refer by analogy to the respective system organization pattern artifacts indicated above.
  - **TMF.01.** – defined earlier
  - **DSM.02.** Is the communication mechanism planned for distant database synchronization capable of coping with varying intensity of data load?
    Metric: Level of suitability for varying data load.
    Scale: Weak – only synchronous communication has been planned; Strong – asynchronous communication including queuing services have been provisioned for.
    Critical values: Weak.
  - **DPO.02.** Defined earlier.
  - **CF.03.** Do the communication mechanisms provide sufficient communication reliability?
    Metric: failure ratio, ratio of the number of negative application report to the total number of applications.
    Critical value: to be established individually.

7.2 Stakeholders Involvement in the Evaluation

Because early architectural decisions are vital to the success of the entire software development it is recommandable that all stakeholders should take part in the system organization pattern evaluation: developing organization's management, end users, maintenance organization, customer (compare: [5]). The system organization pattern should be formally aproved by the stakeholders.

7.3 Architecture Models Used as a Basis for the Evaluation

The proposed evaluation method does not require any particular form of the documentation of the system organization pattern. The latter is typically a text document, sometimes accompanied by the diagrams illustrating the logical, physical and process view of the architecture (compare: [5]).

8 Analysing the Risk of Failure: A Case Study

The evaluation can take a tabular form shown in the tables 1, 2 presenting evaluation of projects characterized in p. 4 and 5 against the goals of 'complexity control' and 'satisfactory performance and reliability'. The bold typeface has been used to indicate the critical metric values.

The number of critical metric values for the questions listed above for PR.1 and PR.2 indicates high level of design risk (most of the question metrics have critical values). The congestion of critical values indicates the risk of complexity exlosion syndrome or performance / reliability pitfalls. On the contrary to the latter, successfull project PR.4 indicates well-chosen architectural decisions (all the question metrics have non critical values).

Table 1. Evaluation of complexity control goal attribute of the developments presented in p. 4 and 5

Question	*PR*.1	*PR*.2	*PR*.3	*PR*.4
DSA.01	**Logical**	Functional	Functional	Functional
DSA.02	**Content**	Data	Data	Data
DSD.01	**High**	**High**	**High**	Low
DPO.01	Hybrid	**Thick**	Hybrid	Hybrid
DPO.02	Short	**Long**	Short	Short
DPO.03	**Exists**	**Exists**	**Exists**	Not exist
DSM.01	**Low**	**Low**	N/A	High
TMF.01	N/A	**None**	N/A	N/A
CF.01	**No**	Yes	Yes	Yes
CF.02	1	1	1	1

Table 2. Evaluation of the sufficient performance and reliability goal of the developments presented in p. 4 and 5

Question	*PR*.1	*PR*.2	*PR*.3	*PR*.4
DSA.01	**Logical**	Functional	Functional	Functional
DSA.02	**Content**	Data	Data	Data
TMF.01	N/A	**None**	N/A	N/A
DSM.02	**Weak**	N/A	Strong	Strong
DPO.02	Short	**Long**	Short	Short
CF.03	High	High	Low	High

9 Managing Critical Design Risks

All the critical values identified during the system organization pattern evaluation indicate a critical design risk, which may result in an unexpected complexity growth or exposion. This risk information should become input to the risk management procedures and carefully monitored during the whole course of the project. For each of these risks appropriate mitigation tactic should be applied – e.g. the risk associated with coping with the long transactions can be mitigated by the introduction of a commercial transaction monitoring solution or enhancing development team with appropriate special field experts.

10 Conclusion

The paper was devoted to the problem of early architectural design assessement. The notion of system organization pattern comprizing early architectural decisions have been introduced and defined. The phenomenon of complexity explosion observed in the large scale software projects have been defined as well.

A novel assessement approach, basing on the GQM evalution technique destinated to the assessement of the system organization pattern have been proposed. Therefore, a set of top level architectural decisions has been defined together with a method for the assessement of these decisions. The proposed approach has been successfully evaluated *ex post* on real life large scale software developments.

The qualities of system organization pattern evaluated in the proposed framework belong to the class of 'ever-green problems': coupling/cohesion, transaction management, database synchronization, etc. These problems may recur as newer and newer software technologies evolve perpetually.

The proposed GQM architecture evaluation framework can easily be expanded with another goals, questions and metrics. It is also recommandable to perform such an *ex post* evaluation on a greater number of large scale software projects. This could form an extensive project assessments database. It would enable to identify on a representative sample risk patterns typical for certain types of design failures. Such a repository would become a valuable source of information to the software architects.

References

1. Dobrica, L., Niemelä, E.: A Survey on Software Architecture Analysis Methods. IEEE Transactions on Software Engineering 28(7), 638–653 (2002)
2. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
3. ISO: IEC: ISO/IEC 9126-1:2001 Software engineering —- Product quality —- Part 1: Quality model. ISO/IEC (2001)
4. Kazman, R., Klein, M., et al.: The Architecture Tradeoff Analysis Method. In: ICECCS'98. Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems, pp. 68–78. IEEE Computer Society Press, Los Alamitos (1998)
5. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (2003)
6. Albin, S.T.: The Art of Software Architecture: Design Methods and Techniques. John Wiley & Sons, Chichester (2003)
7. Kaye, D.: Loosely Coupled: The Missing Pieces of Web Services. RDS Press (2003)
8. Kazman, R., Bass, L., et al.: SAAM: A Method for Analysing the Properties of Software Architectures. In: Proceeding of 16th Int'l Conf. Software Engineering, pp. 81–90 (1994)
9. Basili, V.R., Caldiera, G., Rombach, H.: The Goal Question Metric Paradigm. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1, pp. 578–583. John Wiley & Sons, Chichester (1994)
10. Stevens, W.P., et al.: Structured design. IBM Systems Journal 13(2), 115–139 (1974)
11. Yourdon, E., Constantine, L.L.: Structured Design. Yourdon Press (1978)
12. Gallagher, B.P.: Using the Architecture Tradeoff Analysis Method to Evaluate a Reference Architecture: A Case Study SEI. Technical Note CMU/SEI-2000-TN-007. Carnegie Mellon University (June 2000)

13. Garcia-Molina, H., Ullman, J.D., Widom, J.D.: Database Systems: The Complete Book. Prentice-Hall, Englewood Cliffs (2001)
14. Kazman, R., Barbacci, M., et al.: Experience with Performing Architecture Trade-off Analysis. In: Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, pp. 54–63 (May 1999)
15. Bergey, J.K., Fisher, M.J.: Use of the Architecture Tradeoff Analysis Method (ATAM) in the Acquisition of Software-Intensive Systems SEI. Technical Note CMU/SEI-2001-TN-009. Carnegie Mellon University (September 2001)
16. Jones, L.G., Lattanze, A.J: Using the Architecture Trade-off Analysis Method to Evaluate a Wargame Simulation System: A Case Study. SEI. Technical Note CMU/SEI-2001-TN-022. Carnegie Mellon University (December 2001)
17. Clements, P., et al.: Using the SEI Architecture Tradeoff Analysis Method to Evaluate WIN-T: A Case Study Technical Note CMU/SEI-2005-TN-027, Carnegie Mellon University (September 2005)
18. Barbacci, M., Nord, R., et al.: Risk Themes Discovered Through Architecture Evaluations. TECHNICAL REPORT CMU/SEI-2006-TR-012 ESC-TR-2006-012. Carnegie Mellon University (September 2006)
19. Boucké, N., Weyns, D., et al.: Applying the ATAM to an Architecture for Decentralized Control of a Transportation System. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 180–198. Springer, Heidelberg (2006)
20. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: WICSA05. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press, Los Alamitos (2005)
21. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software, 19–27 (March-April 2005)
22. Malan, R., Bredemeyer, D.: Software Architecture: Central Concerns, Key Decisions, `http://www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF`
23. Malan, R., Bredemeyer, D.: Less is More with Minimalist Architecture. IEEE IT Professional 4(5), 46–48 (2002)

Enabling *Adaptivity* in User Interfaces[*]

Javier Cámara[1], Carlos Canal[1], Javier Cubo[1],
and Juan Manuel Murillo[2]

[1] Department of Computer Science, University of Málaga
Campus de Teatinos, 29071. Málaga, Spain
{jcamara,canal,cubo}@lcc.uma.es
[2] Dept. of Computer Science, University of Extremadura, Spain
Avda. de la Universidad s/n, 10071. Cáceres, Spain
juanmamu@unex.es

Abstract. The development of adaptive user interfaces has traditionally been restricted to research prototypes and few commercial products. Although there have been relevant achievements in the architectural support for self-adaptive context-aware systems [3,19], the notion of context commonly supported is restricted and does not explicitly contemplate the facets of context related to user-application interaction. Furthermore, applications need to comply with the proposed architectures, making the incorporation of adaptivity more difficult (or not possible at all) in the case of already existing applications. This work addresses key issues for the incorporation of self-adaptive behaviour in GUI-Based applications, and proposes an aspect-based framework in order to overcome current limitations.

1 Introduction

As computing applications become more complex and sophisticated, users tend to spend more time and effort trying to instruct and configure them, having to explicitly state an ever-increasing amount of information in order to efficiently carry out the tasks they are demanded. This happens because computing applications are not context-aware entities, and must be supplied with additional (context related) information that we, as human beings, implicitly assume in different situations. Contextual information has a dynamic nature and changes as we interact with our environment. The notion of a changing context dependent on user-application interaction has been subject to research by the Computer Science Community by many years [18], producing a broad range of interactive systems with the same philosophy in common: that it can be worth learning something from the user and adapt to it in some non-trivial way. This kind of system has been labelled in different ways, ranging from *personalized systems* to

[*] This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 106–114, 2007.

Intelligent User Interfaces (IUIs). We will refer to them as *User Adaptive Systems*, or more specifically as *Adaptive User Interfaces* (AUIs) [11]. These contrast with traditional user interfaces, rigid and passive, which allow only a small degree of user customization by setting preferences. Making interfaces flexible and adaptable to the user implies the extraction of a *user model* by retrieving information based on the interaction of the user with the interface. Then, based on predictions made using that information, the system modifies its behaviour and structure for a better interaction with the user.

Dey and Salber define context in [19] as: *Any information that can be used to characterize the situation of an entity. An entity is a person, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.* Specifically, in our proposal we deal with two types of context-related information:

– *Application context*: Information related to UI description, application domain, and task specifications.
– *User context*: Information related to user behaviour and preferences.

There is a third kind of Contextual information relevant to the application referred to as *environmental* or *sensed* [7]. This information is indirectly related to the behaviour of the user and includes location, time, etc. However, in this work we focus just on application and user context information, since the use of environmental context has been broadly studied in the field of context-aware applications [2]. In this work we advocate for the adoption of a new architectural approach to the development of adaptive interfaces based on *Aspect-Oriented Programming* (AOP) [17], which provides the foundations to enable adaptive behaviour in applications which have already been deployed. Hence, rather than dealing with user modeling or learning algorithms, this work is focused on how aspects can be used to apply such techniques on already existing systems, extending them with adaptive capabilities.

The rest of this paper is organised as follows: Section 2 points out the main issues for the incorporation of adaptive behaviour in UIs. Section 3 and 4 describe and detail the design and implementation of the framework. Section 5 describes some related work. Finally, section 6 discusses the benefits our proposal purports as well as some open issues.

2 Issues Reusing and Enabling Adaptivity

(i) Poor reusability. Building an AUI involves the inclusion of additional requirements to the system, such as user modeling techniques (which implies retrieving information derived from the interaction with the user) or dynamic reconfiguration of structure and behaviour. Developing these mechanisms is costly, and often requires the use of sophisticated techniques specifically tailored to each individual application. As a result, the application code which corresponds to the adaptive mechanisms is tightly coupled with the rest of the application, hampering reusability.

(ii) Lack of transparency. Even when the architectural support for the AUI provides a good modularization, AUIs are generally tailored for an specific application [10,20]. In such a way, the effective reuse of new behaviour is affected by its underlying representation.
(iii) User model fragmentation and redundancy. Different applications tend to have different user models. Moreover, each application retrieves and stores its own information locally, and with its own representation, impeding the reuse of this information by other applications. This results in a fragmented and heterogeneous user model which only reflects a partial view of the user's behaviour and preferences to different applications.
(iv) Lack of coordination. Conventional techniques do not consider the possibility of coordinating adaptation between different existing applications located within a shared context. In such a way, applications cannot communicate with each other in order to carry out tasks collaboratively.

3 Framework Architecture

This section describes our aspect-oriented framework, intended to enable adaptive capabilities on passive GUI-based applications. The architecture of this framework is structured using a modified version of the the *Adaptability Aspects* [4] architectural pattern in order to provide better maintainability and modularity. Adaptability Aspects are applied to a base application for the adaptation of its interface. As it can be observed in Figure 1, the framework incorporates the following functional elements:

- **Context Manager.** Identifies context changes and triggers adaptive actions implemented by the aspects. It constantly monitors user input through the `UserMonitor` aspect, and the UI through the `UIMonitor` aspect, identifying the structure, properties, and relations between its components.
- **Adaptation Data Provider**. Consists on a set of classes which manage the information related to the different models required for UI adaptation and its processing. Specifically, the **User Model** holds up information about user context, while **Task Model**, **Domain Model**, and **UI Model** comprise information about application context. These elements provide the input to the **Reasoner** module, where the specific adaptive logic is implemented (learning and inference). The reasoner produces an **Adaptation Model** as the result of the application of the adaptive mechanisms, which is used as a specification for the adaptation to be performed on the interface.

Adaptability Aspects adapt the interface using the adaptation model whenever they match user-generated or UI events. For that purpose, they use a set of **Auxiliary Classes** intended to improve reusability. These provide common mechanisms for the addition, modification, or removal of UI components.

Figure 2 depicts how the elements of the framework interact when a context change triggers an adaptation on the application behaviour:**(a)** The application starts execution.**(b)** The context manager begins to monitor the context continuously.**(c)** Whenever an event in the UI is detected, both adaptability aspects

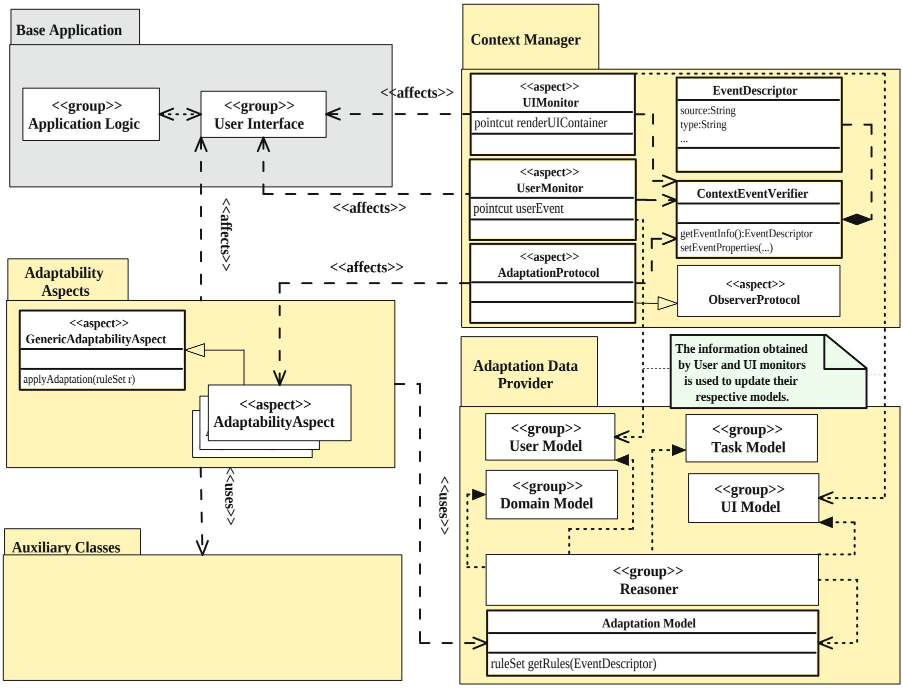


Fig. 1. Framework Architecture. Stereotypes `<<affects>>` and `<<uses>>` are used in some dependency relationships to represent classes whose behavior is monitored or changed by an aspect, or those used as auxiliary classes by an aspect, respectively

and adaptation data provider are notified.**(d)** The adaptation data provider updates the adaptation model.**(e)** Adaptability aspects access the adaptation model to verify if adaptation should be performed. **(f)**Aspects change the application behavior making use of the auxiliary classes. It is worth noticing that when adaptability aspects are notified about a context change, the adaptation to be performed is not carried out inmediately. On the contrary, adaptive behavior is introduced on the base application just after the adaptation model has been updated.

4 Detailing the Framework

Currently, there is a wide range of toolkits which can be used in order to build GUI-based computer applications, such as KDE/Qt, GNOME/GTK/GTK+, MFC, JFC, etc. All of them provide similar abstractions and basic mechanisms both for the programmer and the user. Although our approach is applicable to other toolkits and aspect languages, we use JFC (specifically Swing) and AspectJ[12] in the implementation of our prototype for their widespread use.

Within the **Context Manager**, the `UIMonitor` aspect obtains information about the structure, properties, and relations between UI components. In Swing

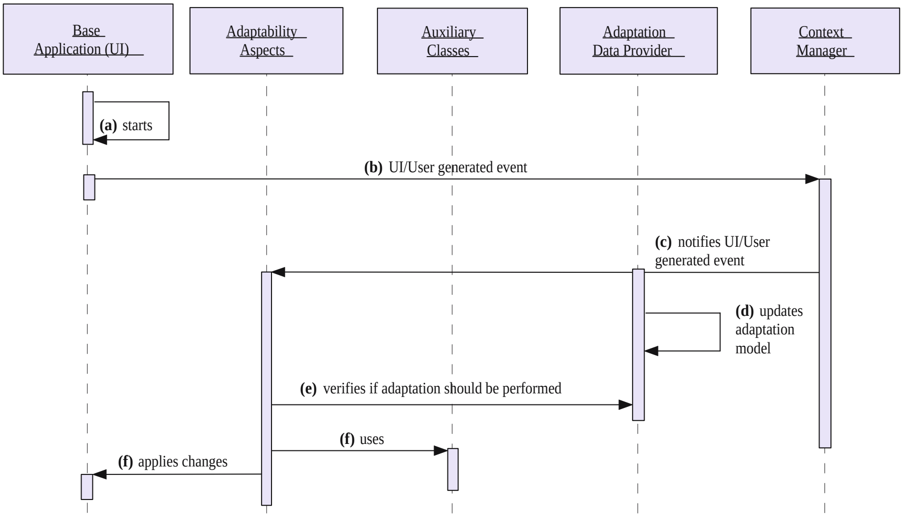

Base Application (UI)
Adaptability Aspects
Auxiliary Classes
Adaptation Data Provider
Context Manager
(a) starts
(b) UI/User generated event
(c) notifies UI/User generated event
(d) updates adaptation model
(e) verifies if adaptation should be performed
(f) uses
(f) applies changes

Fig. 2. Dynamics of the process followed for UI adaptation

and other toolkits, components can either be regular, or have the capability to group other components together. Such components act as *containers* and include *panels*, *windows*, *frames*, and *dialogs*. The structure of a container (*e.g.,* a window), along with its nested components in a hierarchical structure is obtained defining a `renderUIContainer` pointcut to match the join points where a container is rendered. It can be observed how the `Container` object is exposed to the body of the advice applied to the join point. The inspection code in the advice builds recursively a structure with all the child components of the container, which is incorporated into the UI Model along with component properties.

```
aspect UIMonitor {
java.util.List myComponents=new java.util.ArrayList<Component>();
pointcut renderUIContainer (Container c):
   call (* java.awt.Container.setVisible(*)) && target(c);
void getComponentList(Container container,
                      java.util.List<Component> components){
   for (Component component : container.getComponents()){
      if (component instanceof Container)
         getComponentList((Container) component, components);
      components.add(component); }
} ...
after (Container c): renderUIContainer(c){
      if (!UIModel.getComponent(c.getName())){
         UIModel.add(getComponentList (c, myComponents));
         myComponents.clear(); }
}
}
```

The extraction of the properties of each of the components in the structure is realised through reflection. It is worth mentioning that textual information is of special relevance to the purpose of our framework. This kind of information is present in almost any UI component. Note that *Menu Items*, *Buttons*, *Labels*, *CheckBoxes*, etc. have well defined properties (*e.g.,* `Text`, `ToolTipText`, etc.) which provide semantic information about the role of the component in the application.

The implementation of the **User Monitor** as an aspect provides a way of retrieving user implicit information unobtrusively. While interacting with the UI, every time the user types a character or clicks on an object, an event occurs. To detect user action on the interface, an object must implement the `ActionListener` interface. The program must register this object as an action listener on the button (*i.e.,* the event source), using the `addActionListener` method. When the user clicks the button, it fires an action event. This results in the invocation of the action listener's `actionPerformed` method. The argument to the method is an `ActionEvent` object that gives information about the event and its source. The `userEvent` pointcut is defined in the `UserMonitor` aspect to match any invocations of an `actionPerformed` method within the scope of the application:

```
pointcut userEvent(EventObject e, EventListener l):
   execution (void *.*(*)) && args(e) && target(l);
```

Both `EventListener` and the `EventObject` are exposed to the body of the advice applied, which incorporates relevant information about the event (source object, time, action performed, etc.) to the user model. Hartman and Bass [9] provide a detailed discussion about this approach to capturing interaction between user and applications.

Adaptability Aspects are notified by the context manager whenever a specific event occurs. This is achieved extending an implementation of the *Observer* pattern described in [8]. Specifically, whenever an event is matched by monitor aspects, the `EventDescriptor` in the `ContextEventVerifier` class is updated by the aspect. Then, the following `AdaptationProtocol` updates all the observing aspects, applying adaptation if the adaptation model determines that the produced event requires adaptation.

```
public aspect AdaptationProtocol extends ObserverProtocol{
   declare parents: ContextEventVerifier implements Subject;
   declare parents: AdaptivityAspect implements Observer;
   protected pointcut subjectChange(Subject s):
   call (call ContextEventVerifier.setEventInfo(..)) && target(s);
protected void updateObserver(Subject s, Observer o) {
   ContextEventVerifier cev = (ContextEventVerifier) s;
  ((AdaptivityAspect)o).applyAdaptation(
         am.getRules(cev.getEventDescriptor());
 }
}
```

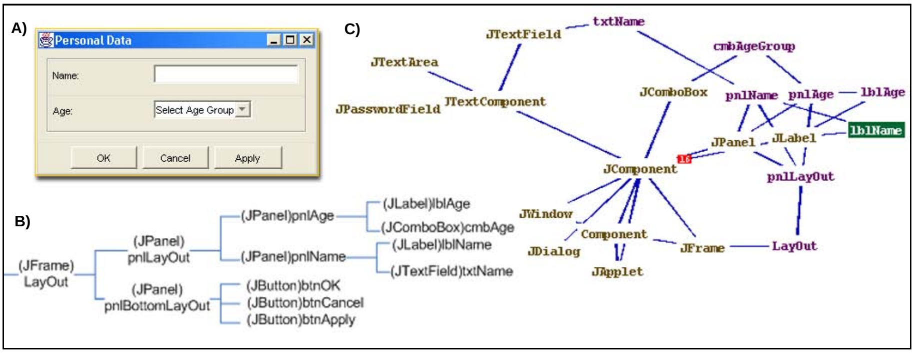


Fig. 3. A) Sample application dialog.**B)** Component hierarchy.**C)** Lattice representation of the UI Model ontology extended with the component hierarchy from the dialog.

The **Adaptation Data Provider** produces appropriate correspondences between user action and UI adaptation. This module enables the system to appropriately process the information which has been previously acquired. In order to represent UI components, the framework uses UI-specific ontologies, and domain-specific ontologies for representing the application's domain. Semantic information present in components is used to establish relations between UI and domain-specific ontologies. For the task model, we integrate a task ontology based on ConcurTaskTrees [16], which is a hierarchical notation which allows the specification of (sub)tasks or nodes that need to be performed to successfully complete a task.

The generation of an Adaptation Model involves significant decision-making capability. Hence, the Reasoner must support the writing of rules specifying the decisions that need to be made. Although these decisions may be sometimes relatively simple, truly adaptive behaviour implies that most of the time adaptation rules are likely to change over time. For this reason the prototype uses a rules engine, where a set of rules can be repeatedly applied to the collection of facts available in the different model ontologies. Rules that apply are executed, modifying the Adaptation Model accordingly. Specifically, the framework uses Jess [5], which is very convenient since it allows direct creation and manipulation of Java objects.

5 Related Work

Supporting GUI adaptation based on AOP is not a new idea. Sendín et al. take in [21] a non-intrusive approach to the problem of GUI plasticity, but depending exclusively on environmental context information. Hence, the UI is adapted depending on device characteristics, location, etc., but it is not responsive to user interaction.

Proposals such as the Context-Broker Architecture (CoBrA) or the Context Toolkit [3,19] relieve developers from building specific adaptive mechanisms, letting them focus on adaptive behaviour. However, these proposals do not explicitly deal with the particularities of user interfaces, and lack the transparency of the AOP approach since applications have to comply with the architecture's specifications.

To our knowledge, there is no proposal available focused on enabling adaptivity in already existing applications based on user-computer interaction context. Furthermore, ours is a non intrusive approach that enables the use of an additive plug-in structure to reuse general patterns of adaptive behaviour.

6 Conclusions

This work advocates for an aspect-based approach to enable adaptive behaviour on already existing, GUI-based applications. We have presented a framework which allows to overcome the different problems described in section 2: **(i)Poor reusability and (ii) Lack of transparency.** The framework's architecture enables reusability and transparency, providing an explicit and non-invasive way of altering and extending the UI. The use of the presented framework permits to apply general adaptation patterns to different facets of regular applications in a transparent way (*i.e.,*the application does not need to be specifically prepared for adaptation and will still benefit from adaptive behaviour not specifically designed for it).**(iii)User model fragmentation and redundancy.** The framework is able to collect user information unobtrusively and in a centralized manner. The use of a global user model accessible to all applications through a *generic user modeling server* [13] enables adaptive applications to cooperatively retrieve and use both implicit and explicit (*i.e.,* preferences) information from the user. This ensures data consistency, and a fast growth of the amount of information obtained from users. As a result, learning and inference based on that information is more accurate and efficient, since applications have better user models available in shorter periods of time. User information is precious to AUIs, to the point that learning algorithms are specifically tailored to work with very restricted sets of information [14]. **(iv)Lack of coordination.** The ontology model supported by the Adaptation Data Provider enables coordinated adaptation since cross-inference and learning can be performed on different application UIs. Domain and Task Models can be developed to comprise several applications, bridging tasks across different UIs.

Currently, our framework prototype is being extended in order to apply it to real-world examples of adaptive systems. We intend to validate our proposal implementing systems which have already been described in the literature [6,10] with our approach to test its applicability. In this sense, we have a special interest in applying it to *end-user/Programming by Demonstration* (PBD) development systems [15,1], a field in which our approach can realise its full potential.

References

1. Burnett, M.M., Cook, C.R., Rothermel, G.: End-user software engineering. Commun. ACM 47(9) (2004)
2. Chen, G., Kotz, D.: A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth College (2000)

3. Chen, H., Finin, T., Joshi, A.: An intelligent broker for context-aware systems. In: Proc. of UBICOMP'03 (2003)
4. Dantas, A., Borba, P.: Adaptability aspects: An architectural pattern for structuring adaptive applications with aspects. In: Proc. of SugarLoafPLoP'2003 (2003)
5. Friedman-Hill, E.: Jess in Action. Manning Publications (2003)
6. Gajos, K.Z., Czerwinski, M., Tan, D.S., Weld, D.S.: Exploring the design space for adaptive graphical user interfaces. In: Proc. of AVI'06 (2006)
7. Gray, P., Salber, D.: Modelling and using sensed context information in the design of interactive applications. In: Nigay, L., Little, M.R. (eds.) EHCI 2001. LNCS, vol. 2254, Springer, Heidelberg (2001)
8. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proc. of OOPSLA'02 (2002)
9. Hartman, G.S., Bass, L.: Logging events crossing architectural boundaries. In: Costabile, M.F., Paternó, F. (eds.) INTERACT 2005. LNCS, vol. 3585, Springer, Heidelberg (2005)
10. Hermens, L.A., Schlimmer, J.C.: A machine-learning apprentice for the completion of repetitive forms. IEEE Expert 9(1) (1994)
11. Jameson, A.: The Human-Computer Interaction Handbook. In: Adaptive Interfaces and Agents, Lawrence Erlbaum Associates (2003)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
13. Kobsa, A.: The Adaptive Web: Methods and Strategies of Web Personalization. In: Generic User Modeling Systems, Springer, Heidelberg (2007)
14. Langley, P.: Machine learning for adaptive user interfaces. In: Brewka, G., Habel, C., Nebel, B. (eds.) KI-97: Advances in Artificial Intelligence. LNCS, vol. 1303, Springer, Heidelberg (1997)
15. Mørch, A.I., Stevens, G., Won, M., Klann, M., Dittrich, Y., Wulf, V.: Component-based technologies for end-user development. Commun. ACM 47(9) (2004)
16. Paterno, F., Mancini, C., Meniconi, S.: Concurtasktrees: a diagrammatic notation for specifying task models. In: Proc. of INTERACT'97 (1997)
17. Filman, R.E., Friedman, D.: Aspect-Oriented Software Development. In: Aspect-Oriented Programming is Quantification and Obliviousness, Addison-Wesley, Reading (2004)
18. Ross, E.: Intelligent user interfaces: Survey and research directions. Technical Report CSTR-00-004, University of Bristol (2000)
19. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: Aiding the development of context-enabled applications. In: Proc. of CHI'99 (1999)
20. Segal, R., Kephart, J.O.: Mailcat: An intelligent assistant for organizing E-mail. In: Proc. of Agents'99 (1999)
21. Sendín, M., Lorés, J., Montero, F., López-Jaquero, V.: Towards a framework to develop plastic user interfaces. In: Chittaro, L. (ed.) Mobile HCI 2003. LNCS, vol. 2795, Springer, Heidelberg (2003)

Architecture Migration Driven by Code Categorization

Rui Correia[1,2], Carlos M.P. Matos[1,2], Reiko Heckel[1],
and Mohammad El-Ramly[3]

[1] Department of Computer Science, University of Leicester, University Road,
Leicester, LE1 7RH, United Kingdom
{rmc20,cmm22,reiko,mer14}@mcs.le.ac.uk
[2] ATX Software, Rua Saraiva de Carvalho 207C, 1350-300 Lisboa, Portugal
[3] Computer Science Department, Cairo University, Egypt

Abstract. In this paper, we report on the development of a methodology for the evolution of software towards new architectures. In our approach, we represent source code as graphs. This enables the use of graph transformation rules, allowing the automation of the transformation process. Prior to its model representation, the source code is subject to a preparatory step of semi-automatic code annotation according to the contribution of each of its parts in the target architecture. This paper first describes the overall methodology and then focuses on the code annotation and model transformation parts. We also discuss issues of the implementation of the approach based on existing tools.

1 Introduction

As business and technology evolve and software becomes more complex, researchers and vendors of tools in reengineering are constantly challenged to come up with new techniques to effectively support the transition of legacy systems to modern architectures.

In this paper, we introduce a methodology to fill the gap that exists in addressing systematically the complexity of architecture migrations. We argue that, starting from a legacy application, such transitions involve different steps of decomposition. Depending on the target architecture, these are made along one or both technological and functional dimensions. Technological decomposition is used in the layering of software systems and may, for example, lead to a 3-tiered architecture, separating logic, data, and user interface (UI). Functional decomposition, used to move to Service Oriented Architectures (SOAs), separates components which, when removing their UI tier, represent candidate services.

Based on a metamodel for both source and target architecture, the methodology presented in this paper consists in (1) categorizing the source code according to the different elements of the target architecture they shall be mapped to, (2) obtaining a metamodel-based representation of the code, (3) transforming it into the target architecture, and (4) generating the target code.

At this point, we have implemented in an automated way the transformation and partially implemented the semi-automatic code categorization, but only for

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 115–122, 2007.

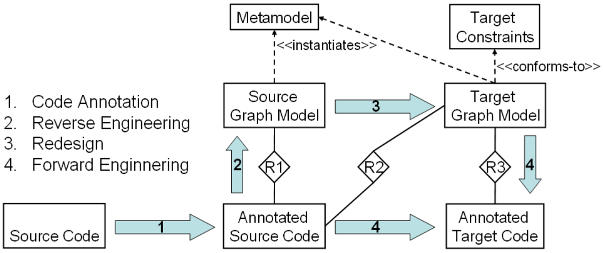


Fig. 1. Methodology for architectural migration

the technological decomposition (or layering). This is where the paper concentrates on. Its main contribution is the automation of the architectural migration using graph transformation rules over a model of the annotated source code. This allows us to: abstract (in large parts of the process) from the specific languages involved and describe transformations in a more intuitive way (compared to code level transformations). Along the paper a small Java client-server application is used to exemplify the implementation of some of the steps. The intended target architecture is 3-tier.

The remainder of this paper is organized as follows: Section 2 presents our methodology for architectural transformation. The code annotation and the model transformation parts of our approach are discussed in Sections 3 and 4 respectively. We review related work in Section 5 and discuss conclusions and further work in Section 6.

2 Architectural Transformation Methodology

In this section we discuss methodological aspects of our approach to architectural redesign. We are following the Horseshoe Model [1], refining it to support automation and traceability. Our methodology consists of the three steps of *reverse engineering, redesign*, and *forward engineering*, preceded by a preparatory step of *code annotation*, as illustrated in Figure 1.

A *metamodel* composed of a type graph that represents the technological paradigm of the system and a list of the code categories needed regarding the target architecture is used. *Target constraints* are also used to ensure that the target model is achieved and complies to the expected one.

1. Code Annotation. The source code is annotated by *code categories*, distinguishing its constituents (classes, methods, or fragments thereof) with respect to their foreseen association to architectural elements of the target system.

The annotation is done by means of comments in the original source code and even though the code is categorized statement by statement, in the end, consecutive statements of the same category are grouped making possible for a whole class or package being annotated with just one code category. This procedure makes the graph model representation of the source code much simpler than if there was no categorization step, since the level of detail used can be much lower. Simpler graph models make the redesign step easier to scale.

This semi-automatic code annotation is based on *categorization rules* defined at the level of the Abstract Syntax Tree (AST), taking into account information obtained through dependency analysis and inputs by the developer. The results may have to be revised and the propagation repeated in several iterations, leading to an interleaving of automatic and manual annotations.

2. Reverse Engineering. From the annotated source code, a *graph model* is created, whose level of detail depends on the annotation. For example, a method wholly annotated with the same code category is represented as a single node, but if the method is fragmented into several categories, each of these fragments has to have a separate representation in the model. The relation *R1* between the original (annotated) source code and the graph model is kept to support traceability. This step is a straightforward translation of the relevant part of the AST representation of the code into its graph-based representation.

3. Redesign. The source graph model is restructured in order to comply to the target architecture. In our approach, code categories provide the control required to automate the transformation process, focussing user input on the annotation phase. During this redesign step, the relation with the original source code is kept as *R2* in order to support the code generation.

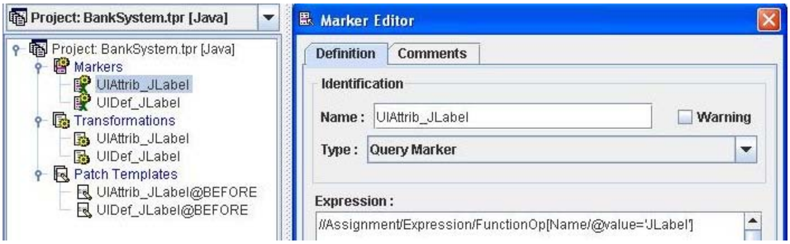
This *code category-driven transformation* is specified by graph transformation rules, conceptually extending those suggested by Mens *et al* [2] to formalize refactoring by graph transformation.

4. Forward Engineering. The target code is generated from the target graph model and the original source code, using their relation *R2* as an input. The result of this step, the annotated code in relation with a graph model, has the same structure as the input to Step 1. Hence, the process can be iterated. This is particularly relevant if the reengineering is directed towards service-oriented systems, because the transformation has to address first the technological and then the functional decomposition.

3 Code Annotation

The annotated source code is obtained through an iteration of manual definition of the categorization rules and the automatic application of them, based on the categories defined in the *metamodel*. After each iteration, manual input might be needed to refine the categorization rules in order to achieve code fully categorized.

In the example mentioned in section 1, our goal is to reach a three-tier architecture, thus we can use the following code categories: User Interface (UI), Logic, Data, Control: UI to Logic, Control: Logic to UI, Control: Logic to Data. For different target architectures, other categories might be defined as well as more complex ways to represent them.



Name : UIAttrib_JLabel

Type : Query Marker

Expression :

```
//Assignment/Expression/FunctionOp[Name/@value='JLabel']
```

Fig. 2. Categorization rule example

The rules used in the categorization process are applied over the AST. They can be programming language specific or depend on previously categorized code and taking into account information obtained through dependency analysis.

We use L-CARE [3], a tool developed in ATX Software, to design the rules. This tool uses XPath [4] to query the AST and allows us to automatically annotate the source code using comments. L-CARE has been used over the last years in multiple large projects, which ensures the scalability of this step. In Figure 2 we show an example of a rule for Java that categorizes all attributes of type *JLabel* as belonging to the *User Interface* concern, since we know that *JLabel* is of this concern.

The source code annotation allows us to abstract to graph model level only the relevant information. This is extremely important to reduce the size of the source graph model and consequently making the transformation process much more efficient.

4 Redesign

The source graph model of Figure 1 is an abstraction of the code achieved through the categorization process. It keeps traceability to the code in order to facilitate the transformation / generation process and it is an instantiation of the metamodel that holds information about the source and target models.

Graph transformation rules are then applied to the source graph model in a fully automated way, transforming it into the intended target architecture.

4.1 Type Graph

To take advantage of graph transformation rules in the transformation process, we developed a type graph which is part of the *metamodel* present in Figure 1.

The model that we are using has the goal of being flexible enough so it can be instantiated by any OO application regardless of the specific technology. This way there is a better chance that it can be reused for different instantiations of our methodology. The high level view of the model can be seen in Figure 3. The *CodeFragment* package is used to represent code elements and is an extension of the type graph presented by Mens *et al* in [5]. This extension was necessary in

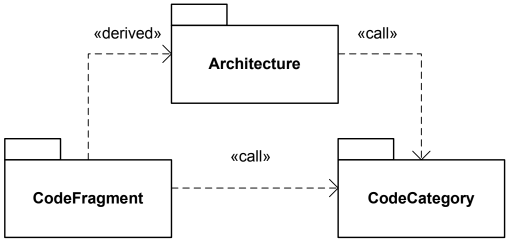


Fig. 3. High level view of the type graph

order to introduce classification attributes and the notion of code blocks, needed because the code categorization requires finer granularity than that of methods. Package *Architecture* includes the concepts of *Component* and *Connector* that allow us to represent the mapping between the programming language elements and the architecture level. The code categories information is represented in package CodeCategory.

Since it is necessary to keep traceability to the code in order to facilitate the transformation / generation process, a method to associate it to the type graph had to be considered. Given that we want to be as language-independent as possible we did not link the type graph directly to the source code but used instead an attribute (ASTNodeID) to associate its elements to the AST of the program.

4.2 Transformation Specification

Our use of graph transformation rules to describe model transformations is similar to what is being used in refactoring research [6]. However, refactoring rules are not enough for all reengineering purposes because sometimes it is necessary to perform transformations that are not completely behavior-preserving. An example of this is when we want to transform a legacy client-server system into a web-based application. The UI has to be changed because of the differences in the user communication paradigm between these different architectures. Another major difference is that our transformations are code category oriented, thus allowing architectural modifications.

An example of transformation rule specification is the *Move Method UI* rule. This rule searches for occurrences of methods classified as UI that are contained in classes that are classified as non UI (for example: having multiple concerns). The result is that those methods are moved to the appropriate UI classes. A small example of this rule application is presented in section 4.3. To specify the rules we are using the Tiger EMF Transformation tool [7], an Eclipse plugin.

4.3 Transformation Execution

For our instantiation of the reengineering methodology we are using code generated by the transformation specification in Tiger.

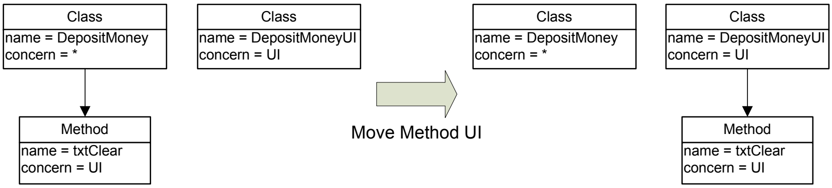


Fig. 4. Graph models before (left) and after (right) the application of rule Move Method UI. The graphs were simplified for readability. (The architectural elements are not shown and, in reality, the attribute ”concern” does not exist directly in the code elements but is used as an association to code category elements.)

The transformation execution applies the rules defined in the transformation specification to the source graph model to obtain the target one.

Part of an example of the source graph model can be seen in the left side of Figure 4. The value “*” for the attribute “concern” means that the element contains more than one concern. For example, class “DepositMoney”, even though it is not visible in the simplified figure, contains methods that belong to different concerns. This occurrence constitutes a potential candidate for the application of the transformation rule *Move Method UI* previously described. When we apply it, the method “txtClear” is moved from the class “DepositMoney” to “DepositMoneyUI”, a class belonging to the UI concern as shown in the right side of Figure 4. This transformation is an example of a rule that contributes to the layering of the application.

4.4 Constraints

The global constraints, in our approach, are imposed by the type graph. This way we ensure that source, intermediate and target models are compliant to the general requirements.

In order to assure that the target model complies to the desired architecture, we define target constraints over the metamodel that correctly reflect the architectural paradigm. For instance, in 3-tier applications, there should be no UI and Logic layer methods in the same class and no direct links from UI to Data These constraints can be defined as graph transformation rules.

5 Related Work

Program transformation can occur in different levels of abstraction. The source-to-source level of transformation is the most established one, both in research and in industrial implementations. There are several research ideas that led to successful industrial tools. Examples from research include TXL [8] and ASF+SDF [9]. DMS from Semantic Designs [10] and Forms2Net from ATX Software [11] are program transformation tools being successfully applied in the industry. Transformations at the detailed design level, due to its applications as maintenance techniques, have an increasing interest that is following the same path. Practices such as Refactoring [12] are driving the implementation of functionalities that automate detailed design level transformations. These are mainly integrated in development environments as is the case of Eclipse [13] and IntelliJ [14]. However, there is still a lot of ongoing research in this area, for instance, the work of Mens *et al* in the determination of dependencies between refactorings [5]. At the architectural level of program transformation there is some important research, e.g. the work in the Software Engineering Institute of CMU [1], but the industrial cases have been limited to specific source and target architectures and programming languages.

6 Conclusion and Future Work

Most of the ongoing research in the context of automated software transformation, as well as existing industrial tools, focus on textual and structural transformation techniques that intend to solve very specific problems within well defined domains (e.g. program restructuring, program renovation, language-platform migration). Our experience indicates that such techniques fall short of addressing in a systematic way the complexity of the architecture-based transformation problem. In practice, when such a problem arises, these approaches have to be combined in a trial and error fashion, the success of which often depends on the experience of the reengineering team and on the specific problem at hand. On the other hand, there exist techniques and tools that work well at an architectural level, but with the main goal of documenting and visualizing the architecture of applications rather than supporting increased levels of automation in architecture-based transformations. Although such tools can provide a very good starting point and facilitate the subsequent effort, in industry projects a reengineering approach that starts with redocumenting architectures is often too limited given the time and budget constraints.

In this work we have presented a systematic approach in order to explicitly address this issue. This paper focused on the code annotation and model transformation techniques to obtain the target architecture. The use of code category-driven graph transformation rules provides us several benefits, including: abstraction from programming language specifics (languages of the same paradigm share most of the same implementation), description of the transformations in a more intuitive way than that of code level transformations, and possibility of using existing tools for the tranformation execution and for analysis over the models (e. g. constraint checking).

Presently we are in the process of completing the tools in order to apply them to a large real-world scenario. This way, it will be possible to test a good set of categorization and transformation rules and see if more need to be developed.

Acknowledgments

R. Correia and C. Matos are Marie-Curie Fellows seconded to the University of Leicester as part of the Transfer of Knowledge, Industry Academia Partnership Leg2Net (MTK1-CT-2004-003169). This work has also been supported by the IST-FET IP SENSORIA (IST-2005-16004).

We would also like to thank L. Andrade and G. Koutsoukos (ATX Software) for their contribution in the development of the overall methodology. M. El-Ramly contributed to this work while lecturer at the University of Leicester.

References

1. Kazman, R., Woods, S., Carrière, J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: WCRE '98. Proceedings of the Fifth Working Conference on Reverse Engineering, pp. 154–163. IEEE Computer Society Press, Washington, DC, USA (1998)
2. Mens, T., Demeyer, S., Janssens, D.: Formalizing behaviour preserving program transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
3. ATX Software: L-CARE, http://www.atxsoftware.com/?sec=products\&it=818
4. W3C: XPath, http://www.w3.org/TR/xpath
5. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software and Systems Modeling (to appear, 2007)
6. Mens, T., Eetvelde, N.V., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. Journal of Software Maintenance and Evolution: Research and Practice 17(4), 247–276 (2005)
7. Tiger EMF Transformation Project: Tiger EMF Transformation, http://tfs.cs.tu-berlin.de/emftrans
8. Cordy, J., Dean, T., Malton, A., Schneider, K.: Source transformation in software engineering using the TXL transformation system. Journal of Information and Software Technology 44(13), 827–837 (2002)
9. van den Brand, M., Heering, J., Klint, P., Olivier, P.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
10. Baxter, I., Pidgeon, C., Mehlich, M.: DMS®: Program transformations for practical scalable software evolution. In: ICSE '04. Proceedings of the Twenty Sixth International Conference on Software Engineering, pp. 625–634. IEEE Computer Society, Washington, DC, USA (2004)
11. Andrade, L., Gouveia, J., Antunes, M., El-Ramly, M., Koutsoukos, G.: Forms2Net - Migrating Oracle Forms to Microsoft.NET. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, Springer, Heidelberg (2006)
12. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Boston, MA, USA (1999)
13. The Eclipse Foundation: Eclipse, http://www.eclipse.org/
14. Jetbrains: IntelliJ IDEA, http://www.jetbrains.com/idea/

Effective Tool Support for Architectural Knowledge Sharing

Rik Farenhorst, Patricia Lago, and Hans van Vliet

Department of Computer Science
VU University Amsterdam
The Netherlands
{rik,patricia,hans}@cs.vu.nl

Abstract. Knowledge management plays an important role in the software architecting process. Recently, this role has become more apparent by a paradigm shift that views a software architecture as the set of architectural design decisions it embodies. This shift has sparked the discussion in both research and practice on how to best facilitate sharing of so-called architectural knowledge, and how tools can best be employed. In order to design successful tool support for architectural knowledge sharing it is important to take into account what software architecting really entails. To this end, in this paper we define the main characteristics of architecting, based on observations in a large software development organization, and state-of-the-art literature in software architecture. Based on the defined characteristics, we determine how best practices known from knowledge management could be used to improve architectural knowledge sharing. This results in the definition of a set of desired properties of architectural knowledge sharing tools. To improve the status quo of architectural knowledge sharing tools, we present the design of an architectural knowledge sharing platform.

1 Introduction

Software architecting is a recognized discipline in software engineering, albeit one that is still emerging [1]. In the last decade, research and industry have primarily considered a software architecture as a high level design captured in sets of components and connectors [2] that can be represented using various viewpoints [3]. In recent years, there has been an increasing awareness that not only the architecture design itself is important to capture, but also the knowledge pertaining to it. Often, this so-called architectural knowledge is defined as the set of design decisions [4,5], including the rationale for these decisions [6], together with the resulting architectural design [7]. Establishing ways to manage and organize such architectural knowledge is one of the key challenges the field of software architecture faces [8].

As illustrated by a survey about the duties, skills, and knowledge of architects, software architecting is a highly knowledge-intensive process [9]. Many different stakeholders are involved in this process. Due to the increase in size and complexity of software systems, architecting means collaborating. Hence, it is often the case that there is no one single all-knowing architect; instead the architect role is fulfilled by more than one person [10]. In order to make well-founded decisions, all involved stakeholders need to obtain relevant architectural knowledge at the right place, at the right time.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 123–138, 2007.

Consequently, sharing such knowledge is crucial, in particular for reusing best practices, obtaining a more transparent decision making process, providing traceability between design artifacts, and recalling past decisions and their rationale.

The need for sharing architectural knowledge was also observed in a study we conducted at a large software development organization [11]. We found that architects rely heavily on communication to work adequately and to produce high-quality results, and that many formal and informal discussions take place in the coffee room, at the hallway, or during other social events or meetings. However, despite the need for sharing architectural knowledge, software architects in practice often stick to familiar technology such as office suites for their daily tasks. Since such tools are rather ineffective to manage and share architectural knowledge, most of this knowledge remains implicit.

In this paper we propose foundations for *effective* tool support for architectural knowledge sharing. To this end, we define a set of properties architectural knowledge sharing tools should have. These properties are derived by defining typical characteristics of the architecting process, and by determining what there is to learn from the knowledge management domain. For the former, we combine our observations of software architecture practice with a review of software architecture literature. For the latter, since architecting is a knowledge-intensive process, we look at best practices known from knowledge sharing literature.

Based on the set of desired properties, we asses the conformance of a number of architectural knowledge sharing tools to these properties. The results indicate that not all of these properties are adequately supported. Consequently, in an attempt to improve the status quo, we propose an architectural knowledge sharing platform that incorporates all desired properties.

The remainder of this paper is organized as follows. In Section 2 we describe our observations of architectural knowledge sharing in a large software development organization. In Section 3 we combine these observations with a review on state-of-the-art software architecture literature in order to define the main characteristics of software architecting. In Section 4 we elaborate on best practices known from knowledge sharing literature to indicate what there is to learn from the knowledge management domain. Based on these best practices, in Section 5 we define a set of desired properties of architectural knowledge sharing tools. In Section 6 we examine existing software architecture tools by investigating how well these tools adhere to the defined properties. In Section 7 the design of our architectural knowledge sharing platform is presented. Finally, Section 8 presents our conclusions and outlines our ongoing and future work.

2 From the Trenches

One can only understand what architects really need, by asking them. This principle motivated us to closely investigate the software architecture practice in our research on architectural knowledge sharing tool support. This investigation helps to explain why only few of the software architecture tools proposed by academics seem to really make it to software architecture industry.

In this section some observations of knowledge sharing in the software architecting process of PGR, a large Dutch software development organization, are elaborated. For about one year we have closely monitored the architecture department of PGRfrom a architectural knowledge sharing perspective. This monitoring included the architecting activities undertaken, the various roles architects have, the information needs and the tools they use for their daily tasks. The research methodology that we followed for the investigation of the architecting process of PGR can best be described as an instantiation of action research. Action research is an iterative research approach in which the researcher actively participates in the studies he performs. The researcher wants 'to try out a theory with practitioners in real situations, gain feedback from this experience, modify the theory as a result of this feedback, and try again' [12]. In our case, the theory involved which architectural knowledge sharing tool support is helpful for architects in their daily work. To come up with improvements, we have diagnosed the current situation, the results of which are described in the remainder of this section.

In PGR architecting plays an important role. The architectural description is used as basis for development, and architectural guidelines need to be adhered to during system development and maintenance. In most software development projects architects work together in teams. One of the reasons for working in a team is that very few architects are skilled in both business and technology related aspects, while in most projects both these aspects play an important role. Consequently, there is a need to communicate and share information – both among architects and other stakeholders.

Architects in PGR have acknowledged that it is often hard to share information in a structured way. As a result, information sometimes gets lost when work is transferred from one department to the other, leading to redundant work and a lower overall quality of the architecture. Sharing relevant information is further constrained by deadline pressures posed by most projects the architects work on. There is usually insufficient time to explore all possibilities or alternatives, or to validate the results gained from brainstorm sessions with colleagues or the customers.

Below we elaborate upon our observations with respect to the three prominent architectural knowledge sharing tools available in PGR: an expertise site, a knowledge repository, and a knowledge maps system.

- **Expertise site.** For the architecture department of PGR an architecture expertise website has been created. This site uses Microsoft Sharepoint as underlying technology and offers functionality such as discussion forums, and news updates. However, during our investigation we found that in its current form the expertise site does not appeal to architects. There is very little content and only a small amount of people are registered users. Furthermore, editing or adding information to the site is non-intuitive and relatively old information is difficult to retrieve. The expertise site is merely used as a document repository where presentations and white papers are stored. Architects indicated that they visit the expertise site only occasionally, and this is partly due to its limited content, but also because they lack a real community feeling in the architecting department in the first place.
- **Knowledge repository.** PGR has developed a knowledge repository that harbors a set of guidelines used to guide architects in creating an architecture. After the architect answers a number of predefined questions, the repository uses its guidelines to advise the architect about the architectural solution. Unfortunately, in practice this repository is hardly used by architects. During interviews the reasons for this

problem became apparent: the tool is highly prescriptive and the architects' perception is that this limits the freedom they have in devising a solution. Moreover, the guidelines stored in the repository are rather outdated and the list of questions is rather large, and therefore time consuming. For these reasons, instead of using the repository architects prefer to edit existing architectural descriptions, since that yields results of similar quality, while saving time. Many architects mentioned that they have tried out the repository once or twice, after which they concluded the added value of using it was limited.

- **Knowledge maps.** PGR has also started an organization-wide initiative that aims to connect knowledge and knowledge workers throughout the organization. This intranet system is an information place where various information sources within the organization are combined and presented to the user. Users are able to fill in so-called knowledge maps that contain their areas of interest or expertise. Knowledge maps should make it easy to find colleagues based on their expertise. Unfortunately, the knowledge maps system is not that successful. The user interface is non-intuitive and slow, and on certain areas of interest, such as software architecture, no knowledge maps can be defined. For these reasons, the knowledge maps system is rather useless for software architects.

An often heard complaint during interviews held with architects in PGR, was that there are too many systems that contain useful information. As a result, people often decide to just ask a colleague directly, or stop looking at all, since they don't know which system harbors what knowledge, and thus what is the right place to look. Architects mentioned that a more central location that acts as glue between existing tools and that could be used as starting point for various knowledge-related tasks, would be a much welcomed alternative for the current state of practice.

3 Characteristics of Architecting

Software architecting is a knowledge-intensive process in which many design decisions are taken. These decisions are taken by carefully considering the available solutions, after which the best alternative is chosen. Architects often apply proven solutions, such as tactics or patterns to guarantee a high-quality architectural design. To make the best decisions, architects need a lot of information. Therefore, available architectural knowledge has to be shared effectively among the stakeholders involved to ensure that all relevant information can be taken into account.

As described in Section 2, PGR is struggling with how to effectively use tools to share architectural knowledge. The architectural knowledge sharing tools in place do not match the needs of the architects. To accommodate architects and other stakeholders in their knowledge needs, *effective* architectural knowledge sharing tool support is needed. Effective here means that the tools align to what architects in practice do, and that the time and effort required for using the tools is limited.

In order to properly define properties of effective architectural knowledge sharing tools, we investigate what a typical architecting process entails from a knowledge management perspective. In this section we use software architecture literature to define five characteristics of the architecting process. By studying the architecting process, we are able to find out how architectural knowledge sharing tool support could be employed to support architects in their knowledge intensive tasks. We do not claim that our set of characteristics is complete, but we believe that by focusing on a broad set of literature we have covered the essential properties of architecting. Please note that not all these properties are unique to architecting; some of them could easily apply to other software engineering disciplines as well, such as (detailed design), testing, etc.

Architecting is consensus decision making. Architecting can be viewed as a decision making process that not only seeks the agreement of most stakeholders, but also resolves or mitigates the objections of the minority to achieve the most agreeable solution [13]. Often various stakeholders with different needs and concerns are involved in the architecting process. This is acknowledged by Bass et al. who present the Architecture Business Cycle to define architecting as a feedback loop between architects, the stakeholders and the architectural solution itself [2]. Although the architects take the final design decisions, they often do so in accordance with the important stakeholders. This decision making process lends itself for knowledge sharing initiatives that allow stakeholders to actively participate in this process. Architectural knowledge sharing tools should enable architects to efficiently work together in a team. Due to the size and complexity of most software systems, it is often infeasible for one architect to be responsible for everything alone. This focus on teamwork is especially true in global software engineering environments. Consequently, the ‘architect role’ is often fulfilled by multiple collaborating architects.

Architecting is iterative in nature. Due to the consensus-driven decision making characteristic, architectures are not designed overnight, but rather in an iterative way. Hofmeister et al. illustrate this iterative nature of architecting by the concept of a backlog that is implicitly or explicitly maintained by architects [14]. This backlog contains smaller needs, issues, problems they need to tackle and ideas they might want to use. Such a backlog drives the workflow, helping the architect determine what to do or decide next. Conceptually the architecture is finished when this backlog is empty. However, as long as the backlog has open issues it is worth relating these issues to the current state of the architecture design. Architects can then judge how these issues could best be addressed while maintaining the important qualities of the architectural design. Architectural knowledge sharing tools therefore should support traceability between knowledge entities, such as architectural decisions and identified problems.

Architecting is an art. Architects are responsible for reflecting the design decisions taken in comprehensive architectural models, by selecting the most suitable views and viewpoints or architecture description language. During these activities the creativity of the architect plays a crucial role [1]. This is particularly true when dealing with novel and unprecedented systems. In such cases, there may be no codified experience to draw upon. Knowledge sharing tools need to take this characteristic of architecting into account and should support the architect’s creativity instead of constraining it. This means that methods and tools probably work better if they are more descriptive in nature.

Architecting impacts the complete life-cycle. Many architects would agree on the statement that an architecture is never finished, but rather stays alive throughout the life of the software system. During maintenance and system evolution an architecture plays an important role in safeguarding architectural qualities. If relevant architectural knowledge is not stored correctly knowledge vaporization may be the result, turning an architecture into a black box [4]. It pays off to make available important architectural knowledge to various stakeholders such as developers and maintainers, instead of only targeting the architects.

Architecting is constrained by time. The previous characteristics of architecting show that architecting is a creative consensus-driven and iterative decision making process. In practice however, a heavy constraint on these characteristics is the available time that architects have. Often, 'time to market' forces architects to choose for suboptimal solutions. This phenomenon is pointed out in [15] where it is stated that ”in practice, architects find only one solution and not multiple alternatives to choose from. This is due to the hard constraints in industrial practice (e.g. time to market or budget) that forces architects to intuitively come up with a single solution based on their existing application-generic knowledge. In effect, this results in the architects not exploring the solution space and potentially missing alternative solutions.”

4 What to Learn from Knowledge Management?

The characteristics of architecting described in the previous section show that architecting is a creative, iterative decision making process often done in collaboration with colleagues and other stakeholders in the lifecycle. The fact that architects only have a limited amount of time to complete this decision making process further shows the need for effective architectural knowledge sharing tools. In this section we elaborate on best practices for knowledge sharing from the KM domain to find out how the architecture domain could learn from this established field. Please note that we restrict ourselves to knowledge sharing factors related to tool support. Various social [16], organizational and cultural [17], or personal factors [18] also heavily influence the success of knowledge sharing, but this is beyond the scope of this paper. More information about incentives for knowledge sharing can be found in [19], or in [11] where incentives for architectural knowledge sharing are identified.

Knowledge in software engineering is diverse and its proportion immense and steadily growing. Improved use of this knowledge is the basic motivation and driver for knowledge sharing in software engineering [20]. Only since the early nineties have the knowledge management and software engineering communities begun to grow together [21]. Since then, various knowledge sharing tools have been proposed in the software engineering domain, leading to concepts such as the Experience Factory [22], experience management systems [23], learning software organizations [24] and Software Engineering Decision Support [25]. In addition, considerable attention has been put to the concept of design rationale [26].

Literature presents warnings for the fact that not all knowledge sharing implementations are automatically successful. In [19] several factors that make knowledge sharing difficult are listed, such as the fact that knowledge sharing is time consuming, and that people might not trust the knowledge management system. Another warning is that striving for completeness is infeasible. In addition, we should be aware of the fact that a lot of the available knowledge cannot be made explicit at all, but instead remains tacit in the minds of people [18]. Sharing such tacit knowledge is very hard [27]. The potential limitations of knowledge sharing notwithstanding, we believe it is crucial to assist architects in practice with their daily work. Tools should assist architects in their knowledge-intensive tasks, by enabling them to discover, share, and manage architectural knowledge.

In order to design successful tools for knowledge sharing, a strategy needs to be chosen. Hansen et al. distinguish two main knowledge management strategies: codification and personalization [28]. Whereas codification is aimed at systematically storing knowledge so that it becomes available to people in the company, the personalization strategy focuses on storing information *about* knowledge sources, so that people know who knows what. In the architecting process, some architectural knowledge might benefit from a codification strategy, whereas other types of knowledge could be better shared using personalization approaches. A hybrid approach, first coined in [29], is therefore worth considering. Such a hybrid approach could provide a balance between formalized and unstructured knowledge. According to [30], such a balance is an important prerequisite to stimulate the usage of tools.

To define in more detail how a hybrid architectural knowledge sharing approach should look like we can draw on a study about knowledge sharing by Brink [31]. Brink describes that four steps need to be executed in order to create "an interconnected environment supporting communication, collaboration, and information sharing within and among office and non-office work activities; with office systems, groupware, and intranets providing the bonding glue". Firstly, information and explicit knowledge components must be stored online, indexed and mapped, so people can see what is available and can find it (e.g. using digitally stored documents or yellow pages). Secondly, communication among people needs to be supported, by assisting in the use of best practices to guide future behavior and enable sharing of ideas (e.g. emails, bulletin boards, or discussion databases). Thirdly, tacit knowledge needs to be captured using for instance communities of practice, interest groups, or competency centers (e.g. groupware and electronic whiteboards). Lastly, methods are required that offer a virtual space in which a team can collaborate interactively, irrespective of geographic distribution of the team members or time. To enable the four steps described above, Brink defines three categories that form technological enablers for knowledge sharing [31]: knowledge repository (for sharing explicit knowledge); knowledge routemap (for sharing explicit and tacit knowledge), and collaborative platforms (for sharing tacit knowledge).

The need for a combined approach that stimulates the collaboration of architects and that supports sharing both tacit and explicit knowledge, is also acknowledged in [32]. The authors describe seven knowledge work processes that range from finding codified information to establishing social networks and collaborating in communities. The author states that these knowledge processes can not be seen independently, but often are interrelated. For example, an architect searching for information on security might a) initiate a search on this specific topic, b) negotiate with colleagues about the meaning of what was just found, c) create new ideas based on the discussions and by using common sense, and d) try to maintain a social bond with these colleagues at the same time.

Based on the discussed knowledge management literature, we argue that architectural knowledge sharing tools can best follow a hybrid approach that combines codification and personalization methods, and that also stimulates collaboration between the stakeholders of the architecting process. More stable knowledge – such as best practices and architectural tactics – could be codified in a repository, less formalized knowledge could be spread in the organization more effectively using knowledge routemaps, and a collaborative platform allows architects and other stakeholders to work together on an iterative decision making process.

5 Desired Properties of Architectural Knowledge Sharing Tools

Based on the five characteristics of software architecting (Section 3) and best practices from knowledge management literature (Section 4) we define seven desired properties of architectural knowledge sharing tool support.

1. **Stakeholder-specific content.** Because *Architecting impacts the complete lifecycle* various stakeholders are involved in the decision making process, and all these stakeholders need specialized views on the available content, such as open issues, approved decisions, or scheduled meetings. Architectural knowledge sharing tools should make it possible to distinguish between certain types of knowledge. Users of the tool can then choose what architectural knowledge they want to retrieve. The search functions should therefore match with the profiles of different users, such as developers, maintainers, architects, or project managers.
2. **Easy manipulation of content.** Since *Architecting is iterative in nature*, architects follow a continuous iterative decision making process. Easy manipulation of content will keep the decision making process up to speed, whereas more rigid tool support that does not allow easy manipulation could instead slow it down.
3. **Descriptive in nature.** Because *Architecting is an art*, architects should not be constrained too much in their tasks. Architectural knowledge sharing tools should not prescribe how architects should manage architectural knowledge, for example by offering an abundance of predefined models, guidelines, or templates. Instead the tools should allow a descriptive perspective on the available architectural knowledge that does not limit the architects' creativity.
4. **Support for architectural knowledge codification.** Since *Architecting is constrained by time*, architects could be helped by quickly finding relevant architectural knowledge. For certain types of knowledge that is not subject to frequent changes, a codification strategy probably works best. Architects can then easily retrieve solutions that have proven themselves in the past, and reuse these solutions accordingly. This property also relates to the knowledge repository category mentioned by [31], and the statement from [30] that there should be a proper balance between formalized (i.e. codified) and less structured (i.e. personalized) knowledge.
5. **Support for architectural knowledge personalization.** Because *Architecting is consensus decision making*, architectural knowledge is not always immediately 'stable' enough to codify, because until consensus has been reached, decisions could change. For such knowledge, a personalization strategy could prove useful to enable architects to find who knows what, in a similar way as the knowledge routemaps proposed by [31]. Personalization techniques are also valuable to support the discussions and negotiations between stakeholders [32].

6. **Support for collaboration.** Because *Architecting is consensus decision-making* architectural knowledge tool support should explicitly support collaboration between different users. This property relates to the groupware criterion of [31] as well as to the collaboration requirement of [32]. This property enables architects to actively involve all important stakeholders in the decision making process. Since most architects are specialized in certain areas, tool support that supports collaboration also allows architects to use a ‘divide and conquer’ approach whenever possible.
7. **Sticky in nature.** This property could be seen as orthogonal to the others in the sense that it is an essential property to motivate people to start using an architectural knowledge sharing tool in the first place. With this we mean that people should be motivated to start using the tool, as elaborated upon by [33] in which several motivational factors are described. To prevent users from neglecting the tool after having played with it once, special features should be incorporated in the tool to let it obtain a certain level of stickiness [34]. Tools that are sticky motivate users to keep coming back to it, increasing the chance of widespread adoption in practice.

6 The Status Quo of Architectural Knowledge Sharing Tools

Based on the seven identified desired properties of architectural knowledge sharing tools, in this section we are assessing the status quo of tool support in the software architecture domain. We have selected a set of tools that represent the current state-of-the-art in architectural knowledge management tools, since all these tools have recently been introduced in software architecture literature and they all explicitly target architectural knowledge management as well.

The academic tools that are assessed include Archium [5], ADDSS [35], DGA DDR [36] and PAKME [37]. Archium is a tool environment proposed by Jansen et al. that is aimed at establishing and maintaining traceability between design decision models and the software architecture design [5]. Capilla et al. have proposed a web-based tool called ADDSS for recording and managing architectural design decisions [35]. Falessi et al. have devised a specific design decision rationale documentation technique, which is driven by the decision goals and design alternatives available [36]. Hence, it is called the Decision Goal and Alternatives (DGA) DDR technique. Ali Babar et al. have proposed a Process-based Architecture Knowledge Management Environment (PAKME) that allows storing generic architectural knowledge (such as general scenarios, patterns, and quality attributes), and project specific architecture knowledge (such as concrete scenarios, contextualized patterns, and quality factors) [37]. To form a balanced set of tools from academia and practice, we add to our assessment PGR’s three architectural knowledge sharing tools: the knowledge repository, expertise site and knowledge maps system. More details on these three tools are described in Section 2.

The results of the assessment are reflected in Table 1. For the assessment of the tools of PGR we were able to draw on our observations in this organization. For the assessment of the academic tools we used published literature about the tools as primary source of information. We have based the scores on our interpretation of the tools, but acknowledge that it is possible that the tools have evolved recently. Please note that

Table 1. Status Quo of Architectural Knowledge Sharing Tools

	SA			***KM***			
Desired property **Software arch. tool**	Stakeholder-specific content	Easy manipulation of content	Descriptive in nature	Support for AK codification	Support for AK personalization	Support for collaboration	Sticky in nature
Archium	-	+	+	+	-	-	?
ADDSS	-	-	+	+	-	-	?
DGA DDR	-	-	+	+	-	-	?
PAKME	-	+	+	+	-	+	?
PGR knowledge repository	-	-	-	+	-	-	-
PGR expertise site	-	+	+	+	-	-	-
PGR knowledge maps	-	-	+	-	+	-	-

we do not intend to give a strict judgment on the tools, but rather indicate whether they conform to the defined properties. If they do, this is reflected in Table 1 with a '+' score; if they do not, this has resulted in a '-' score.

Stakeholder-specific content is a property that we haven't found explicit support for in any of the studied tools. Archium is designed for a single user who could use the tool to establish and maintain traceability between design decision models and the software architecture design. The authors of ADDSS mention multi-perspective support as one of the envisioned features of ADDSS, but in the current prototype this is not yet implemented. DGA DDR and PAKME also do not mention stakeholder-specific content as a feature, but rather focus on the codification of the architectural knowledge in general. The same goes for PGR's knowledge repository, and the expertise site, although the latter allows users to find a lot of different types of information, but this information is not tailored to users. PGR's knowledge maps system is suffering from the same limitation since users can find experts on certain topics based on the knowledge maps, but the view they are able to obtain on this information is not customizable.

Easy manipulation of content is incorporated in three of the tools we studied. The language used in Archium offers explicit support for addition and modification of architectural design decisions. In PAKME, a maintenance component provides various features to modify, delete and instantiate different artifacts. This component also includes repository administration functions. PGR's expertise site in theory allows various stakeholders to change or update knowledge at a regular basis, although architects mentioned that such changes are not always "easy" to make. In PGR's repository and knowledge maps system modifying content is non-intuitive and time-consuming, resulting in a negative score for this property. With both DGA DDR and ADDSS we believe the underlying model used in these tools is rather formal and limited in scope. As a result, chances are that practitioners feel too constrained by having to comply to these models. An issue specific to ADDSS is that architecting is not assumed to be iterative.

This focus becomes apparent in ADDSS because the user can add the design decisions taken and then add a view to these decisions to gain traceability. However, changing the design decisions is only possible by starting a new iteration, and if this is done a new view also has to be included to prevent loss of traceability. Between iterations no connections are possible, so manipulating existing architectural knowledge is hard.

Descriptive in nature is something most tools in our investigation are, except PGR's repository that strongly prescribes the solution based on the questions that need to be filled in. This is related to the fact that most of the tools also score well on the *support for AK codification* property. Most tools follow a typical codification strategy. Users are able to add information to the tool that is then stored for future retrieval. The one exception to this approach is PGR's knowledge maps systems, which offers a mechanism for people in the organization to find each other. This *support for AK personalization* is unfortunately not well covered by all other tools. A related critique on the tools studied is that they also score low on explicit collaboration aspects, since they are built around the assumption that architects mainly codify information for reuse purposes. Although PGR's knowledge maps system allows experts to find each other, it does not offer structured means to let these experts collaborate. As a result most of the investigated tools get a '-' score on the *support for collaboration* property. The single exception here is PAKME. PAKME is built on top of an open source groupware platform to provide collaboration using content management, project management and the like.

To conclude our assessment, we have investigated whether the various tools are *sticky in nature*. To properly determine this for ADDSS, DGA DDR, Archium and PAKME, hands on experience is required, hence the question marks in Table 1. For the other three tools, we can assess the stickiness based on interviews with architects from PGR. The results of these interviews indicate that none of the three architectural knowledge sharing tools in PGR is sticky, since users are not motivated to keep using the tools: the knowledge repository is outdated and offers little added value; the expertise site is not flexible in adding, manipulating or retrieving architectural knowledge; the knowledge maps are very rigid and do not offer specific topics on software architecture, rendering this system useless for the architects.

In summary, we conclude that most of the architectural knowledge sharing tools we studied are descriptive in nature and focus primarily on codification. To improve the status quo, more emphasis should be put on stakeholder-specific content, support for personalization, explicit support for collaboration, and general characteristics that make tools appealing and sticky in the long run.

7 Effective Tool Support for Architectural Knowledge Sharing

We are currently working on an architectural knowledge sharing platform. In this section, the basic architecture and main vision of this platform are elaborated upon. Our platform is designed to incorporate all desired properties introduced in Section 5.

The platform is designed using a blackboard architecture. In this architecture all relevant architectural knowledge is stored centrally, and functionality is defined to operate on this knowledge. Stakeholders have various entry points that allow operations on the architectural knowledge. The central vision is to create an environment that allows

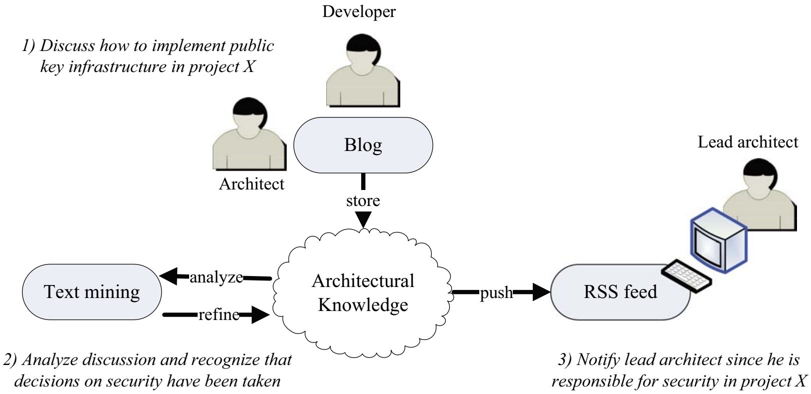

Developer
1) Discuss how to implement public key infrastructure in project X
Architect
Blog
store
Lead architect
Text mining
analyze
refine
Architectural Knowledge
push
RSS feed
2) Analyze discussion and recognize that decisions on security have been taken
3) Notify lead architect since he is responsible for security in project X

Fig. 1. A Scenario for the Architectural Knowledge Sharing Platform

architectural knowledge sharing in an effective manner, meaning that the platform assists architects in their daily work, without requiring them to spend much time or effort on it. This could be achieved by allowing intelligent management of the architectural knowledge stored in the platform, for example by structuring unstructured information.

To illustrate part of the functionality of our architectural knowledge sharing platform, consider the following scenario that is also depicted in Figure 1. A developer and architect who both work on project X have a discussion about how to best implement a public key infrastructure and they discuss various alternative solutions. Since they reside at different locations, they use a blog for their discussion. The blog, being part of the architectural knowledge sharing platform, integrally stores the discussion. This unstructured information is not very meaningful to stakeholders who are not directly involved in the discussion, but a structured summary is valuable to the lead architect of the project, since in our example he is ultimately responsible for the architecture design in Project X, including security. To this end, a text mining service, also part of the platform, opportunistically analyzes the blog discussion and determines that a decision on security has been made. Consequently, a summary of this architectural knowledge is sent to the lead architect using a RSS feed. Although the lead architect is unaware of the blog discussion that has taken place, he is able to determine whether the decision made by the architect and developer is the correct one. If this is not the case, he could intervene in time, preventing possible security problems later on in the project.

In the above example, the interplay between the blog, text mining and RSS feeds is apparent and it indicates how to enrich unstructured information and subsequently share it. We believe many similar kind of scenarios could be valuable to accommodate architectural knowledge sharing. The following central features are incorporated in the platform in order to allow codification and personalization of architectural knowledge, and to enable collaboration between stakeholders:

Blogs and wikis. Blogs and wikis are employed to allow designers and architects to easily communicate and collaborate. As a result, other stakeholders can quickly acquire information about the current status of the project, such as the design decisions that have been made, alternatives that have been considered, etc. One important motivation for people to blog is the ability to create a community feeling [38]. To foster communication between architects, and to motivate them to share architectural knowledge, such a community feeling is essential. Wikis offer support for collaboration. Using the Wiki, architects can work on the same project by concurrently editing (parts of) the architectural descriptions or meeting minutes.

Text mining. Blogs and Wikis are typical personalization approaches in which a lot of unstructured information is stored. However, potentially relevant architectural knowledge is much more valuable if it is codified as a reusable asset. Text mining techniques, see e.g. [39], are employed to enrich unstructured architectural knowledge present in the platform.

Best practices database. To store best practices or other reusable assets, the platform contains a best practices database. Architectural knowledge is codified in predefined formats, and could be retrieved for various purposes, such as reusing past design decisions, or to find out what guidelines exist on a certain topic. Architects can search this database via a standard web interface.

RSS feeds. RSS feeds are employed to push relevant architectural knowledge to certain stakeholders, or to notify subscribed stakeholders if new architectural knowledge emerges. RSS feeds are complementary to the more traditional pull mechanism of using the best practices database. Users can subscribe to certain topics of interest and get updated without having to search the platform themselves, which is a lightweight approach to share architectural knowledge among relevant stakeholders.

Expert finding. Similar to the ‘yellow pages’ concept, the expert finding facility allows users to easily find colleagues based on experience, interests or projects on which they work. By connecting people in the architecting process, we increase ‘team building’ within the organization, and foster discussions that can result in higher quality solutions. This approach also conforms to the *sticky in nature* property, because people like to share thoughts more easily if they have a ’group feeling’, and creating such an environment ensures a certain degree of stickiness.

Our platform conforms to all seven desired properties of architectural knowledge sharing tools. First of all, it has *support for architectural knowledge codification* (best practices database, text mining), and *support for architectural knowledge personalization* (blogs and expert finding), making it a hybrid architectural knowledge sharing environment. In addition, the platform explicitly *focuses on collaboration* between architects (Wiki). Moreover, the various services allow *easy manipulation of architectural knowledge* that is stored. In addition to supporting codification, personalization, and collaboration, we envision additional features to make the platform more appealing, for example by supporting flexible and personalized access for all stakeholders through a personal start page. Such a start page allows for *stakeholder-specific content* so that users can indicate which information, projects, or people they are interested in. In this sense, the platform becomes the preferred starting point for architectural knowledge sharing in the organization, without posing too much restrictions on its use. We envision the platform to act as glue between various knowledge sources in the organization. As described in Section 2 architects of PGR would greatly appreciate such an integrated platform. If we ensure that users perceive a certain degree of freedom in using the platform, the platform is inherently *descriptive in nature* as well. Lastly, to ensure that the platform is *sticky in nature*, it incorporates motivating factors, such as authority ranking and equality matching [33] In addition, the platform harbors facilities to support persistent and visible reputation tracking. People that help others using the architectural knowledge platform in any way, for example by sharing information, or by publishing content, are rewarded by gaining a certain reputation. According to Bush [34], support for reputation tracking appeals to users and motivates them to keep using the platform, hence increasing its stickiness.

8 Conclusions and Future Work

Software architecting is a knowledge intensive process. Consequently, tool support for architectural knowledge sharing has various benefits, such as reusing best practices, teaching staff, and support efficient collaboration between stakeholders. However, our observations in software architecture practice show that practitioners often stick to traditional tools, such as office suites, for their daily work, thereby missing the opportunity to effectively share architectural knowledge.

In this paper we have defined seven properties that architectural knowledge sharing tools should have to be effective. These properties have been defined by drawing on experience and literature in both the software architecture and knowledge management domain. By viewing software architecture from a knowledge management perspective we were able to determine which best practices from this field apply to the architecting process. Although we believe the identified properties are essential from an architectural knowledge management perspective, it should be noted that some of these properties might to a certain extent apply to other Software Engineering disciplines as well.

Based on the properties, we have assessed a number of existing software architecture tools. The results of this assessment indicate that the status quo of architectural knowledge sharing tool support lacks full conformance to the seven desired properties.

To improve the status quo, we have presented the design of an architectural knowledge sharing platform. This platform offers a hybrid architectural knowledge management approach and supports collaboration between stakeholders in the architecting process. To this end, it incorporates lightweight features using state-of-the-art techniques, such as Wikis, blogs, RSS feeds and text mining. We have shown that our platform conforms to all identified properties, thereby increasing the chance for successful widespread adoption in software architecture practice.

Our ongoing and future work focuses on the realization of our platform. We will populate the platform with several features and evaluate their success in industrial settings. In addition, we plan to extend our assessment of existing architectural knowledge sharing tools using hands on experience. Results of these assessments serve as valuable input in our effort to arrive at effective tool support for architectural knowledge sharing.

Acknowledgements

This research has been partially sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering

Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge.

References

1. Eeles, P.: The Process of Software Architecting. Technical Report (2006), available online: `http://www-128.ibm.com/developerworks/rational/library/apr06/eeles/index.html`
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. SEI Series in Software Engineering. Addison-Wesley Pearson Education, Boston (2003)
3. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Reading (2002)
4. 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)
5. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: WICSA. 5th Working IEEE/IFIP Conference on Software Architecture, Pittsburgh, USA, pp. 109–120. IEEE Computer Society Press, Los Alamitos (2005)
6. van der Ven, J.S., Jansen, A., Nijhuis, J., Bosch, J.: Design decisions: The Bridge between Rationale and Architecture. In: Dutoit, A. (ed.) Rationale Management in Software Engineering, pp. 329–346. Springer, Heidelberg (2006)
7. Kruchten, P., Lago, P., van Vliet, H.: Building up and Reasoning about Architectural Knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 39–47. Springer, Heidelberg (2006)
8. Shaw, M., Clements, P.: The Golden Age of Software Architecture. IEEE Software 23(2), 31–39 (2006)
9. Clements, P., Kazman, R., Klein, M., Devesh, D., Reddy, S., Verma, P.: The Duties, Skills, and Knowledge of Software Architects. In: WICSA. 6th Working IEEE/IFIP Conference on Software Architecture (2007)
10. Eeles, P.: Characteristics of a Software Architect. Technical Report (2006), available online: `http://www-128.ibm.com/developerworks/rational/library/mar06/eeles/index.html`
11. Farenhorst, R., Lago, P., van Vliet, H.: Prerequisites for Successful Architectural Knowledge Sharing. In: ASWEC. 18th Australian Software Engineering Conference, Melbourne, Australia, pp. 27–36 (2007)
12. Avison, D., Lau, F., Myers, M., Nielsen, P.A.: Action Research. Communications of the ACM 42(1), 94–97 (1999)
13. Eeles, P.: The Benefits of Software Architecting. Technical Report (2006), available online: `http://www-128.ibm.com/developerworks/rational/library/may06/eeles/index.html`
14. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: Generalizing a Model of Software Architecture Design from Five Industrial Approaches. In: WICSA. 5th Working IEEE/IFIP Conference on Software Architecture, Pittsburgh, USA, pp. 77–86 (2005)
15. Lago, P., Avgeriou, P.: 1st Workshop on SHaring and Reusing ARchitectural Knowledge (Final Workshop Report). ACM SIGSOFT Software Engineering Notes 31(5), 32–36 (2006)
16. Kankanhalli, A., Tan, B.C.Y., Wei, K.K.: Contributing Knowledge to Electronic Knowledge Repositories: An Empirical Investigation. MIS Quarterly 29(1), 113–143 (2005)
17. Cummings, J.: Knowledge Sharing: A Review of the Literature. Technical report, The World Bank Operations Evaluation Department (2003)

18. Nonaka, I., Takeuchi, H.: The Knowledge-Creating Company. Oxford University Press, Oxford (1995)
19. Ghosh, T.: Creating Incentives for Knowledge Sharing. Technical report, MIT Open Courseware, Sloan school of management, Cambridge, Massachusetts, USA (2004)
20. Rus, I., Lindvall, M.: Knowledge Management in Software Engineering. IEEE Software 19(3), 26–38 (2002)
21. Aurum, A., Jeffery, R., Wohlin, C., Handzic, M.: Managing Software Engineering Knowledge. Springer, Heidelberg (2003)
22. Basili, V.R., Caldiera, G., Rombach, D.H.: The Experience Factory. In: Encyclopedia of Software Engineering, vol. 2, pp. 469–476. John Wiley & Sons Inc., Chichester (1994)
23. Seaman, C.B., Mendonçą, M.G., Basili, V.R., Kim, Y.M.: User Interface Evaluation and Empirically-Based Evolution of a Prototype Experience Management Tool. IEEE Transactions on Software Engineering 29(9), 838–850 (2003)
24. Althoff, K.D., Bomarius, F., Tautz, C.: Knowledge Management for Building Learning Software Organizations. Information Systems Frontiers 2(3/4), 349–367 (2000)
25. Ruhe, G.: Software Engineering Decision Support - A new Paradigm for Learning Software Organizations. In: LSO. 4th Workshop on Learning Software Organizations, Chicago, USA, pp. 104–113 (2002)
26. Dutoit, A.H., McCall, R., Mistrik, I., Paech, B.: Rationale Management in Software Engineering. Springer, Heidelberg (2006)
27. Haldin-Herrgard, T.: Difficulties in Diffusion of Tacit Knowledge in Organizations. Journal of Intellectual Capital 1(4), 357–365 (2000)
28. Hansen, M.T., Nohria, N., Tierney, T.: What's Your Strategy for Managing Knowledge? Harvard Business Review 77(2), 106–116 (1999)
29. Desouza, K.C., Awazu, Y., Baloh, P.: Managing Knowledge in Global Software Development Efforts: Issues and Practices. IEEE Software 23(5), 30–37 (2006)
30. Hall, H.: Input-Friendliness: Motivating Knowledge Sharing Across Intranets. Journal of Information Science 27(3), 139–146 (2001)
31. van den Brink, P.: Social, Organization, and Technological Conditions that Enable Knowledge Sharing. PhD thesis, Technische Universiteit Delft (2003)
32. Röll, M.: Distributed KM - Improving Knowledge Workers' Productivity and Organisational Knowledge Sharing with Weblog-based Personal Publishing. In: Blogtalk 2.0. European Conference on Weblogs, Vienna (2004)
33. Boer, N.I., van Baalen, P.J., Kumar, K.: The Importance of Sociality for Understanding Knowledge Sharing Processes in Organizational Contexts. Technical Report ERS-2002-05-LIS, Erasmus Research Institute of Management (ERIM), Rotterdam (2002)
34. Bush, A.A., Tiwana, A.: Designing Sticky Knowledge Networks. Communications of the ACM 48(5), 66–71 (2005)
35. Capilla, R., Nava, F., Pérez, S., Dueñnas, J.C.: A Web-based Tool for Managing Architectural Design Decisions. In: SHARK. 1st ACM Workshop on SHaring ARchitectural Knowledge, Torino, Italy, ACM Press, New York (2006)
36. Falessi, D., Becker, M., Cantone, G.: Design Decision Rationale: Experiences and Steps Ahead Towards Systematic Use. In: SHARK. 1st ACM Workshop on SHaring ARchitectural Knowledge, Torino, Italy, ACM Press, New York (2006)
37. Ali Babar, M., Gorton, I., Jeffery, R.: Toward a Framework for Capturing and Using Architecture Design Knowledge. Technical Report UNSW-CSE-TR-0513, The University of New South Wales (2005)
38. Nardi, B.A., Schiano, D.J., Gumbrecht, M., Swartz, L.: Why We Blog. Communications of the ACM 47(12), 41–46 (2004)
39. Fan, W., Wallace, L., Rich, S., Zhang, Z.: Tapping the Power of Text Mining. Communications of the ACM 49(9), 77–82 (2006)

A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures

Gemma Grau and Xavier Franch

Universitat Politècnica de Catalunya (UPC)
c/ Jordi Girona 1-3, Barcelona E-08034, Spain
{ggrau,franch}@lsi.upc.edu

Abstract. There is a recognized gap between requirements and architectures. There is also evidence that architecture evaluation, when done at the early phases of the development lifecycle, is an effective way to ensure the quality attributes of the final system. As quality attributes may be satisfied at a different extent by different alternative architectural solutions, an exploration and evaluation of alternatives is often needed. In order to address this issue at the requirements level, we propose to model architectures using the *i** framework, a goal-oriented modelling language that allows to represent the functional and non-functional requirements of an architecture using actors and dependencies instead of components and connectors. Once the architectures are modelled, we propose guidelines for the generation of alternative architectures based upon existing architectural patterns, and for the definition of structural metrics for the evaluation of the resulting alternative models. The applicability of the approach is shown with the Home Service Robot case study.

1 Introduction

There is a gap between requirements and architectures which is mainly due to the fact they use different terms and concepts to capture the model elements relevant to each one [20]. However, there is a connection between architectural design decisions and the quality attributes of the final system and, so, it is possible to analyse and to evaluate an architecture in the context of the goals and requirements that are levied to the systems that will be build from it [8]. On the other hand, quality attributes may be satisfied by different alternative architectural solutions (i.e., architectural patterns) and, so, there is often the need of exploring and evaluating several alternatives.

In this paper we propose to address the generation and evaluation of architectures at the requirements level by using a goal-oriented approach. Goal-oriented models allow expressing the intentional concepts using the same constructs for the requirements and for the architectures. We consider that they are an adequate formalism for representing software architectures because they allow expressing usual architecture-related concepts such as component, node, file, resource, dependency and so on. Additionally, goal-oriented models are becoming intensively used in fields such as requirements engineering and organizational process modelling, which has two main implications. In the one hand, transition from organizational and system

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 139–155, 2007.

models to architecture models can be smoother due to the use of the same formalism, and even traceability benefits from this. On the other hand, contributions and findings made in these other fields can be assessed and eventually incorporated into software architecture modelling and analysis.

Because of that, goal-oriented models [25], [28] have already been used for representing software architectures addressing the gap between requirements and architectures [25]. Among the different existing goal-oriented proposals, we remark the *i** framework [27], a goal-oriented modelling language that allows representing the functional and non-functional requirements of an architecture using actors and dependencies instead of components and connectors. For more details on the adequacy of using *i** models for representing and analyzing software architectures, we refer to [17].

In this paper we use the *i** framework in order to support the generation and evaluation of alternative architectures. We are interested in doing this process in a reliable way and, thus, we propose a set of guidelines for performing both activities in a systematic manner. Therefore, the generation of alternatives is based on the use of architectural patterns [6] and the evaluation of alternatives is done by applying structural metrics [14] over the produced models. In order to show the applicability of our approach, we apply the proposed steps and guidelines to the Home Service Robot case study as presented in [22], [23].

The remainder of this paper is organized as follows. The problem statement for the Home Service Robot case study is presented in section 2. In section 3 we present a brief introduction to the *i** framework. In our approach we do not consider the generation and evaluation of alternative architectures as an isolated process and, so, the context of applicability and tool support are presented in section 4. In section 5 we propose a set of guidelines for the generation of alternative architectures based on existing architectural patterns. Our proposal of metrics for evaluating the resulting models and a set of guidelines for defining the metrics are presented in section 6. Finally, section 7 presents the conclusions and future work.

2 The Home Service Robot Case Study

This case study is based on the problem statement of the prototype of a Home Service Robot (HSR) for daily services provided in [22], [23]. The case study has been chosen because the details about the problem statement are very clear, it is simple enough to be analysed and understood in the context of a paper, and also because mobile robots are commonly used in software architecture examples [26].

According to [22], [23], the HSR is a prototype that supports the following daily home services:

- **Call and Come (CC).** This service analyzes the audio data sampled in order to detect predefined sound patterns. If a “come” command is recognized, the robot tries to detect the direction of the sound source, rotates to the direction of the sound source and tries to recognize a human. If the caller’s face is detected, the robot moves forward until it reaches within 1 meter from the caller. If a “Stop” command is recognized while the robot is moving, the robot stops.

- **User Following (UF).** The robot locates a user and constantly checks vision data and sensor data for keep following the user. The robot follows the user within 1 meter range. If the robot misses the user, it notifies him by saying "I lost you" and the action terminates.
- **Security Monitoring (SM).** The robot patrols around a house for surveillance using a map. Intrusion or accidents are defined as patterns recognizable from vision and sound data. If such an event is detected, the robot notifies the user directly via an alarm or indirectly through a home server.
- **Tele-presence (TP).** A remote user can control the robot using a PDA. The robot sends the remote user a map of the house and the user can command the robot to move to a specific position. In addition, the robot can send captured images to the remote PDA for surveillance.

In order to provide those services, the HSR has the following hardware components: a Single Board Computer that controls the peripherals; a Front Camera to allow face recognition, user tracking, security monitoring and tele-presence; a Ceiling Camera to do map building and self-positioning; 8 SL Microphones to interpret speaker commands and locate its specific position; a Structured Light Sensor to detect obstacles and recognize footsteps; an Actuator to allow the HSR movement; an LCD display to show information; a Wireless Lan to communicate to the Home Server; and, finally, a Speaker to generate sound.

For more details about the HSR problem statement we refer to [22], [23].

3 The *i** Framework

The *i** framework proposes the use of two types of models for modelling systems, each one corresponding to a different level of abstraction: the Strategic Dependency (SD) model represents the intentionality of the process and the Strategic Rationale (SR) model represents the rationale behind it. SD models focus on the relationships among different actors that cooperate for satisfying some goals and, so, they are more interesting from the architecture point of view. Consequently, in this paper we focus on SD models.

A SD model consists of a set of nodes that represent actors and a set of dependencies that represent the relationships among them. Dependencies express that an actor (*depender*) depends on some other (*dependee*) in order to obtain some objective (*dependum*). Thus, the *depender* depends on the *dependee* to bring about a certain state in the world (goal dependency), to attain a goal in a particular way (task dependency), for the availability of a physical or informational entity (resource dependency) or to meet some non-functional requirement (softgoal dependency). These four types of *dependum* allow two different types of relationships: *intentional*, representing what behaviour a component expects from other parts of the system; and *operational*, representing how one component communicates with other parts of the system. Therefore, intentional relationships are represented by:

- **Goal dependencies** stating functional requirements, e.g. the User depends on the HSR for the goals *Come when called* and *Accidents are avoided*.

- **Softgoal dependencies** stating high-level non-functional requirements, e.g. the User depends on the HSR depends for the softgoals *User position location is accurate* and *Robot movement is efficient*.
- **Resource dependencies** stating flow of concepts, and remarkable some type of knowledge, or a concept, relevant for the domain that does not physically exist, e.g. the concept *Voice Command* in the context of the HSR.

On the other hand, *i** may be also used to represent architectural concerns by means of the following operational relationships:

- **Task dependencies** stating service invocation, e.g. the HSR depends on the User for the tasks *Introduce position in map* and *Introduce Sound patterns*.
- **Resource dependencies** stating information interchange, e.g. the HSR depends on the user for the resource *Tele-surveillance images*.
- **Goal dependencies** stating fit criteria for non-functional requirements, e.g. the HSR *Ensures security by avoiding obstacles*. Note thus that non-functional requirements change from being represented as soft goals to goals.

For more details about *i**, we refer to [21], [27]. The graphical notation is shown in Fig. 1. using, as an example, the SD dependencies between the HSR and the User.

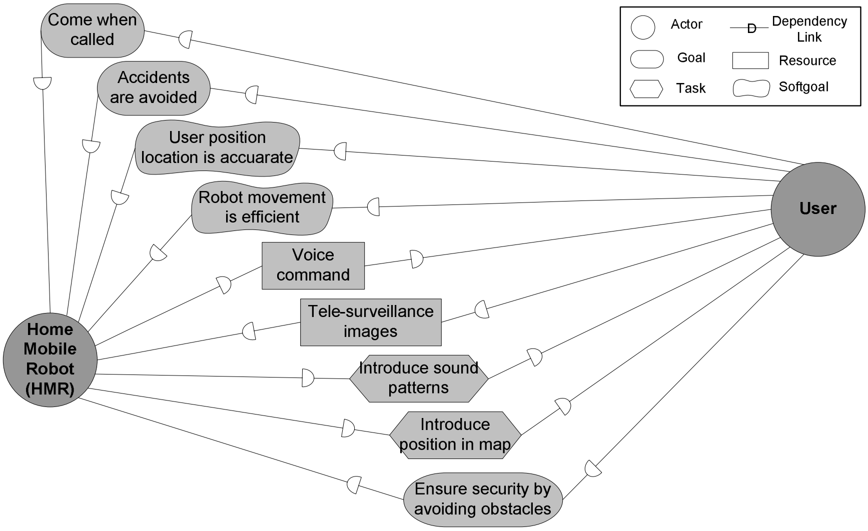


Fig. 1. Excerpt of an *i** model for the HSR case study

4 Context of Applicability and Tool Support

In our approach we do not consider the generation and evaluation of alternative architectures as an isolated process but as a part of a reengineering framework, that we have named ReeF [16]. The reasons for such a claim are twofold. On the one hand, most of the reengineering approaches consider the generation and evaluation of alternative solutions. On the other hand, in most of the cases, it is possible to state the premise that there is always a current process undertaken by humans or by a legacy system that can be used as a departing point for reengineering. Several proposals in the field of software architectures also follow a reengineering approach, among them we remark [3], [22], [23].

We have refined the ReeF generic framework into SAR*i*M [16], a Software Architecture Reengineering *i** Method, which is composed of the following phases: 1) Analysis of the source software architecture; 2) Conceptualization of the analysed software architecture into an *i** model; 3) Elicitation of new requirements for the software architecture; 4) Exploration of candidate software architecture solutions; 5) Assessment of the generated solutions using evaluation techniques; and 6) Creation of the specification for the new software architecture. In this paper we focus on phases 4 and 5. However, we assume that other existing techniques have been used for the first three phases and, so, before the generation and evaluation of architectures, there is an existing source *i** architecture model and a set of quality attributes to be used as the starting point. These artefacts can be generated with many existing techniques, among which, we have chosen the PR*i*M method [19] for generating the source *i** architecture model, and KAOS [9] for obtaining the requirements and quality attributes.

In Fig. 2 we provide an overview of the proposed process. We can observe that the generation of alternatives begins with the definition of the generic *i** architectural patterns. These patterns are constructed using already existing architectural solutions, which are selected according to the quality attributes to be achieved. Once the generic *i** architectural patterns are defined, the alternative architectures are generated by applying a dependency analysis and matching process over the source *i** architecture model and each selected generic *i** architectural pattern, resulting to set of alternative *i** architecture models. We remark that the generic *i** architectural patterns can be stored and reused when reapplying the process. Finally, once the alternative *i** architecture model are generated, they can be evaluated by defining or reusing structural metrics, providing the evaluation results that would assess the selection of the most suitable architecture.

In order to perform the generation and evaluation of alternatives in a reliable way, tool support is needed. We propose to use J-PR*i*M [18], a tool that supports the first

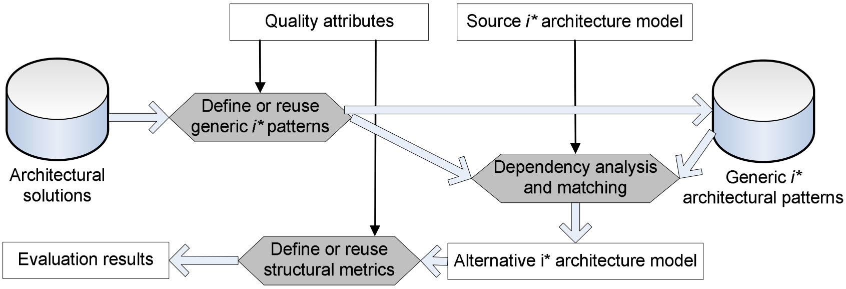

Quality attributes
Source *i** architecture model
Define or reuse generic *i** patterns
Architectural solutions
Dependency analysis and matching
Generic *i** architectural patterns
Evaluation results
Define or reuse structural metrics
Alternative i* architecture model

Fig. 2. Overview of the process for the generation and evaluation of alternative architectures

five phases of ReeF using the techniques proposed in PR*i*M [19] and, thus, it has been adapted to the generation and evaluation of alternatives that we propose.

5 Generation of the Alternative Architectures

There are several existing proposals on the generation of architectures, for instance [3], [4], [20], among others. These methods have in common that they all use an existing architecture specification as a departing point; they all propose the use of well-known architectural solutions for the generation of alternatives; and, they all choose the solutions according to previously obtained quality attributes.

On the other hand, existing work on the generation of alternative architectures using the *i** framework as a modelling language adopt a similar approach. This work is mainly represented by [24], [2], and it is oriented towards agent-oriented software architectures. In their context, patterns are used for generating organizational architectures which are represented in *i**. Based on these patterns, the *i** models are build by matching the concepts represented in the *i** organizational patterns with the functional and non-functional requirements for the new software architecture.

Based on this existing work, our approach for the generation of the alternative software architectures proposes four guidelines that transform existing architectural solutions into generic *i** architectural patterns and, then, use a matching process between the dependencies expressed in the source *i** architecture model and the ones defined in the generic *i** architectural patterns. These guidelines can be intertwined and iterated as needed. We remark that guidelines 1.1 and 1.2 are only applied once for each architectural pattern, allowing reusability when reapplying the method.

- **Preliminaries: Pattern Selection.** There are many architectural patterns that can be used for generating software architectures. So, in order to generate alternative architectures in a controlled way, we will only explore those patterns that help the achievement of the quality attributes we want for the final software system. The way these quality attributes are elicited from the stakeholders remains out of the scope of this paper, however, we mention that most of the quality attributes will be represented as softgoals in the *i** model. A way to select the patterns is to use the NFR [7] approach for modelling the contributions of each softgoal to the architectural patterns and, then, select those with a positive contribution. It is also possible to check the properties of each pattern in a pattern catalogue [6], a feature-solution graph [4], or a Property to Style Mapping Table [20], among others.

For instance, if the quality attribute to achieve is Maintainability we may select a Blackboard pattern, and if Exchangeability is needed, we may select a layered architecture. None of these architectural solutions have to be selected if efficiency is a crucial point for the architecture.

- **Guideline 1.1: Actor Identification.** Once the pattern is selected, we analyse it in order to identify the architectural components suggested by the pattern. Each component will be modelled as an actor in the *i** model of the new alternative architecture.

For instance, in the *Blackboard architectural pattern* as defined in [6], three actors are identified: the Blackboard, the Knowledge Source and the Control. We remark that, according to the pattern documentation, several Knowledge Sources can be used. Moreover, in some cases, the specific number and name of the components remains undefined in the pattern. For instance, that's the case of the *Layers architectural pattern* as defined in [6]. In this situation, in order to discover all the actors we have first to determine the number of layers and the abstraction level that they represent. This can be done by applying our own criteria or by adapting the criteria used to define other layer architectures, such as the OSI 7-Layer Model or the TCP/IP protocol [6].

- **Guideline 1.2: Definition of the generic *i** architectural pattern.** Once the actors are defined, the architectural solution is deeper analysed in order to abstract the general responsibilities of each actor and the generic dependencies with the other actors. As a result we obtain a generic *i** architectural pattern. The information needed for such an analysis is the one documented in the architectural solution. In order to enforce the link between requirements and architectures when deciding the kind of a certain dependency, we propose to adapt the six CBSP architectural dimensions proposed in [20] into the *i** framework:
  - *Task dependencies* model those elements that describe or involve processing components.
  - *Resource dependencies* model those elements that describe or involve data components.
  - *Goal dependencies* model those elements that that describe system-wide features or features pertinent to a large subset of the system's components or connectors.
  - *Softgoal dependencies* model those elements that describe or imply data or processing component properties, bus properties or system properties.

  In order to allow further reuse of the documented generic *i** architectural pattern the source of the pattern and the decisions taken during its definition have to be documented.

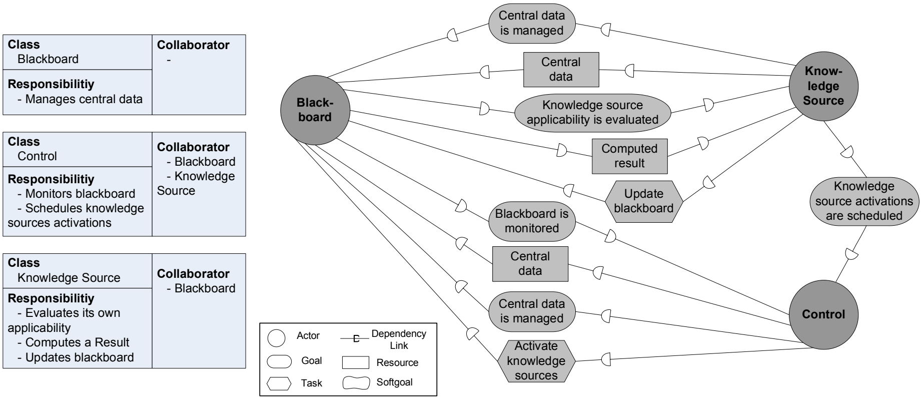
In Fig. 3. we show how we have defined the generic *i** architectural pattern for the *Blackboard architectural pattern*. At the left of the figure we can see the classes and their responsibilities as they are documented in [6]. We can also observe that we have added an *i** actor for each of the classes of the pattern. The dependencies have been established as follows:

- The Blackboard manages central data, which is a system feature and so, it is modelled as the goal dependency *Central data is managed*. As central data is a data component, a resource dependency *Central data* is also stated. As both the Knowledge Source and the Control depend on the Blackboard for the central data management, each dependency appears twice.
- The Control monitors the Blackboard and schedules the Knowledge Sources activations. Both are system features and so they are represented as goal dependencies. Thus, the Blackboard depends on the Control for *Blackboard is monitored* whilst the Knowledge Source depends on the Control for the goal *Knowledge source activations are scheduled*. We remark that the Control

monitors the Blackboard by analysing the Central data (which is an already existing dependency). On the other hand, as the Control involves a process for scheduling the Knowledge Sources, we need a task dependency stating that the Control depends on the Blackboard for *Activate knowledge sources*.

- The Knowledge Source evaluates its own applicability by using the central data. Thus, the Blackboard depends on the Knowledge Source for the goal *Knowledge source applicability is evaluated*. The Knowledge Source has the responsibility to compute a result (which involves a data component) and to update the Blackboard (which involves a processing component). Thus, the Blackboard depends on the Knowledge Source for the resource *Computed result*, and the Knowledge Source depends on the Blackboard for the task *Update blackboard*.

- **Guideline 1.3: Actors analysis and matching.** Using the source *i** architecture model and the generic *i** architectural pattern of the solution to be applied, we analyse the dependencies in both models in order to match the related elements and establish the equivalence between the source *i** architecture model actors and the generic *i** architectural pattern actors. As it is proposed in [17], in both groups we distinguish four kinds of actors:
  - Human actors. i.e., the final users of the software system.
  - Organizational actors. i.e. the organizations that provide or require services from the software system and its final users.
  - Software actors. i.e., the software system that is in charge to satisfy the human actor requirements. The software system can be represented by a unique software actor or by a set of actors that represents components and interact one with each other.
  - Hardware actors. i.e., the hardware devices in those software systems where we need to obtain certain information from the environment.



Class Blackboard	Collaborator -
Responsibilitiy - Manages central data	

Class Control	Collaborator - Blackboard - Knowledge Source
Responsibilitiy - Monitors blackboard - Schedules knowledge sources activations	

Class Knowledge Source	Collaborator - Blackboard
Responsibilitiy - Evaluates its own applicability - Computes a Result - Updates blackboard	

Fig. 3. Abstraction of the generic *i** pattern for the Blackboard architectural pattern

We remark that there are some actors on the source *i** architecture model that may not have an equivalence in the generic *i** architectural pattern and viceversa, for instance the actors that represent humans, organizational or hardware components in the source *i** architecture model are not typically actors of the generic *i** architectural patterns. This aspect is solved in the next guideline with the reallocation of responsibilities.

If we match the concepts of the Home Service Robot (HSR) and the *Blackboard architectural pattern* we can observe that the HSR involves a human actor (the User), a software actor (the Single Board Computer that is the component that controls the HSR), and several hardware actors that interact with the user (i.e., Front Camera, Microphones, Actuator, etc., see section 2 for more details). However, we can also observe that, although the actors on the generic *i** architectural pattern and the ones on the source *i** architecture model conform two disjoint groups, the HSR software actor that controls the HSR can be refined into the set of actors proposed by Blackboard *i** architectural pattern.

The *Blackboard architectural pattern* contemplates the possibility of having several Knowledge Sources. The strategy we follow to decide the number of Knowledge Sources and their specific responsibilities is to analyse other existing blackboard configurations specific for robots. Among them we have chosen the one proposed in [26], which suggest the following Knowledge Sources, that we model as actors in the alternative *i** architecture model: the Lookout, which monitors the environment for landmarks; the Pilot, which is in charge that planning the current path and control the robot actuators; and, finally, the Map Navigator, which plans the high-level path. The actor Control of the Blackboard *i** architectural pattern corresponds to the Captain component in [26] and the hardware actors can be considered as the perception subsystem in [26].

- **Guideline 1.4: Reallocation of responsibilities.** Once the actors of the source *i** architecture model and the generic *i** architectural pattern have been analysed, we create the new alternative *i** architecture model with the following actors:
  - The software actors of the generic *i** architectural pattern.
  - The human, organizational, and hardware actors of the source *i** architecture model.

  As the actors of the source *i** architecture model may not be considered on the generic *i** architectural pattern, the dependencies related with these actors have to be reallocated on the actors suggested by the pattern. This reallocation is done by matching the different elements in both models until having the entire source *i** architecture model dependencies represented following the structure of the generic *i** architectural pattern.

As a result of the matching activities we create a new alternative *i** architecture model with the human and hardware actors of the Home Mobile Robot source *i** architecture model and the software actors of the Blackboard *i** architectural pattern as it has been customized applying the previous guideline. Once this is done, the reallocation of responsibilities is carried out as follows: the processes for locating the user, analysing the distance with objects and detecting predefined intrusions patterns fall into the

Lookout actor; the current path planning, including the control of the movement actuators (rotation and advance functions) fall into the Pilot actor; the analysis of the current position and the planning of the tele-surveillance path, remain inside the Map Navigator; and, finally, the interpretation of user commands and the monitoring of all his/her actions is done by the Control actor. Dependencies steaming from or going to the hardware actors remains unchanged on the hardware actors' side and are reallocated into the Blackboard actor in the Software side, the rest are reallocated in the software actors as mentioned.

6 Evaluation of the Alternative Architectures

There are many proposals that address the evaluation of alternatives, and there is also already existing work to compare the different evaluation techniques [10]. According to [8], there are several categories of evaluation techniques:

- *Questioning techniques* allow investigating any area of the project at any state of readiness and include scenario-based methods [3], [4];
- *Measuring techniques* require the existence of some artefact to measure and include the definition of metrics for an static analysis of the structure, being common to use an Architecture Description Language for that purpose; and,
- *Hybrid techniques* that combine elements from questioning and measuring techniques, such as the ATAM method [8].

In a deeper analysis of the techniques used in each category, we can observe that most of the methods that evaluate architectures at their early stages use scenario-based techniques, and that Architecture Description Languages represent a much lower level of detail and focus on the evaluation of the behaviour and performance.

There is also work that addresses the evaluation of alternatives modelled within the *i** framework. Despite that most of this proposals use reasoning-based techniques [27], structural metrics are also being used [5], [12], [13], [14].

Based on the structure of the *i** SD models, it is possible to analyse the degree of fulfilment of the quality attributes for each alternative architecture, which allows evaluating the generated alternatives and informing their selection. The quality attributes can be evaluated with metrics in the form proposed in [14]. Metrics are defined in terms of the actors (actor-based metrics) and the dependencies (dependency-based metrics) of the model. It is also possible to distinguish between global and local metrics, where global metrics give an overall value of the quality-attribute under consideration and local metrics uses maximum and minimum values to locate specific elements. As we want to evaluate generated architectures, we will only work with global metrics, for the definition and use of local metrics see [14].

- **Global actor-based metrics.** Given an architectural property P and an *i** SD model that represents a system model M = (A, D), where A are the actors and D the dependencies among them, an actor-based architectural metric for P over M is of the form:

$$P(M) = \frac{\Sigma a\colon a \in A\colon \text{filter}_M(a) \times \text{correctionFactor}_M(a)}{\| A \|}$$

being filter_M: A→[0,1] a function that assigns a weight to the every actor (e.g., if the actor is human, software or from a specific kind), and $\text{correctionFactor}_M$: A→[0,1] a function that corrects the weight of an actor considering the dependencies stemming from or going to it.

- **Global dependency-based metrics:** Given an architectural property P and an *i** SD model that represents a system model M = (A, D), where A are the actors and D the dependencies among them, a dependency-based architectural metric for P over M is of the form:

$$P(M) = \frac{\Sigma d\colon d(\mathrm{a,b,x}) \in \mathrm{D}\colon \mathrm{filter}_M(d) \times \mathrm{correctionFactor}_{M,dee}(a) \times \mathrm{correctionFactor}_{M,der}(b)}{\| D \|}$$

  being filter_M: D→[0,1] a function that assigns a weight to the every *dependum* (e.g., if the *dependum* is goal, resource, task, softgoal if it is from a specific kind), and $\text{correctionFactor}_{\text{M,der}}$: A→[0,1] and $\text{correctionFactor}_{\text{M,dee}}$: A→[0,1] two functions that correct the weight accordingly to the kind of actor that the *depender* and the *dependee* are, respectively.

In order to guide the definition of the filters and correction factors proposed by the metrics and perform the evaluation of the generated architectures, we propose the following guidelines.

- **Preliminaries: Quality Attributes Selection.** Quality attributes tend to be non-functional requirements or constraints that have already arisen in the previous phases of the method and, as such, they are modelled as softgoals in the source *i** architecture model. However, not all the quality-attributes are equally important and, thus, we have to choose the most relevant to the new architecture. This can be done using different techniques being one of them prioritising the requirements (e.g., by considering individual stakeholder ranking of properties).

- **Guideline 2.1: Defining the Evaluation Goal.** The Goal Question Metric (GQM) paradigm [1] is commonly used for defining metrics. For instance, in [11] the GQM is used to analyse what has to be measured. In our case, the scope of measurement is restricted, as we already know that we want to measure the degree on what the software architecture ensures a quality attribute. Thus, the general form of the evaluation goal will be:
  - **To** evaluate **the** <quality attribute> of the modelled software architecture **in order to** assess **it.**

  For instance, the evaluation goal for assessing the quality attribute maintainability is:
  - **To** evaluate **the** maintainability of the modelled software architecture **in order to** assess **it.**

- **Guideline 2.2: Defining the Goal Questions.** Once the goal is defined, questions for evaluating the goal have to be defined, in the same way as it is proposed in [1] and applied in [11].

For instance, for assessing the goal defined for maintainability, the question is:

- What elements do affect maintainability?

In the literature, there is evidence that maintainability is better achieved it in those architectures that present a low level of coupling and a high level of cohesion [6].

- **Guideline 2.3: Defining the Goal Questions Metrics.** Metrics are used to assess the questions and, as we have explained at the beginning of this section, they can be actor-based or dependency-based according to [14]. For deciding the kind of metric we propose to define the following questions:

  - What are the architectural elements that are more relevant for the quality attribute?

  If the components are more relevant, we define an actor-based metrics. If the connections are more relevant we will define a dependency-based metric. Once the kind of metrics is defined, we have to choose the values to be assigned to $\text{filter}_M(a)$ and $\text{correctionFactor}_M(a)$ in actor-based metrics, and the ones for $\text{filter}_M(d)$, $\text{correctionFactor}_{M,der}(a)$ and $\text{correctionFactor}_{M,dee}(a)$ in dependency-based metrics.

For guiding the selection of the most suitable structural element, we propose the set of questions shown in Table 1. The contents of the table was defined after a deep analysis of the structural elements on the *i** framework. This kind of analysis is similar to the one performed when applying metrics over UML Class Diagrams [15].

As a result, in Table 1 we present the set of questions for actor-based metrics. We observe that a certain actor can be filtered according to its kind and the specific

Table 1. Questions, answers and examples for stating the filters and correction factors of actor-based metrics

Metric element	Question	Answer	Example Value
1.1. Actor-based: $\text{filter}_M(a)$			
	Does the kind of the actor or the actor itself affects the quality attribute?		
		No	$\text{Filter}_M(a) = 1$
		Yes, the kind of component affects the quality attribute.	$\text{Filter}_M(a) = \begin{cases} w, \text{ if } a \in \text{Human} \\ x, \text{ if } a \in \text{Software} \\ y, \text{ if } a \in \text{Hardware} \\ z, \text{ otherwise} \end{cases}$
		Yes, the specific component affects the quality attribute	$\text{Filter}_M(a) = \begin{cases} m, \text{ if } a = \text{ActorA} \\ n, \text{ if } a = \text{ActorB} \\ \ldots \end{cases}$
1.2. Actor-based: $\text{correctionFactor}_M(a)$			
	Does the actor dependencies or the actors related with the dependecies affects the quality attribute?		
		No	$\text{CorrectionFactor}_M(a) = 1$
		Yes, the number of dependencies affects it.	$\text{CorrectionFactor}_M(a) = \dfrac{1}{\#\text{Dep}(a)}$
		Yes, the number of dependencies ER affects it.	$\text{CorrectionFactor}_M(a) = \dfrac{1}{\#\text{Dep}_{er}(a)}$
		Yes, the number of dependencies EE affects it.	$\text{CorrectionFactor}_M(a) = \dfrac{1}{\#\text{Dep}_{ee}(a)}$
		Yes, the number of actors related with a affects it.	$\text{CorrectionFactor}_M(a) = \dfrac{1}{\#\text{Actor}(a)}$

component it represents. A correction factor can be applied if the number of dependencies related with the actor (#dep(a)) negatively affects the quality attribute; if only the dependencies where the actor is a depender (#Dep$_{\mathrm{er}}$(a)) negatively affects the quality attribute; if only the dependencies where the actor is a depender (#Dep$_{\mathrm{ee}}$(a)) negatively affects the quality attribute; or, if it is the total amount of actors related with the actor (#actor(a)) that negatively affects the quality attribute.

We remark that both the filters and the correction factors can be further refined as needed until getting the desired level of detail. For instance, we may only be interested in the number of actors related with the actor that are of a certain kind or that represent and specific component. Also other arithmetical combinations are possible, if they allow providing more accuracy in the results. Dependency-based metrics would be defined following a similar approach.

As we have mentioned before, maintainability is better achieved in those architectures that present a low level of coupling and a high level of cohesion. In the structure of the *i** models a low level of coupling can be measured by stating an actor-based metric, where the number of actors related with the current actor negatively affects the property:

Actor-based coupling metric: $\mathrm{filter}_{M}(\mathrm{a}) = 1$ **and** $\mathrm{correctionFactor}_{M}(\mathrm{a}) = \dfrac{1}{\#\mathrm{Actor(a)}}$

In a similar manner, cohesion is related with the number of dependencies that steam from or goes through each actor. If the same dependency appears more than once, cohesion is damaged. In this case we can define a dependency-based metric as follows:

Dependency-based cohesion metric: $\mathrm{filter}_{M}(\mathrm{d}) = \dfrac{1}{\#\mathrm{Duplicated(d)}}$ **and** $\mathrm{correctionFactor}_{M,\mathrm{der}}(\mathrm{d}) = 1$, $\mathrm{correctionFactor}_{M,dee}(\mathrm{d}) = 1$

- **Guideline 2.4: Evaluating the Metrics.** The evaluation of the metrics is done by applying the corresponding actor-based or dependency-based formula with the values stated in the previous guideline. As alternative *i** architecture models can be large and complex, tool support is essential. As we have mentioned in section 4, we use J-PR*i*M [18] to support the evaluation of the alternatives according to the defined metrics.

In order to show the application of the metrics, we have generated and evaluated 4 different alternatives architectures for the HSR in J-PR*i*M [18]. In Fig. 4. we show an schema of how the dependencies are distributed according to the patterns: A) Blackboard; B) 8-Layers defining the 8 levels as proposed in [26]; C) 3-Layers defining the 3 levels as proposed in [6], and D) a Control-loop as defined in [26].

The results of the evaluation are presented in Table 2. According to the coupling metric, we observe that those alternative *i** architecture models where there are more components and these components have dependencies with few other ones, score better for coupling (e.g., Layered architectures, being 8 levels better than 3). On the other hand, those alternative *i** architecture models where there are less dependencies for data interchange between different components, score better for cohesion (e.g., the Control loop architecture is more cohesive than the Layered architectures). Therefore, the solution that provides a better trade-off of this aspects is the Blackboard pattern.

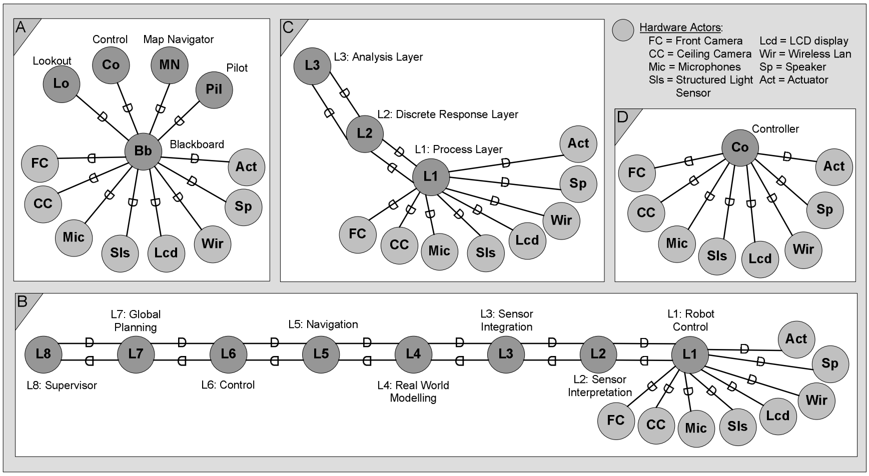


Fig. 4. Schema of the generated alternative *i** architecture models

Table 2. Evaluation results for the metrics indicating cohesion and coupling over 4 different architectural styles

Property	Blackboard pattern	8-Levels layered architecture	3-Levels layered architecture	Control-loop architecture
Coupling	0.6250	0.5814	0.6065	0.8125
Cohesion	0.5217	0.1611	0.4000	0.95

7 Conclusions and Future Work

In this paper we present a set of guidelines for the generation and evaluation of alternative architectures. Our proposal uses the *i** framework, a goal-oriented modelling language that represents the software architecture functional and non-functional requirements using actors and dependencies between them. The guidelines assume that an initial source *i** architecture model and a set of relevant quality attributes have been obtained previously to the execution of the guidelines. From this point of view, we address the generation and evaluation of alternatives by adapting existing architecture solutions to the *i** framework by generating generic *i** architectural patterns. The elements on those patterns are analysed and matched against the ones on the source *i** architecture model in order to obtain the alternative *i** architecture models. Finally, these models are evaluated by applying structural metrics, which are defined by following a set of guidelines that follows the Goal Question Metric paradigm [1]. This process is supported by J-PR*i*M [18].

Among the benefits of the proposed approach we remark the following three. First, architectures are modelled at the early stages of the requirements process using a goal-oriented language, which we believe reduces the gap that is usually found between requirements and architectures. Second, it allows applying structural metrics directly on the requirements model, allowing the evaluation of alternative architectures without having to build any other artefact. Finally, as we are representing architecture-related concepts at the requirements level, we can benefit from the contributions and experience on both the use of the *i** framework and the research on the generation and evaluation of alternatives.

Regarding the capabilities to deal with the modelling, generation and evaluation of software architectures, our process satisfies the desiderata proposed in [26] as follows:

- **Composition.** The *i** framework allows describing a system as a composition of independent components and connections, where the components are represented by actors and the connections are represented by means of dependencies between these actors.
- **Abstraction.** The *i** framework allows describing the components and their interaction at different abstraction levels. Thus, the system can be represented as a unique software actor or as a set of software actors representing the components of the software architecture.
- **Reusability.** Reusability is achieved at two levels. On the one hand, generic *i** architectural patterns are created only once for each architectural solution and can be used in other applications of the process. On the other hand, generated *i** architectures can be used as the source *i** architecture model in further iterations of the process.
- **Configuration.** The generated *i** architectures are based on existing architectural solutions, which clearly states that the system structure is independent from the elements being structured.
- **Heterogeneity.** It is possible to combine several architectural descriptions modelled within the *i** framework, and also to switch the level of detail they represent (for instance, from the whole system to the representation of architectural patterns or architectural styles).
- **Analysis.** We propose to analyse the resulting *i** models using structural metrics as proposed in [14], however other analysis techniques within the *i** framework can be used. They can be based on the structural properties of the *i** framework [2], [12], [13], or based on the reasoning capabilities it provides [27], [28].

As future work, we aim at creating a catalogue of generic *i** architectural patterns and a catalogue of reusable structural metrics. We are interested in stating which types of architectural attributes can be evaluated with structural metrics, and how to define them and use them in a simple way in order to make the evaluation of alternatives more systematic. Although the use of J-PR*i*M has been adequate for supporting the development of the Home Service Robot case study, more experimentation will be done in order to provide accurate data on the effort and benefits of using this approach in industrial case studies.

Acknowledgements. This work has been partially supported by the CICYT programme project TIN2004-07461-C02-01. Gemma Grau work is supported by an UPC research scholarship.

References

1. Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. Encyclopedia of Software Engineering. Wiley, Chichester (1994)
2. Bastos, L.R.D., Castro, J.F.B.: Enhancing Requirements to derive Multi-Agent Architectures. In: Proceedings of WER 2004, pp. 127–139 (2004)
3. Bengtsson, P., Bosch, J.: Scenario-based Software Architecture Reengineering. In: Proceedings of the 5th International Conference on Software Reuse, pp. 308–317 (1998)
4. de Bruin, H., van Vliet, H.: Scenario-based Generation and Evaluation of Software Architectures. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 128–139. Springer, Heidelberg (2001)
5. Bryl, V., Massacci, Mylopoulos, J., Zannone, N.: Designing Security Requirements Models Through Planning. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 33–47. Springer, Heidelberg (2006)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. A System of Patterns, vol. 1. John's Wiley & Sons Ltd, Chichester (2001)
7. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Dordrecht (2000)
8. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures. Methods and Case Studies. Addison-Wesley, Reading (2002)
9. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
10. Dobrica, L., Niemelä, E.: A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28(7), 638–653 (2002)
11. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press (1996)
12. Franch, X., Maiden, N.: Modeling Component Dependencies to Inform their Selection. In: Erdogmus, H., Weng, T. (eds.) ICCBSS 2003. LNCS, vol. 2580, Springer, Heidelberg (2003)
13. Franch, X.: On the Quantitative Analysis of Agent-Oriented Models. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 495–509. Springer, Heidelberg (2006)
14. Franch, X., Grau, G., Quer, C.: A Framework for the Definition of Metrics for Actor-Dependency Models. In: Proceedings of RE 2004, pp. 348–349 (2004)
15. Genero, M., Piattini, M., Calero, C.: A Survey of Metrics for UML Class Diagrams. Journal of Object Technology 4(9), 59–92 (2005)
16. Grau, G., Franch, X.: ReeF: Defining a Customizable Reengineering Framework. In: CAiSE 2007. LNCS, vol. 4495, pp. 485–500. Springer, Heidelberg (2007)
17. Grau, G., Franch, X.: On the Adequacy of *i** Models for Representing and Analysing Software Architectures. In: RIGiM 2007 (at ER 2007). Proceedings of the First International Workshop on Requirements, Intentions and Goals in Conceptual Modelling (to appear, 2007)
18. Grau, G., Franch, X., Ávila, S.: J-PR*i*M: A Java Tool for a Process Reengineering i* Methodology. In: Proceedings of RE 2006, pp. 352–353 (2006)
19. Grau, G., Franch, X., Maiden, N.A.M.: A Goal Based Round-Trip Method for System Development. In: Proceedings of REFSQ 2005, pp. 71–86 (2005)
20. Grünbacher, P., Egyed, A., Medvidovic, N.: Reconciling software requirements and architectures with intermediate models. Software and Systems Modeling 3(3), 235–253 (2004)

21. The *i** wiki (last accessed, May 2007), http://istar.rwth-aachen.de/
22. Kang, K.C., Kim, M., Lee, J., Kim, B.: Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets - a case study. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 45–56. Springer, Heidelberg (2005)
23. Kim, M., Lee, J., Kang, K.C., Hong, Y., Bang, S.: Re-engineering Software Architecture of Home Service Robots: A Case Study. In: Inverardi, P., Jazayeri, M. (eds.) ICSE 2005. LNCS, vol. 4309, pp. 505–513. Springer, Heidelberg (2006)
24. Kolp, M., Giorgini, P., Mylopoulos, J.: Organizational Patterns for Early Requirements Analysis. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 617–632. Springer, Heidelberg (2003)
25. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Proceedings of ISRE 2001, pp. 249–263 (2001)
26. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
27. Yu, E.: Modelling Strategic Relationships for Process Reengineering. PhD. thesis, University of Toronto (1995)
28. Yu, E.: Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In: ISRE 1997. 3rd IEEE Intl. Symposium on Requirements Engineering (1997)

Hierarchical Verification in Maude of L*f*P Software Architectures

Chadlia Jerad[1], Kamel Barkaoui[2], and Amel Grissa Touzi[1]

[1] LSTS - ENIT, BP 37, le Belvedere 1002 Tunis, Tunisia
jerad.chadlia@gawab.com, amel.touzi@enit.rnu.tn
[2] CEDRIC - CNAM, 292, Rue Saint-Martin Paris 75003, France
barkaoui@cnam.fr

Abstract. Software architecture description languages allow software designers to focus on high level aspects of an application by abstracting from details. In general, a system's architecture is specified in a hierarchical way. In fact, hierarchical components hide, at each level, the complexity of the sub-entities composing the system. As rewriting logic is a natural semantic framework for representing concurrency, parallelism, communication and interaction, it can be used for systems specification and verification. In this paper, we show how we can take advantage of hierarchical modeling of software systems specified with L*f*P, to prototype model checking process using Maude system. This approach allows us to hide and show, freely and easily, encapsulated details by moving between hierarchical levels. Thus, state explosion problem is mastered and reduced. In addition, system's maintainability and error detection become easier and faster.

1 Introduction

During the past decade, architectural design has emerged as an important subfield of software engineering. In fact, a good architecture can help ensure that a system will satisfy key requirements. Consequently, a new discipline emerged, which concerns formal notations for representing and analyzing architectural designs: Architecture Description Languages (ADL) [1]. ADL allow to model hierarchical components systems. These systems make visible the hierarchy and hide, at each level the complexity of the sub-entities. The compositional aspect together with the separation between functional (behavior hierarchical modeling) and non functional aspects help the implementation and maintenance of complex software systems. After modeling step, designers want then to make sure that the built application is safe. Rewriting Logic (RL) formalism has been used in the verification of systems models [2]. Maude is an executable specification language based on rewriting logic, that includes a Linear Temporal Logic (LTL) model checker as a module [3].

Hierarchical modeling of functional aspects can be seen as behavior refinement, since there are visible and hidden behaviors. Many works dealt with the notion of refinement. Particularly, we mention the work of Malcolm and Goguen

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 156–170, 2007.

in [4] that concerns refinement and rewriting logic. Their work combines order sorted and hidden sorted equational logics. However, our work is about behavior refinement in rewriting logic rules.

In this paper, we show how we can take advantage of hierarchical modeling of distributed software systems specified in L*f*P, Language for Prototyping [5], to prototype model checking process using Maude system. L*f*P is a formal approach dedicated to the description of distributed software architectures. We chose L*f*P because, in this language, functional aspects are hierarchically defined. In [6] and [7], we have shown, respectively, how we can translate L*f*P semantics into RL by a systematic mapping and how we use Maude's model checker in order to verify properties expressed in LTL. Unfortunately, this mapping does not preserve L*f*P description hierarchical aspects. The following work fills this lack and allows to perform, what we call "hierarchical model checking". Using this technique, we can hide or show encapsulated details in a free and easy way when performing the verification process, by moving between hierarchical levels. Moreover, state explosion problem is mastered and reduced and error detection become easier and faster.

This paper is organized as follows: in section 2, we recall basic concepts related to ADL and RL. Section 3 introduces hierarchical modeling of distributed systems using L*f*P language. In section 4, we present the rules of the new mapping of L*f*P semantics into rewriting logic. Hierarchical model checking principles, advantages and concretization using Maude are detailed in section 5. In section 6, we conclude the paper and give some perspectives about further work.

2 Basic Concepts

In this section, we introduce basic concepts of architecture description languages and rewriting logic.

2.1 Architecture Description Languages

Software architecture plays typically a key role as a bridge between requirements and code. By providing an abstract description (or model) of a system, the architecture exposes certain properties, while hiding others. Initially, the architectural design was largely an ad hoc affair [1]. One development has been the creation of formal notations for representing and analyzing architectural designs. ADL usually provide both a conceptual framework and a concrete syntax for characterizing software architectures [1,8]. The use of ADL offers several advantages. Indeed, ADL allow: the reuse of architectural design specifications, rapid prototyping, supports use for evolution and re-engineering and better understanding of architectural designs through early errors detection, and quality of service analysis [1,8].

In hierarchical component frameworks, different components can be assembled together creating a new self contained component which can be itself assembled to other components in upper level of hierarchy. Hierarchical components make visible the hierarchy of the system and hide, at each level the complexity of the sub-entities. The compositional aspect together with the separation between functional and non functional aspects help the implementation and maintenance of complex software systems.

A number of ADL have been proposed for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages. As example of ADL, we note Rapide [9], Wright [10], and L*f*P [11]. We find L*f*P particularly interesting. In fact, its methodology is relevant, it allows the hierarchical modeling of functional features and a code generator have been developed.

2.2 Rewriting Logic

The theory of rewriting logic has proved to be very useful to find a unifying framework for concurrency formalisms [12]. In [13], Meseguer showed in a number of examples that it is suitable as a common framework for concurrency. For rewriting logic, the entities in question are concurrent systems having states and evolving by means of transitions. Rewrite rules in the theory describe which elementary local transitions are possible in the distributed state by concurrent local transformations [14]. In object-oriented rewriting logic, object-oriented message-passing style and functional styles can be freely mixed. Messages are then exchanged between objects, and cause communication events by application of rewrite rules. Concurrent rewriting modulo structural axioms of associativity, commutativity and identity capture abstractly the essential aspects of communication in a distributed object oriented configuration made up of concurrent objects and messages [14].

In recent years, several executable specification languages based on RL have been designed and implemented. Maude system is an efficient tool for systems' specification and analysis based on rewrite logic. Typically, Maude specifications are executable [3]. Maude system provides a syntax for representing object oriented concepts. Maude's object oriented syntax models the concurrent state of the system (or configuration) as a multi-set of objects and messages. Objects and messages are singletons of type configuration. Complex structures are, thus, represented as union of configuration singletons. The concurrent interactions between the objects are controlled by rewriting rules. We assume readers to be familiar with Maude system. For details about Maude's syntax, readers may refer to [3]. In Maude system, we can use properties specifications, expressed in LTL and implemented model checking procedures, to check properties in a given module starting from an initial state [3]. Indeed, the LTL model checker is described and specified as a Maude's module.

3 Hierarchical Modeling of Distributed Systems with L*f*P

When dealing with distributed architectures, we have to tackle with structure control problem. L*f*P [11,5,15], which was developed within the MORSE project [16], is a formal approach to distributed software architectures. L*f*P is inspired from RM-ODP reference model [17]. This language models the composition of the system (non functional aspects) through a set of interacting components, while functional aspects are hierarchically defined. In this section, we present L*f*P language and its hierarchical structuring elements through an example.

3.1 L*f*P Methodology

L*f*P methodology [5] is a model-based development. This approach aims at implementing evolutionary prototyping capabilities. L*f*P model can be partially generated from UML standard diagrams. However, it contains enriched information compared to other UML diagrams. Once the L*f*P model is produced, system synthesis can be performed. When all properties stated in the L*f*P model are verified, the code generator produces pieces of programs to be compiled and deployed in the target execution environment [5]. L*f*P specifications are a set of hierarchical diagrams representing the behavior of models' components. Each L*f*P model uses two complementary diagrams: L*f*P-Architecture Diagram (L*f*P-AD) and L*f*P-Behavior Diagrams (L*f*P-BD) [15].

We consider the example of a distributed task manager system. Let us first introduce our example with the UML class diagram of figure 1 that shows the main model components. This system is composed of a task scheduler machine, and three units: Unit1, Unit2 and Unit3. Each Unit performs a specific job on a worksheet given by the task scheduler machine. When done, the unit returns the worksheet back to the machine with an evaluation mark, which is an integer describing the result. The task scheduler schedules the workflow between the different units. After receiving the worksheet from a unit, it sends it to the next unit (in a circular way) if the evaluation mark is positive, while it returns it to the sending unit if the evaluation mark is negative.

The `TaskScheduler` class declares two methods. The `receive` method allows to get from a unit the worksheet and the evaluation mark. The `schedule` method determines to which unit the machine should send the worksheet. `Unit` is an abstract class that declares four methods. The `receive` method gets the worksheet from the task scheduler machine. The `send` method allows to send the worksheet and the evaluation mark to the machine. The `performJob` method performs the job on the given worksheet and the `computeEvalMark` computes

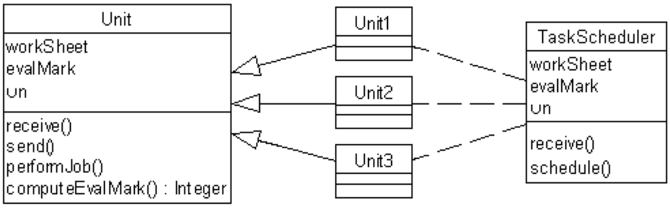


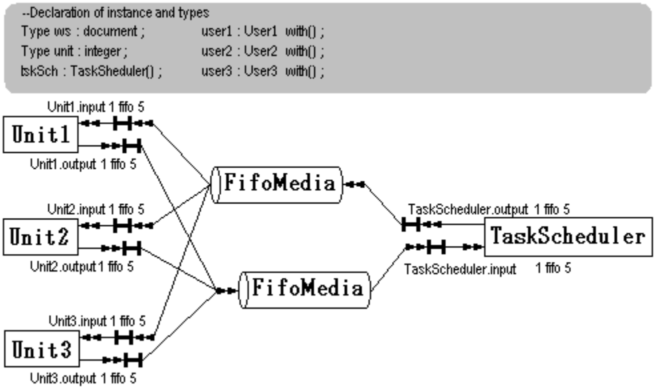
Fig. 1. UML class Diagram of the task manager system

and returns the evaluation mark of the process. `Unit1`, `Unit2` and `Unit3` classes inherit from the class `Unit`. Each one implements its own methods `performJob` and `computeEvalMark`.

3.2 L*f*P Architecture Diagram

The L*f*P-AD [5,15] presents the different components constituting the system and the relations among them. It is composed of three main elements: classes, medias and binders. Classes and medias are the L*f*P components. L*f*P classes represent the applicative model components. They correspond to an UML implementation class and express some functional aspects of the system. L*f*P medias connect instances of L*f*P classes and correspond to UML associations, aggregations or compositions. They specify binding constraints and communication protocol semantics. The execution contract of L*f*P components is then specified by the means of L*f*P-BD. L*f*P binders are buffer messages allowing the connection between classes end medias. Binding constraints specify: (i) a reference to the connected binding point, (ii) communication mode (synchronous or asynchronous), (iii) ordering policy (fifo or bag) and (iv) binding multiplicity. L*f*P ports are the physical connection points of binders.

We recall here the example of the task manager system. We now reformulate the UML specification in order to obtain the backbone of the L*f*P specification. Figure 2 shows the obtained L*f*P-AD. This diagram contains `Unit1`, `Unit2`, `Unit3` and `TaskScheduler` classes, in addition to two new components `FifoMedia`, which correspond to association relations. These medias encapsulate the communication protocol. We notice that similar binders link classes to medias (synchronous, multiplicity 1, fifo, 5). The L*f*P-AD also contains a declaration part in which we specify classes instances.



```
--Declaration of instance and types
Type ws : document ;              user1 : User1  with() ;
Type unit : integer ;             user2 : User2  with() ;
tskSch : TaskSheduler() ;         user3 : User3  with() ;
```

Fig. 2. L*f*P-AD of the task manager system

3.3 L*f*P Behavior Diagrams and Hierarchical Structuring Elements

L*f*P-BD [5,15] specifies, in a hierarchical way, the execution contract of each component. The global structure of the L*f*P-BD corresponds to a state/transition automaton. L*f*P-BD contains: (i) a declarative part and (ii) the diagram itself. It contains basic elements (nodes) wired together. The basic elements of L*f*P-BD are: states, transitions, instantiation transitions and communication transitions. There are two kinds of transitions: simple and hierarchical. Hierarchical transitions model calls to L*f*P-triggers and L*f*P-methods. The L*f*P-trigger is someway like a private method of a class, but should not be considered as one. In one hand, it collects a set of transitions that can be called in different points of a behavior diagram. In the other hand, it does not declare parameters or a returning value, and can not be used to model recursion. The L*f*P-method is someway like a public method of a class. It is activated after the reception of the needed message. Consequently, L*f*P-methods are used for modeling remote procedure calls or message communication. However, L*f*P-method can not contain calls to other L*f*P-methods. Communication transitions express communication events as messages sending and/or reading. Table 1 summarizes the graphical representation of L*f*P-BD nodes.

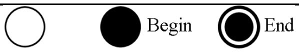
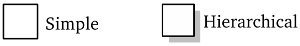
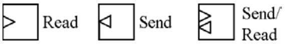
Systems hierarchical modeling with L*f*P concerns functional aspects, while non functional aspects are not specified in a hierarchical way. These functional aspects are specified through L*f*P-BD.

Definition 1. *A sub-LfP-Behavior Diagram is an LfP-BD that defines the contents of a hierarchical transition [5].*

Hierarchical structuring elements of L*f*P-BDs are sub-L*f*P-BDs decomposing hierarchical transitions. In sub-L*f*P-BD, designers can define local variables, as well as instructions. This automata is itself likely to contain sub-diagrams. Thus, we obtain several hierarchy levels.

We illustrate the use of L*f*P-BD for components behavior specification through the task manager system. However, for simplicity reasons, we will focus only on `TaskScheduler` component. The behavior contract of this class is specified using L*f*P-BD (figure 3). The L*f*P-BD contains a declarative part specifying ports, variables and a list of methods and triggers, in addition to the diagram itself. In a hierarchical way, we specify the behavior of each of the methods/triggers

Table 1. L*f*P-BD nodes representation

Symbol node	Graphical representation
State	
Transition	
Instantiation	
Communication	

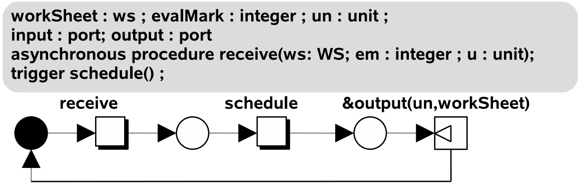
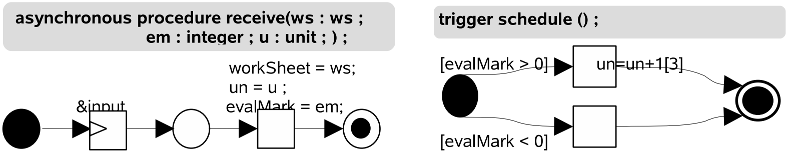

workSheet : ws ; evalMark : integer ; un : unit ;
input : port; output : port
asynchronous procedure receive(ws: WS; em : integer ; u : unit);
trigger schedule() ;
receive
schedule
&output(un,workSheet)

Fig. 3. L*f*P-BD of `TaskScheduler` class



(a) `receive` method (b) `schedule` trigger

Fig. 4. L*f*P-BD of `receive` method and `schedule` trigger

declared in `TaskScheduler` class. Figure 4-a and figure 4-b show the execution contracts of, respectively, `receive` method and `schedule` trigger.

4 Mapping of L*f*P Semantics into RL

In this section, we present the mapping of L*f*P semantics into rewriting logic semantics with Maude language. A previous work concerning the mapping has been presented in [6], but without handling behavior diagrams hierarchical structure. We resolve this lack in this paper.

4.1 Mapping of L*f*P Basic Concepts

In order to express the mapping operation with respect to L*f*P language concepts, we created module `LfP` that contains the declaration of basic concepts. This module will be included in all the other modules. Figure 5 shows the `LfP` module.

In `LfP` module, we declare the sort `Port`, for L*f*P ports, the sub-sorts `ClassCid` and `MediaCid` for L*f*P components class identifiers and the sub-sorts `ClassOid` and `MediaOid` for L*f*P components object identifiers. We also declare the messages `LfPMsg` and `RetMsg`. However, messages declarations are specific to the considered application. Thus, it may change from a system to another. Three operations are declared in this module. `BDState` operation is a subsort of `Configuration` and indicates in which state (indicated by an integer) is the given object. `BinderCM` is a binder connecting a class to a media, while `BinderMC` is a binder connecting a media to a class. The parameters of these two operations

```
mod LfP is
  pr INT . pr STRING . inc CONFIGURATION .
  sorts Port ClassCid MediaCid ClassOid MediaOid .
  subsorts ClassOid MediaOid < Oid .
  subsorts ClassCid MediaCid < Cid .
  op LfPMsg RetMsg : ClassOid Port String Int -> Msg .
  op BDState  : Oid Int -> Configuration .
  op BinderCM : ClassOid Port MediaOid Int Int -> Configuration .
  op BinderMC : MediaOid ClassOid Port  Int Int -> Configuration .
  ...
endm
```

Fig. 5. `LfP` module

include the media, the class and its communication port, the capacity of the binder and the number of messages ”actually” in it.

4.2 Mapping of L*f*P-Architecture Diagram

Since rewriting logic allows modeling concurrent objects, L*f*P-AD is mapped into object-oriented rewrite theories in rewriting logic. Table 2 summarizes this mapping. We notice that this mapping preserves the original modularity of the L*f*P-AD.

Binders are responsible for message passing from classes to medias and vice-versa. Therefore, we mapped the ordering policy of binders into rewrite rules and conditional rewrite rules. Indeed, a message goes through four transfers: class-binder, binder-media, media-binder and binder-class. These transfers are modeled by rewrite rules that describe messages' state changes. For this purpose, we define the state of a message by the specification of its location (class, media or binder) and its function (sent or read). Possible states are: ”`S`” (the message is in the sender class), ”`SB`” (the message is in the sender binder), ”`SMIn`” (the message entered the media), ”`RMOut`” (the message is ready to be transferred to the binder), ”`RB`” (the message is in the reader binder) and ”R” (the message is in the reader class).

Table 2. Mapping of L*f*P-AD into RL

L*f*P-AD	RL
L*f*P-AD	Object-oriented theories
L*f*P class	Object module
L*f*P media	Object module
L*f*P binder	Terms having the operations `BinderCM` or `BinderMC` on top
Ordering policy	Rewrite rules on Messages
Component instance	Object

4.3 Mapping of LfP-Behavior Diagram

LfP-BD specifies the transition of the system from one state to another, in response to certain events. Thus, we give an identification number to each LfP-state . The identification number of the LfP-state "Begin" (respectively "End") belonging to a sub-LfP-BD must be equal to the identification number of the LfP state source (respectively target) of the corresponding hierarchical transition. LfP states are mapped into terms having the operation `BDState` on top. Since LfP transitions firings cause state changes in the system, they are mapped into conditional and/or non-conditional rewriting rules. These rules describe the creation, modification and destruction of messages and objects. The mapping of LfP-BD nodes is done according to the rules appearing in table 3.

Table 3. Mapping of LfP-BD into RL

LfP-BD	RL
LfP-BD of an LfP component	Set of rewrite rules (conditional or/and unconditional)
LfP-BD state	Term having the operation `BDState` on top
LfP-BD transition	Rewrite rule
LfP-BD transition with a guard	Conditional rewrite rule
LfP-BD read transition	`LfPMsg` message at the state "R"
LfP-BD send transition	`LfPMsg` message creation at the state "S"
LfP-BD send/read transition	`RetMsg` messages at the states "S" and "R"
LfP-BD class instantiation	Creation of an instance in the rule

All the obtained rules are then grouped in a single module importing all the other modules obtained from the mapping of the LfP-AD. Accordingly, this module will be the specification of the whole system in rewriting logic semantics.

4.4 Rules for Mapping LfP Transitions

In Maude system, rewriting rules can have labels. In our mapping process, we will use these labels in order to store some information about corresponding transitions. These information are needed for performing hierarchical verification. We adopt the syntax in figure 6 for writing rules' labels. The rule label parameters are: (i) `HLevel`, which is an integer indicating the hierarchical level of the corresponding transition (in the next section, we give the definition of the hierarchical level of a given transition (definition 4)), (ii) `TType`, which is a character indicating the type of the corresponding transition and (iii) `AddLabel`, which is an additional label that users may want to add. `TType` takes either the value 'S', if the transition is simple, or 'H' if the transition is hierarchical. We note that the string `AddLabel` must not contain the sub-strings "-S-" or "-H-" to avoid confusion with `TType` value. In the same way we find out the label parameters of a conditional rule corresponding to an LfP guarded transition.

It is important to notice that the final module will contain the description of all the system hierarchy levels. In addition, the transfer from the initial state to

```
rl [HLevel-TType-AddLabel] :
     M1 ... Mn < O1 : F1 | at1 > ... < Om : Fm | atm >
  => M'1 ... M'q < Oi1 : Fi1' | ati1' > ... < Oik : Fik' | atik'>
     < Q1 : D1 | at"1 > ... < Qp : Dp | at"p > .
```

Fig. 6. Rule syntax

the final state in each sub-L*f*P-BD will be described twice. Indeed, we have a direct transfer resulting from mapping the corresponding hierarchical transition and a detailed transfer resulting from the mapping of the sub-L*f*P-BD.

4.5 Example

In this section, we recall the example of the task manager system introduced in section 3. After applying our mapping, we obtain seven rewriting rules for the `TaskScheduler` class. We note that before writing the rules, we gave to L*f*P-BD states the identification numbers 1, 2 and 3, from left to right. Figure 7 shows the rules corresponding to the mapping of its L*f*P-BDs (rules `[0-H-receive]`, `[0-H-schedule]` and `[0-S-]`) and the L*f*P-BD of the `schedule` trigger (rules `[1-S-]` and `[1-S]`). We note that we do not present the whole rules' syntax for clarity reasons.

We notice that we can easily regenerate the L*f*P-BD from this set of rewriting rules described with Maude.

5 Hierarchical Model Checking in L*f*P Using Maude

In our work, we take advantage of L*f*P-BD hierarchical structuring to set our approach of *prototyping* verification process by applying hierarchical model checking.

```
*** TaskScheduler LfP-BD rules
rl [0-H-receive] : < TS : TaskScheduler | > BDState(TS, 1)
  => < TS : TaskScheduler | > BDState(TS, 2) .
rl [0-H-schedule] : < TS : TaskScheduler |  > BDState(TS, 2)
  => < TS : TaskScheduler | > BDState(TS, 3) .
rl [0-S-] : BDState(TS, 3)
  < TS : TaskScheduler | output : out , un : u , workSheet : wsh >
  => < TS : TaskScheduler | output : out , un : u , workSheet : wsh >
  BDState(TS, 1) LfPMsg(TS, out, "S", 0, u, wsh) .
*** schedule LfP-BD rules
crl [1-S-] : < TS : TaskScheduler | un : u , evalMark : N > BDState(TS, 2)
  => < TS : TaskScheduler |  un : s(u) , evalMark : N > BDState(TS, 3)
  if N < 0 .
crl [1-S-] : < TS : TaskScheduler | un : u , evalMark : N > BDState(TS, 2)
  => < TS : TaskScheduler |  un : u , evalMark : N > BDState(TS, 3)
  if N > 0 .
```

Fig. 7. Mapping of `TaskScheduler` L*f*P-BD and `schedule` L*f*P-BD

5.1 Hierarchical Verification Principles

In order to introduce hierarchical verification principles, we first present definitions that we settled for hierarchical systems modeling using LfP.

Definition 2. *A given LfP-BD is of hierarchical level* 0 *if it is the LfP-BD of an LfP component.*

In recursive way, we define LfP-BD of hierarchical level N, $N \in \mathtt{N}^{+} \setminus \{0\}$.

Definition 3. *A given LfP-BD is of hierarchical level N, $N \in \mathtt{N}^{+} \setminus \{0\}$ if it is a sub-LfP-BD of hierarchical transition belonging to an LfP-BD of hierarchical level* $(N-1)$.

In LfP-BD, we have two types of transitions: simple, and hierarchical. In our work, we add another criterion for characterizing transitions, which is the hierarchical level. Using the two last definitions (2 and 3), we give the definition of an LfP-BD transition of hierarchical level I.

Definition 4. *A given LfP-BD transition is of hierarchical level I if it belongs to an LfP-BD of hierarchical level I.*

Now we give the definition of hierarchical verification.

Definition 5. *The hierarchical verification at level N of a given system's model DS specified in LfP, is the verification of the sub-system containing only the mapping of LfP-BD of hierarchical level M, $0 \leq M \leq N$, after removing hierarchical aspects (this means that we perform verification in the flattened subsystem).*

From the definition 5, we deduce that hierarchical verification at level N means that we make abstraction of all the LfP-BD which hierarchical levels are strictly upper than N. Thus, details of the hierarchical level $N+1$ and upper are not visible. We can express this as details encapsulation at verification time. At the same time, we keep only the details of the hierarchical levels 0 to N and without redundancy.

Let DS be the LfP specification of a distributed system S. Let N be a level of DS's hierarchy. By abstraction, we derive the following properties:

Property 1. If DS satisfies a given property φ at level N, then at each level M, $0 \leq M \leq N$ where φ can be expressed, DS satisfies the property φ at that level M.

Property 2. If DS satisfies a given property φ at level N and does not satisfy φ at level N', $N' > N$, then the error related to φ is located between the levels $N+1$ and N'.

5.2 Hierarchical Verification Advantages

Our new approach of hierarchical verification makes abstraction of the encapsulated details in upper levels. Thus, the obtained model to check contains less rules than the initial model. Consequently, the state explosion problem is mastered and reduced. In addition, errors detection can be performed at earlier stages of specification, even before building the whole model. In addition, errors localization is much easier since we have a reduced number of suspicious rules. Moreover, hierarchical verification has the advantage of maintainability and the re-factoring. Indeed, if a given system satisfies a property φ at level N and we make a modification at a level M, $M > N$, the system is still satisfying φ at level N. In this way, the verification of the new system will be faster and easier.

5.3 Hierarchical Modules Extraction in Maude

In order to implement hierarchical model checking in Maude, we need to build a new module M' of hierarchical level N, starting from the initial module M containing all the rules describing the system. We first give the definition of a module of hierarchical level N (definition 6).

Definition 6. *A module M' of hierarchical level N is a submodule of M flattened down to level N.*

We establish the following relations between the rules of the starting module M and the rules of the built module M'. Let R be a rule in the module M corresponding to an L*f*P-transition T of hierarchical level Tl.

- If $Tl > N$, the rule R will not belong to M'.
- If $Tl = N$, the rule R will belong to M'.
- If $Tl < N$, we have two cases :
  - If T is hierarchical or method, the rule R will not belong to M'.
  - And if T is simple, the rule R will belong to M'.

In Maude, we can make use of META-LEVEL functions to create new strategies [3]. In our case, we need an operation to remove certain rewrite laws according to the given level. The functional module `HIERARCHY` (figure 8) allows us to create modules at a given hierarchical level. This module imports the modules `META-LEVEL` and `CONVERSION` and declares the operations `name`, `getHLevel` and `rmvSH`. The operation `name` gives the label of a given rule, while `getHLevel` gives its hierarchical level. `rmvSH` operation gives a set of rules, according to the relations we just detailed. Another operation is declared in this module, which is `metaHRed`. This operation expresses the strategy that creates a hierarchical module at the given level (of sort `Qid`), and calls upon the default strategy (in metaReduce) to reduce the given term according to the new module rules.

```
fmod HIERARCHY is
  pr META-LEVEL . inc CONVERSION .
  op name : Rule -> String .
  op getHLevel : Rule -> Float .
  op rmvRH : RuleSet Qid -> RuleSet .
  op metaHRed : Module Term Qid -> ResultPair .
  ***Variables declaration...
  eq name(rl T1 => T2 [label(Q) A] .) = string(Q) .
  eq name(crl T1 => T2 if C [label(Q) A] .) = string(Q) .
  eq getHLevel(R) = float(substr(name(R),0, find(name(R),"-",0)) + ".0") .
  eq rmvRH(R RS, Q) = if getHLevel(R) > float(string(Q) + ".0")
     then rmvRH(RS, Q)
     else if getHLevel(R) == float(string(Q) + ".0") then R rmvRH(RS, Q)
          else if find(name(R),"-S-",0) == notFound then rmvRH(RS, Q)
               else R rmvRH(RS, Q)
     fi fi fi .
  eq rmvRH(none, Q) = none .
  eq metaHRed(mod N is I sorts S . SS O MS ES RS endm, T1, Q)
     = metaReduce(mod N is I sorts S . SS O MS ES rmvRH(RS,Q) endm, T1) .
endfm
```

Fig. 8. The functional module `HIERARCHY`

5.4 Hierarchical Model Checking Application with Maude

In this section, we assume readers to be familiar with Maude's LTL model checker [3]. In order to check our L*f*P specification, we create a new module. This module imports the module describing the whole system, `MODEL-CHECKER` module and `LTL-SIMPLIFIER` module. The sort `Configuration` should be declared as subsort of `Prop` and `State`. We then declare the operation `propertyToCheck` of kind `ModelCheckResult`, that calls the `modelCheck` operation on given `State` and `Formula`.

Let's take the example of our task manager system. We need to check if for a certain `TSch TaskScheduler` constant object, the third state is reachable or not. For thispurpose, we declare the initial state and the property to check:

```
op position : Oid Int -> Prop .
eq BDState(O, N) C:Configuration |= position(O, N) = true .
eq initial = < TSch: TaskScheduler | ..., evalMark: 0 > BDState(TSch, 1) .
eq final = position(TSch, 3) .
op propertyToCheck : -> ModelCheckResult .
eq propertyToCheck = modelCheck(initial, [] ~ final) .
```

Hierarchical verification of this property at level 0 and at level 1 gives the following results.

```
Maude> red in HIERARCHY : metaHRed(['TASK-MANAGER-CHECK],
              'propertyToCheck.ModelCheckResult, '0) .
rewrites: 16
result ResultPair:'counterexample['...]
```

```
Maude> red in HIERARCHY : metaHRed(['TASK-MANAGER-CHECK],
              'propertyToCheck.ModelCheckResult, '1) .
rewrites: 15
result ResultPair:'true.Bool,'Bool
```

The execution shows that the property (the third state is not reachable from the first one) is satisfied at level 0 since we obtain a counter example. However, the property is not satisfied at level 1. The error is then localized at level 1 (error localization). If we recall the rewrite rules of that level in figure 7 and the state `initial`, we will find that the case where the attribute `evalMark` is equal to zero is not undertaken. Consequently, we have to go back to the L*f*P-BD of the method `schedule` to correct our L*f*P specification (faster error detection). If the made corrections concern only this diagram, then we do not need to check the property at level 0. We directly move to level 1 checking (maintainability).

6 Conclusion

The abstraction in architecture description notations makes them suitable for model checking. In our work, we took advantage of meta-programming power with Maude to set our approach for hierarchical model checking of distributed software systems specified in L*f*P. This approach aims at prototype the verification process. It has the advantages of mastering and reducing the state explosion problem by freely hiding and showing encapsulated details while moving between hierarchical levels. In addition, the system's maintainability and error detection become easier and faster. Moreover, our work demonstrates the expressive power of rewriting logic. In the future, we will study hierarchical model checking by modules composition.

References

1. Garlan, D.: Software architecture. Encyclopedia of Software Engineering, School of computer science, Carnegie Mellon University (2001)
2. Leucker, M.: Rewriting logic as a framework for building generic tools for verifying concurrent systems. RWTH Aachen, Germany (1998), URL: `citeseer.ist.psu.edu/leucker98rewriting.html`
3. Clavel, M., et al.: Maude manuel (version 2.1) (March 2004)
4. Malcolm, G., Goguen, J.: Proving correctness of refinement and implementation. Technical report, Oxford University (January 1996)
5. Gilliers, F.: Développement par prototypage et Généation de Code à partir de LfP, un langage de modélisation de haut niveau. PhD thesis, Pierre et Marie Curie University, Paris VI, France (2005)
6. Jerad, C., Barkaoui, K.: On the use of rewriting logic for verification of distributed software architecture description based l*f*p. In: Proceedings the 16th IEEE International Workshop on Rapid System Prototyping, Montreal, Canada, June 2005, pp. 202–208. IEEE Computer Society Press, Los Alamitos (2005)
7. Jerad, C., Barkaoui, K., Grissa Touzi, A.: Vérification des architectures de systmes distribués par utilisation du ltl model checker de maude. In: 8th African Conference on Research in Computer Science, Cotonou, Benin, pp. 99–106 (November 2006)

8. Richard, N., Medvidovic, N., Taylor: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1) (January 2000)
9. Kenney, J.J.: Executable Formal Models of Distributed Transaction Systems Based on Event Processing. PhD thesis, Department of Electrical Engineering, Stanford University, USA (June 1996)
10. Allen, R.J.: A Formal Approach to Software Architecture. PhD thesis, School of Computer Science, Carnegie Mellon University, USA (May 1997)
11. Regep, D., Kordon, F.: L*f*P: A specification language for rapid prototyping of concurrent systems. In: 12th International Workshop on Rapid System Prototyping, Monterey, California, pp. 90–96 (2001)
12. Meseguer, J.: Rewriting as a unified model of concurrency. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 384–400. Springer, Heidelberg (1990)
13. Meseguer, J.: Rewriting logic as a semantic framework for concurrency. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, Springer, Heidelberg (1996)
14. Marti-Oliet, N., Meseguer, J.: Rewriting logic: Roadmap and bibliography. Theoretical Computer Science (June 2001)
15. Gilliers, F.: Description de la sémantique du langage LfP. Work document, RNTL MOSE Project MORSE-SRS-031114-V0.15-FGI, Pierre et Marie Curie University, LIP6, France (June 2003)
16. MORSE project web site, `http://www.lip6.fr/morse/`
17. Wegmann, A., Naumenko, A.: Conceptual modelling of complex systems using an rm-odp based ontology. In: 5th International Enterprise Distributed Object Computing Conference, Seattle, USA, pp. 27–34 (September 2001)

First Class Connectors for Prototyping Service Oriented Architectures

Kristian Ellebæk Kjær

Department of Computer Science, University of Aarhus
Åbogade 34, 8200 Århus N., Denmark
argo@daimi.au.dk

Abstract. Prototyping Service Oriented Architectures based on web services is a complex and time consuming process. Several steps, some of them complicated, are required for even the simplest application. Therefore, it is desirable to be able to create prototypes using more familiar constructs, entirely within a single programming language, and then define some of the components as web services, and others as clients.

We present a framework which enables programmers to create web services and clients in ArchJava, an extension to Java which supports components and connectors as first class entities, by defining interfaces to services as ports on components. This supports rapid creation of prototypes by defining a component and connector structure of a web service based system in ArchJava, and then later, with only minor modifications, change the prototype to use web services. The services and client components will not be aware of this. From their point of view they are talking through connectors. Client components can also be connected to existing web services.

1 Introduction

Web services have become a popular way to integrate existing systems in Service Oriented Architectures [1], since they allow services to be created as an additional layer between the system and the internet and allow for connecting services provided by different administrative domains. Furthermore, they are based on common standards, allowing integration of versatile systems in a convenient manner. However, large systems based on web services are inherently complex, and design time decisions, such as what to service enable and defining interfaces must be decided early in the design process, and later changes can only be carried out at significant cost.

When designing complex systems, prototyping has proven to be an effective method for determining the viability of architectures and functionality. Floyd defines software prototypes as executable systems which demonstrate relevant parts of the software [2], Bardram et. al defines an architectural prototype as a set of executables which are created to investigate relevant architectural qualities [3], and have described the kind of qualities which are suitable for exploration in such prototypes [4]. They describe the concept of architectural prototypes as follows:

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 171–178, 2007.

> An *architectural prototype* consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. *Architectural prototyping* is the process of designing, building, and evaluating architectural prototypes.

Architectural prototypes are not concerned with functionality, which is non-architectural as observed by Bass. et al [5]. Bardram et al also observes, that this is the characteristic which makes architectural prototypes cost-efficient. Equally important, they note that even though architectural prototypes can be build by hand, tool support might be available for a given domain. In our case, we build a framework for constructing such prototypes in an efficient manner.

In a related article by Bardram et al [4], the authors explore what kind of qualities, as defined by Bass et al, are suitable for exploration with architectural prototypes. Among these, the central architectural quality attributes are *conceptual integrity* and *buildability*. The first is related to the architecture being clear, and not eroded. The second is whether the architecture can actually be build in a timely manner. E.g. the prototype may be created to explore new technologies or architectures. Among the system quality attributes, we are especially interested in are *modifiability*, *testability*, and *performance*. Software architecture does not determine these qualities, but have an impact. Architectural prototypes are especially well suited for modifiability analysis, since all design time artifacts are available as executables. Performance can not be completely determined by architectural prototypes, but may establish more or less precise bounds on latency of invocations, depending on how time consuming business logic is.

Building web services for prototyping is complex and expensive, since even non-functional prototypes involves several steps:

1. Setting up at least one application server,
2. defining interfaces as wsdl,
3. generating stubs, skeletons, and types based on the wsdl file,
4. creating an, possibly empty, service implementation,
5. creating a client,
6. creating a service file, and
7. deploying the service file on the application server.

Furthermore, steps 2 through 7 must be repeated for each change in the structure or interfaces. Although parts of this can be automated, it is a cumbersome task.

Is should be clear from the above, that building prototypes is desirable and often necessary to determine key qualities of the system. Rapid prototyping allows several different architectures to be explored.

To enable rapid prototyping, it is desirable to only be concerned with the *structure* of the system. ArchJava is an extension to Java which supports building programs using first class components and connectors [6]. Communication integrity is enforced in the sense that a component can only communicate with its parent, i.e. the component which created it, and with components to which it is explicitly connected through a connection. These features can be utilised for creating prototypes by concentrating on a component and connector view of the prototype as described by Bass et. al [5], and then later perform tests and measurements on the system using web services.

There exists a number of other projects, which support rapid prototyping of software systems, such as SoftArch/MTE [7]. However, most of these are not especially suited for service oriented architectures. However, some prototyping tools for web services do exist. Easy SOA is a tool for rapid prototyping tool for web services, that run inside a web browser [8], and the Rapid Prototyping framework for SOAs provides a message oriented system for interactively handling messages between services, also inside a web browser [9]. However, while both allow for creating the structure of a system, they do not use familiar programming concepts, and assume the existence of descriptions of the service interfaces.

In the rest of this paper, we first describe connectors from a conceptual view in Sec. 2, describe the framework in Sec. 3, an example of how the framework is used in Sec. 4, performance of the current prototype in Sec. 5, and summarise and discuss future work in Sec. 6.

2 Connectors

It has been argued by many that connectors play an important role in software architecture and even that they should have a first class representation in programming languages. Aldrich et al have implemented first class connectors in the ArchJava language [10], Dashofy et. al. argues that connectors should be present in architecture description languages [11], something which has also been argued by Medvidovic and Taylor [12], and Shaw et al argues that connectors deserve first-class status [13,14].

For our purpose, a connector is a first-class entity, which embodies communication. This naturally follows from the view offered by ArchJava. However, when a connector is bound to an ArchJava component and a web service, the ArchJava connector conceptually only embodies one role in a two role connector, since the conceptual connector is the entire communication channel. The other role is embodied in code running inside the application server. As described later, in Sec. 3, this code is also part of the framework.

3 The Framework

Central to the framework is a custom ArchJava connector, **WSConnector**, which has two primary uses: Connecting two ArchJava components where one is instantiated as a web service, and connecting an ArchJava component to an existing Web Service. When connecting two ArchJava components, one of the components will actually be run as a web service, while the other will connect to it. The framework assists in this, be supplying the necessary building blocks. This implies, that the client component can only have required methods in the port that connects to the service, while the service component can only have provided methods in its service port. However, it is possible for a service to

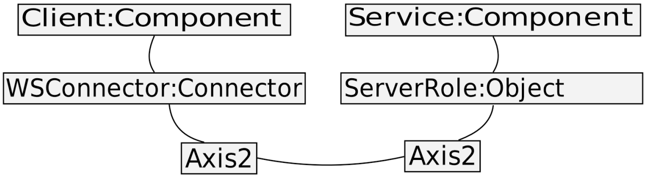


Fig. 1. Connecting a client and a service through an Archjava connector using Axis2. Axis2 instantiates the **ServerRole** object, when a message for the service is received. **ServerRole** then instantiates the actual component.

```
<service name="Echo">
<description>
        Simple Echo Service for testing purposes.
    </description>
    <parameter name="ServiceClass" locked="false">kilo.prototype.axis.
        echo.EchoService
    </parameter>

    <operation name="echo">
        <messageReceiver class="kilo.prototype.axis.connector.ServerRole
            "/>
    </operation>
</service>
```

Fig. 2. `services.xml` defining the Echo operation implemented by the class **EchoService** with **ServerRole** as message receiver

utilise other services by calling them through other ports during invocation of the service. The framework uses the Axis2 application server to run the web service [15], and the service object is only instantiated upon the reception of a message.

As mentioned earlier, the ArchJava connector is a single role in the conceptual connector. The other role is represented by the **ServerRole** class, which acts as a message receiver in an Axis2 service. The architecture of a system with one client connected to a single service is illustrated in Fig. 1. **WSConnector** uses Axis2 to call the remote web service, while the service is instantiated by Axis2, with **ServerRole** receiving the message and delegating it to the service component.

WSConnector uses external classes for marshaling and demarshaling, which allows for changing the encoding of messages.

Invocation through the connector in Axis2 is accomplished by creating a `services.xml` file which specifies **ServerRole** as the message receiver for the service, as illustrated in Fig. 2. When Axis2 receives a request for the **Echo** service, it will instantiate **ServerRole**, which can in turn instantiate the service object, the class of which is provided by the Axis2 framework.

3.1 Marshalling

To avoid reliance on wsdl-files, the ArchJava connector uses reflection over method signatures to implement marshaling and demarshaling. This is an important part, since it is what allows services to be used without the time consuming work. ArchJava provides its own reflection implementation in the **archjava.reflect** package while Java uses **java.lang.reflect**. Since ArchJava assumes the first interface, while Axis2 expects types to belong to the second interface, the connector uses **archjava.reflect** on the client side, and **java.lang.reflect** on the server side, in **ServerRole**. This gives rise to some limitations, since they have slightly different expressiveness.

The marshaling attempts to follow the same conversions as Axis2 and JAX-RPC but since documentation for those two packages are incomplete, some variations may occur. For example, the only type of array supported is arrays of **java.lang.String** and arrays of complex types. This is due to a limitation in ArchJava's support for reflection over arrays, since arrays of simple types are not supported.

3.2 Complex Types

Since the framework is intended for prototyping, using simple types is usually sufficient. However, when connecting to existing services, it might be necessary to use complex types. Luckily, wsdl files will be available for existing services, and classes implementing complex types can be generated from these files, although most generators have limitations in which types can be generated. Another possibility is to create such types by hand.

```
connect pattern EchoClient.Echo, EchoService.Echo with WSConnector {
        connect(EchoClient sender, URL service) throws IOException {
                return connect(sender.Echo, EchoService.Echo) with new
                        WSConnector(connection, service, "echo", "echo",
                             1);
        }
}

public component class EchoClient {
        ...
        public port interface Echo {
                requires connect(URL service) throws IOException;
                requires String echo(String s);
        }
        ...
}
```

Fig. 3. Using **WSConnector** to connect an **EchoClient** component and an **EchoService** component. The code defines a connect pattern, and will be called to instantiate a new **WSConnector** when **EchoClient** creates a new EchoPort. The Echo port of **EchoClient**. `connect` matches the constructor of the connector used, while the rest of the methods defines which methods the port require from the port is should be connected to.

4 Example Systems

To use custom connectors, connect patterns for the connector must be defined. As an example, a pattern for connecting the `Echo` port of an **EchoClient** component with the `Echo` port of an **EchoService** component is shown in Fig. 3. The signature of the pattern correspond to the required `connect` method of the `Echo` port, also shown in Fig. 3. The method corresponds to the constructor of components, and connections can be instantiated dynamically by using the keyword *new* on an `EchoPort`.

5 Performance

Although it is possible to use the framework for evaluating many different architecture qualities, one interesting property is its usefulness for measuring performance. This is only possible if the performance of the framework is similar to the performance of real systems.

A simple measure of performance is the total invocation time of operations. We have used a simple echo service, which just returns its string argument, and as a baseline, we use implementations for Axis1 and Axis2. The average of 1000 invocations are shown in Tab. 1.

Table 1. Average, minimal, and maximal invocation time of a call to an operation for Axis, Axis2, and **WSConnector** between a client and a remote web service

System	Average	Minimal	Maximal
Axis	57	41	438
Axis2	66	36	3157
WSConnector	112	42	1835

As can be seen from the table, there is a significant difference between minimal, maximal, and average invocation time. This is most likely partly due to limitations of the machines running the tests.

WSConnector is considerable slower than both Axis and Axis2. This is most likely due to the cost of reflection. To offset this, we plan to create a future version where each connector is especially tailored for connecting to a specific services by generating it from interfaces. This should yield a performance similar to Axis2.

6 Conclusion and Future Work

We have created a framework for building prototypes of web service based systems, where designers can concentrate on a component and connector view of the system, and then change the prototype to use web services. The main motivation is to be enable cheap and easy development of prototypes, which is paramount for enabling prototypes to be build at all. The use of reflection over interfaces

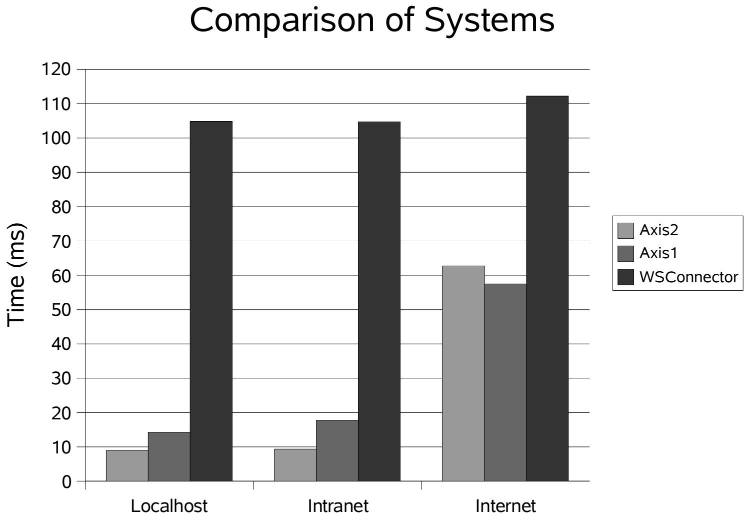


Fig. 4. Comparison of Axis1, Axis2, and WSConnector. WSConnector is somewhat slower, mostly due to two layers of reflection in each call, whereas both versions of Axis use auto generated, specialised code for marshaling.

for marshaling allows the programmer to ignore web services entirely, and then later add them to the system. We expect the framework will prove itself useful for creating elaborate prototypes of future web service based systems.

The framework will be used in a current project at the University of Aarhus, to implement initial architectural prototypes of scenarios, before we build vertical prototypes. Before building actual prototypes, the performance of the Connector should be improved to better match the performance of existing web service technologies, such as Axis2. This will most likely be done by generating the appropriate code, instead of relying on reflection at runtime. This should yield a substantial increase in performance. Also, the support for complex types in Axis2 is limited, and we are still investigating which method of generating complex types gives the best result.

Acknowledgements. The research presented in this paper has been partly funded by the ISIS Katrinebjerg project KILO and the EU project “Hydra” (IST-034891; `http://www.hydra.eu.com`).

References

1. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web services architecture. Technical report, W3C Working Group (February 2004)
2. Floyd, C.: A systematic look at prototyping. In: Budde, R., Kuhlenkamp, K., Mathiassn, L., Züllighoven, H. (eds.) Approaches to Prototyping, pp. 1–18. Springer, Heidelberg (1984)

3. Bardram, J.E., Christensen, H.B., Hansen, K.M.: Achitectural prototyping: An approach for grounding architectural design and learning. In: WICSA'04. Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (2004)
4. Bardram, J.E., Christensen, H.B., Corry, A.V., Hansen, K.M., Ingstrup, M.: Exploring quality attribues using architectural prototyping. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, Springer, Heidelberg (2005)
5. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Professional, Reading (2003)
6. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE '02. Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM Press, New York (2002)
7. Grundy, J., Cai, Y., Liu, A.: Softarch/mte: Generating distributed system testbeds from high-level software architecture descriptions. Automated Software Engg. 12(1), 5–39 (2005)
8. Yamaizumi, T., Sakairi, T., Wakao, M., Shinomi, H., Adams, S.: Easy soa: Rapid prototyping environment withweb services for end users. In: ICWS '06. Proceedings of the IEEE International Conference on Web Services, pp. 931–932. IEEE Computer Society, Washington, DC, USA (2006)
9. Zinnikus, I., Elguezabal, G.B., Elvesæter, B., Fischer, K., Vayssière, J.: A model driven approach to agent-based service-oriented architectures. In: Fischer, K., Timm, I.J., André, E., Zhong, N. (eds.) MATES 2006. LNCS (LNAI), vol. 4196, pp. 110–122. Springer, Heidelberg (2006)
10. Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language Support for Connector Abstractions. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
11. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engeneering Methodology 14(2), 199–245 (2005)
12. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engeneering 26(1), 70–93 (2000)
13. Shaw, M.: Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In: Proceeding of Workshop on Studies of Software Design (January 1994)
14. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering 21(4), 314–335 (1995)
15. Apache Software Foundation: Axis2/Java - Apache Axis2/Java - Next Generation Web Services (accessed, June 2007), http://ws.apache.org/axis2/

Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach*

Fernando Losilla, Cristina Vicente-Chicote, Bárbara Álvarez, Andrés Iborra,
and Pedro Sánchez

División de Sistemas e Ingeniería Electrónica (DSIE)
Universidad Politécnica de Cartagena, 30202 Cartagena, Spain
{Fernando.Losilla,Cristina.Vicente,balvarez,Andres.Iborra,
Pedro.Sanchez}@upct.es

Abstract. Nowadays, Wireless Sensor Networks (WSN) are a very promising research field since they find application in many different areas. Current proposals for WSN system development are mainly focused on implementation issues and they rarely rely on a Software Engineering methodology which supports their entire development life-cycle. The Model-Driven Engineering (MDE) approach can contribute to solve this problem by allowing designers to model their systems at different abstraction levels, providing them with automatic model transformations to incrementally refine abstract models into more concrete ones. In this vein, this paper presents a MDE approach to WSN application development. Three levels of abstraction have been defined which allow designers to build: (1) domain-specific models, (2) component-based architecture descriptions, and (3) platform-specific models. Automatic model transformations between these three abstraction levels have been designed and, in order to demonstrate the viability of the proposal, a real WSN application has been developed using the implemented tools.

Keywords: Model-driven engineering, component-based software architecture, domain specific languages, wireless sensor networks, Eclipse platform.

1 Introduction

Recent technological advances have led to the emergence of Wireless Sensor Networks (WSN). These systems are able to observe the physical world and to obtain useful information from it. They can process the retrieved data, make decisions on it, and carry out concrete operations on the environment [1]. Nowadays, Wireless Sensor Networks find application in many different domains, such as: environmental monitoring, tele-medicine, or precision agriculture, among others [2]. In 2003 the MIT's Technology Review [3] published a study where WSN applications were cited as "*one of the top ten technologies that will change the world*". However, current

* This research has been funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02) and the Regional Government of Murcia Seneca Program (02998-PI-05).

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 179 – 194, 2007.

techniques for implementing this kind of systems seem to be not powerful enough to deal with their growing complexity.

Current proposals for WSN application development are mainly focused on implementation issues. Actually, most of these systems are built from scratch following an experience-based method, which advocates for selecting the most appropriate target platform first, and then the WSN domain-specific operating system (e.g. TinyOS [4]) and programming language (e.g. NesC [5]). The lack of a Software Engineering methodology which supports the entire development life-cycle of these applications, commonly results in highly platform-dependent designs, difficult to maintain, scale and reuse.

The Model-Driven Engineering (MDE) approach can help reducing this dependence of the software development process on the final execution platforms [6]. MDE revolves around models (defined at different levels of abstraction), and automatic model transformations, aimed at incrementally refining models into final application code. Models are defined in terms of formal meta-models (or modelling languages). These include the concepts needed to describe a system (or a set of systems) at a certain level of abstraction, and the relationships existing between them. In order to describe the model transformations, that is, how abstract models are refined into more concrete ones, a mapping between their corresponding meta-models must be defined. Thus, applying a MDE approach requires defining both, the appropriate meta-models and the corresponding model transformations.

This paper presents a MDE approach to WSN application development aimed at improving the flexibility and reusability of their designs. Three meta-models have been defined at different levels of abstraction together with the corresponding model transformations. Designers model their systems using only the WSN domain concepts included in the highest level meta-model. These initial models are then successively refined through model transformations until the final application code is automatically generated.

Before entering into details, the following section presents a motivation example based on a real WSN application for *precision agriculture*, which highlights the lack of flexibility and reusability of current WSN application designs. This is followed by an outline of the research goals and process. Then, the rest of the paper is organized as follows. First, Section 2 briefly presents the platform selected to implement the tools developed as part of this work. Then, the different meta-models and model transformations implemented as part of the proposal are presented in sections 3 to 6. Section 7 reviews some related works and, finally, Section 8 presents the conclusions and some future research lines.

1.1 A Motivation Example

The MITRA WSN application consists of thirty TinyOS-based nodes deployed in an almond orchard located in the semiarid region of Murcia, in the southeast of Spain. Given the of shortage water in this region, the prime objective of the system is to regulate tree irrigation according to water stress, that is, to water the trees only when it is needed. Water stress is measured using the heat pulse compensation method. This method consists in generating a heat pulse through the axial line of the tree trunk and measuring the sap temperature at two different points along this line. Similar temperatures indicate a fast sap flow and this suggests that some watering is needed.

The MITRA application was initially developed using a traditional approach. A new small electronic sap flow sensor was designed and the software to control both, local data processing and wireless communications was implemented in NesC, a component-based programming language for TinyOS-based WSN applications.

The resulting system was highly dependent on the TinyOS-NesC platform and on the custom sap sensor. The lack of flexibility of the design required several changes, both in hardware and in software, to cope with every small change in the application. This led us to search for a more flexible design solution, as described in the following subsection.

1.2 Research Goals and Progress

As stated before, the main goal of this research is to define a new model-driven methodology for WSN application development which allows developers to build more flexible and reusable designs. This goal was initially address considering the following sub-goals:

SG1. Define a WSN Domain-Specific Language (DSL) which enables the description of this kind of applications at a very high level of abstraction. Models built using this WSN DSL should pick up all functional and non-functional system requirements, but they should not include any design or implementation decisions. For this DSL to be really useful, a model editor (preferably a graphical one) must be implemented to support new Domain Model (DM) creation and validation.

SG2. Define the NesC [5] meta-model from the current language specification. A graphical modelling tool for creating new NesC component-based models would be desirable, but not required since NesC models will be automatically generated from WSN DSL ones.

SG3. Define a model-to-model (M2M) transformation which maps the concepts included in the WSN DSL to those included in the NesC meta-model.

SG4. Define a model-to-text (M2T) transformation which automatically generates NesC code from NesC models.

SG1 and SG2 were addressed in parallel by different teams. When these targets were completed (fully implemented and tested), part of the team working on the NesC meta-model started working on the M2T transformation (SG4), while the rest of the people involved in the research started working on the M2M transformation (SG3).

While the M2T transformation was successfully completed quite fast, the transformation between the WSN DSL and the NesC meta-model seemed a really hard problem to solve. This was mainly due to the huge semantic distance existing between the concepts included in both meta-models. At this point, two options were considered: (1) to introduce some lower abstraction concepts regarding system design into the DSL, or (2) to define an additional abstraction level between the two meta-models.

Option (1) meant reducing the abstraction level of the DSL and forcing developers to include design decision within their models. In contrast, option (2) offered evident advantages and just a few drawbacks. On the one hand, the use of an intermediate meta-model could help bridging the gap between WSN domain concepts and NesC primitives. As a consequence, and despite the need of defining two M2M transformations instead of one, the complexity of these transformations would be significantly lower. On the other hand, this intermediate meta-model could serve as an appropriate architecture description language for defining the system in terms of its components and the relationships existing between them.

With this aim in mind, a subset of the UML 2.0 [7] meta-model, including components (for describing the system structure) and state-machine and activity diagrams (for describing component behaviour), was defined to mediate between the WSN DSL and the NesC meta-model.

Fig. 1 (*left*) outlines the elements of the proposal, that is, the set of meta-models and model-transformations defined to obtain NesC code from WSN Domain Models (DM). The intermediate architecture description language has been highlighted in order to emphasize its key role in the process.

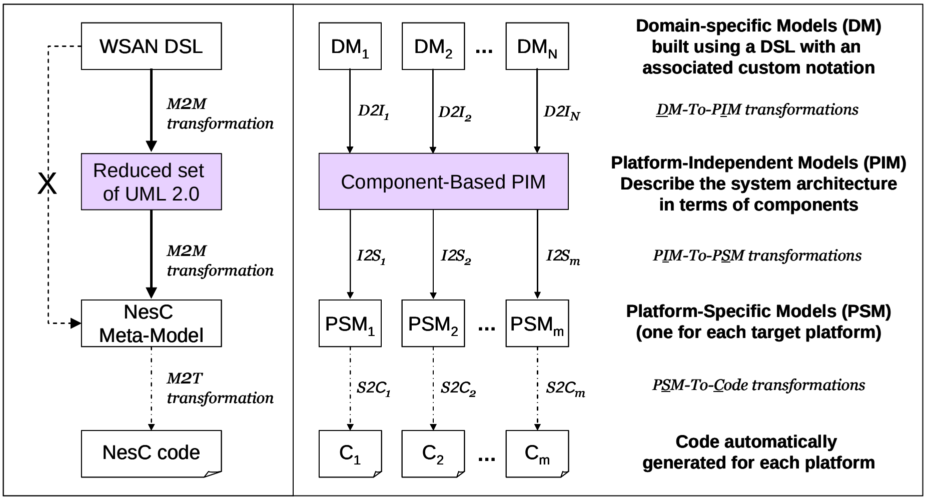

WSAN DSL
M2M transformation
Reduced set of UML 2.0
X
M2M transformation
NesC Meta-Model
M2T transformation
NesC code
DM1
DM2
...
DMN
D2I1
D2I2
D2IN
Component-Based PIM
I2S1
I2S2
I2Sm
PSM1
PSM2
...
PSMm
S2C1
S2C2
S2Cm
C1
C2
...
Cm
Domain-specific Models (DM) built using a DSL with an associated custom notation
DM-To-PIM transformations
Platform-Independent Models (PIM) Describe the system architecture in terms of components
PIM-To-PSM transformations
Platform-Specific Models (PSM) (one for each target platform)
PSM-To-Code transformations
Code automatically generated for each platform

Fig. 1. (*Left*) Meta-models and model transformations included in the proposed MDE approach to WSN development. (*Right*) Models defined using the intermediate meta-model and the meta-model itself can be fully reused if new PSMs/DSLs were addressed in the future.

In addition to the already mentioned benefits of including an intermediate abstraction level, it is worth noting that models defined at this level can be fully reused if new target platforms were considered in the future. Furthermore, if new domains were addressed, the meta-model itself could be reused since it has been designed to be not only platform-independent but also domain-independent. This idea is illustrated in Fig. 1 (*right*).

2 The Eclipse Platform: The Selected MDE Environment

All the tools implemented as part of this work, including all the meta-models and model transformations outlined in the previous section, have been developed using the MDE facilities provided by the Eclipse platform. This free open-source environment offers one of the most widely used implementation of the OMG standard Meta-Object Facility (MOF) [8], called Eclipse Modelling Framework (EMF) [9].

Although EMF currently supports only a subset of MOF, called EMOF (Essential MOF), it allows designers to create, manipulate and store both models and meta-models. This is the reason why many MDE-related initiatives are currently being developed around Eclipse and EMF. Among them, and directly related to the tools implemented as part of this research, it is worth mentioning the following ones:

- The Graphical Modelling Framework (GMF) [10], which enables the implementtation of graphical modelling tools from any EMF meta-model.
- The Eclipse Modelling Framework Technologies (EMFT) [11] which enables, among other things, the definition and evaluation of OCL queries and constraints on EMF models.
- The Atlas Transformation Language (ATL) [12], which provides the standard Eclipse solution for model-to-model transformations.
- MOFScript [13], which supports text (and more specifically code) generation from MOF-based models.

All these Eclipse plug-ins will be briefly detailed in the sections where the tools implemented as part of this work are presented.

3 Defining a WSN Domain-Specific Modelling Language

The definition of a WSN domain-specific modelling language (meta-model) is aimed at helping domain experts to describe their systems using only the WSN concepts they are familiar with. At this initial stage, no design decisions or concerns about the final target platform must be taken into account. Conversely, models at this level must provide a clear picture of system defined at a high level of abstraction. For instance, a WSN domain-specific model should supply information about the overall system functionality, how this functionality is partitioned into the different nodes, how this nodes are grouped and physically distributed, which information do they get from the environment, how do they communicate with each other and how frequently, where is the data processed/stored (locally or remotely), how it is presented to the user, etc.

The WSN Domain-Specific Language (DSL) presented in this paper has been designed to help domain experts to include all this information in their models. Thus, it provides the concepts most commonly used by the WSN community, together with the relationships that may appear between these concepts (see Fig. 2).

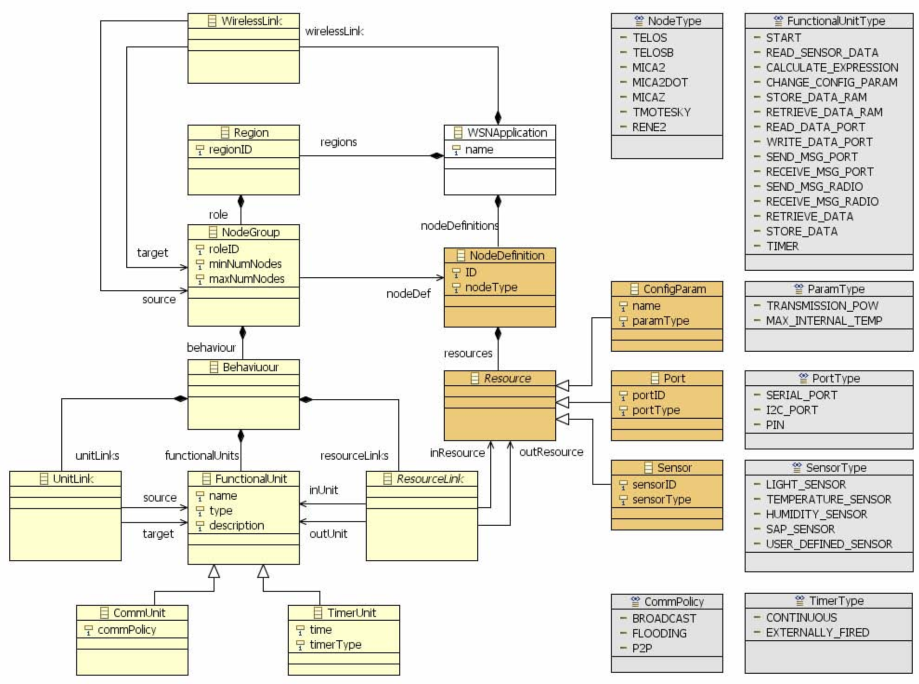


Fig. 2. WSN Domain Specific Language (WSN DSL)

Both structural and behavioural elements have been included in the meta-model. The structure of a WSN application is defined in terms of `Regions` connected by means of `WirelessLinks`. All the nodes performing similar tasks are grouped into a logical `NodeGroup`, while all the `NodeGroups` physically deployed together are considered to belong to the same `Region`.

The common `Behaviour` of the nodes belonging to the same `NodeGroup` is defined in terms of `FunctionalUnits`. The meta-model includes an enumerated set of predefined functional units (`FunctionalUnitType`) which contains, among others, data management (read/write from/to sensors/ports/memory), expressions calculation, timers, etc. `FunctionalUnits` can be linked together by means of `UnitLinks` and with external `Resources` (i.e. `Ports` and `Sensors`) by means of `ResourceLinks`. All these links together define the data-flow behaviour of the `NodeGroup`. Timers model explicitly internal node control-flow behaviour, while the external synchronization mechanisms regarding inter-node message passing is only implicitly represented in the models.

All the concepts and relationships included in this meta-model are quite useless if no tool is provided to support the creation and validation of new models from it. The following section presents the graphical notation and the modelling facility implemented on top of the WSN DSL previously described.

3.1 The WSN Graphical Modelling Tool

As previously stated, all the tools implemented as part of this work have been developed using the MDE facilities provided by the Eclipse Platform. In particular, the WSN DSL has been defined as an EMF [9] meta-model, and the graphical modelling tool, implemented to help domain experts to create new WSN models, has been developed using the GMF [10] Eclipse plug-in.

GMF allows designers: (1) to create a graphical representation for each domain concept appearing in a EMF meta-model, (2) to define a tool palette for creating and adding these graphical concepts to their models, and (3) to define a mapping between all the previous artefacts, i.e. meta-model concepts, their graphical representations, and the corresponding creation tools.

In addition, GMF can be used in conjunction with the EMFT [11] Eclipse plug-in to define new restrictions which can not be included in the meta-model (given the limitations of using class diagrams). These restrictions are defined as OCL constraints and they are validated at modelling time using the EMFT plug-in facilities.

The MITRA system, previously described in the introduction, has been depicted using the implemented WSN graphical modelling tool. As shown in Fig. 3, the model includes two regions. The first one contains two NodeGroups, one representing the MITRA nodes deployed in the almond orchard (SAP Monitoring NodeGroup) and another representing the Irrigation Control NodeGroup (containing only one node). The second region contains only one Sink NodeGroup with a single node. The behaviour of each of these NodeGroups has been defined following this three-step process:

1. Select the sensors to be read from those available in the selected NodeDefinition.
2. Select the activities performed by the NodeGroup from the enumerated set included in the WSN DSL, and
3. Link all these elements together to fulfil the system requirements, respecting the syntactic rules defined by the meta-model and the additional OCL rules included in the GMF application.

The following section describes how these WSN graphical models are automatically transformed into UML-based Platform-Independent Models (PIMs). Both the reduced set of the UML 2.0 meta-model, selected as the intermediate architecture description language, and the model-to-model transformation used to refine domain-specific models to the PIM level, are presented.

4 From Domain-Specific Models to UML-Based PIMs

As previously discussed in the introduction, we have chosen a simplified version of the standard UML 2.0 [7] meta-model as the intermediate specification language. This intermediate abstraction level is aimed at bridging the semantic gap existing between the domain and the platform meta-models, reducing the complexity of the required model transformations.

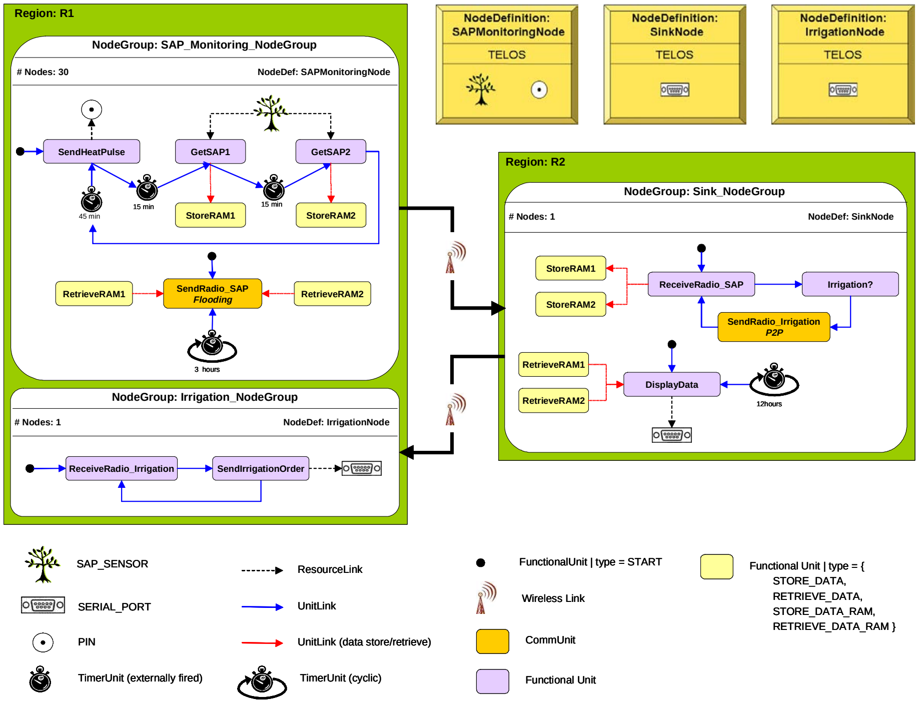


Fig. 3. MITRA system model depicted using the WSN DSL graphical modelling tool

The intermediate meta-model has been defined taking some of the elements included in the UML 2.0 meta-model and, more specifically, some of the concepts defined within the component, state-machine, and activity diagrams. Components are used to specify the system structure, while state-machines and activity diagrams are used to define component control-flow and data-flow behaviour, respectively.

The part of meta-model related to component specification includes simple and complex components, ports, port links, interfaces and services. The state-machine part includes states, pseudo-states (initial, join, and fork), orthogonal regions, transitions and events, and the activity diagram part, includes simple and complex activities (including conditionals and loops), timers, and activity links.

This meta-model has also been developed using EMF although, in this case, implementing a graphical modelling tool was unnecessary since (1) models at this level are not defined by the user but automatically obtained from DSL models and, (2) EMF provides a basic reflexive model editor which is sufficient to check the correctness of these intermediate models and which allows designers to manually introduce slight variations into them to test different architectural configurations.

Regarding the Model-to-Model (M2M) transformation required to refine DSL models into component-based PIMs, it has been implemented using the Eclipse Atlas Transformation Language (ATL) [12]. This hybrid declarative and imperative language enables to define mappings between the concepts included in different meta-models, that is, how each concept (or set of related concepts) in the source meta-model can be transformed into a concept (or set of related concepts) in the target meta-model. One-to-one transformations are desirable but commonly they are only possible when the concepts included in the two meta-models are semantically close.

In this case, the transformation from the WSN DSL and the intermediate UML-based meta-model requires some relatively complex mappings, although some of them are also quite direct. Some of the rules, included in this ATL model transformation, are outlined next:

- Each `NodeGroups` is mapped to a `Component`.
- Given the data-flow oriented behaviour of WSN `NodeGroups`, a very simple `StateMachine` is associated to each `Component` including only an `Initial PseudoState`, and two `States`: *Working* and `Final`.
- Uncoupled sets of `ActivityUnits` are placed into different `OrthogonalRegions` in the *Working* `State`, since these activities are executed in parallel. This is the most complex transformation rule since it requires detecting unconnected graphs of activities.
- Each `Timer FunctionalUnit` is mapped to a UML `TimerActivity`.
- All the messages sent via wireless from a `NodeGroup` to another are modelled as UML `SendSignalActions`.
- Signals send by each NodeGroup to an output resource (`Ports`) are converted into UML `SendSignalActions` and, conversely, signals received from input resources (`Sensors`) are transformed into UML `AcceptEventActions`.
- Each `AcceptEventAction` requires adding a new `Event` to the `StateMachine` and an `InternalTransition` in the *Working* `State` fired by this event.

The result of applying the ATL transformation on the initial MITRA DSL model is a UML-like platform-independent component model, which describes the system structure and behaviour in terms of its components and the relationships existing between them. Although, as stated before, we have not implemented a graphical model editor for this intermediate level, the following figure (created using a basic

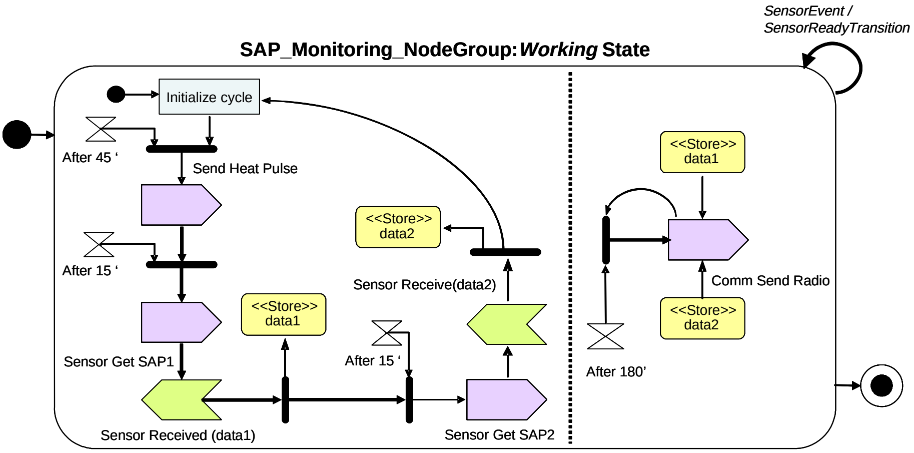


Fig. 4. Behaviour of the SAP monitoring `Component`

drawing tool) has been depicted using the UML graphical notation in order to provide an easier to understand representation of part of the resulting model, and more precisely, the behaviour of the SAP Monitoring `Component` (`NodeGroup`).

As it can be readily appreciated in Fig. 4, the transformation has divided the *Working* `State` of the SAP Monitoring `Component` into two `OrthogonalRegions`. These regions contain the two uncoupled sets of activities identified in the `NodeGroup` (see Fig. 3), one describing the sensing loop and another describing how the collected data is sent to the Sink `NodeGroup` via wireless. Similarly, the *Working* `State` of the Sink `Component` is also divided into two `OrthogonalRegions`, one including the activities related to data collection (from SAP monitoring nodes), irrigation need estimation, and control message delivery (to the Irrigation Control NodeGroup), and another including data retrieval and display activities.

The following section describes how these intermediate models are automatically transformed into TinyOS-NesC Platform-Specific Models (PSMs), applying a new ATL model-to-model transformation.

5 From PIMs to NesC Component Models

Nowadays, TinyOS [4] is the most widely used operating system for WSN application development and, accordingly, a wide variety of tools supporting it can be currently found in the marketplace. Among them, the solution developed by the TinyOS team is the NesC [5] component-based programming language, also extensively used.

A NesC meta-model has been implemented, according to the NesC 1.1 specification, to support the last stage of the proposed MDE approach. This meta-model comprises the following concepts: `Modules` (which define component implementation), `Configurations` (which define component groups and `Wirings` between them), and `Interfaces` (which include a list of `CommandPrototypes` and `EventPrototypes`). Modules must implement all the `Commands` included in the interfaces they provide and all the `Events` in the interfaces they use.

NesC applications are designed following a quite regular component pattern and thus, the rules needed to define the ATL transformation between PIMs and NesC models are not very complex. Some of these rules are outlined next:

- A different NesC model is created for each `Component` in the PIM.
- All `TimerActivities` defined in the PIM are implemented by a unique predefined NesC module, called *TimerC*. This module provides a parameterized *Timer* interface which includes a *fired* event.
- Conversely, all `SendSignalActions` are implemented by a unique predefined *GenericComm* module which provides different interfaces for each type of message. All these interfaces include a *SendMsg* event.
- A Main module must be necessarily included in the design since it is invoked when the application starts. This module is in charge of initializing the rest of modules.
- A top level `Configuration` module must be defined to wire all the previously defined components.

Fig. 5 presents the NesC model associated to the sap monitoring component defined in the PIM. Like in the previous case, this graphical representation has been manually depicted using a conventional drawing tool, since a graphical NesC model editor has not been implemented yet. However, the depicted model faithfully represents the model obtained as the result of applying the implemented ATL transformation.

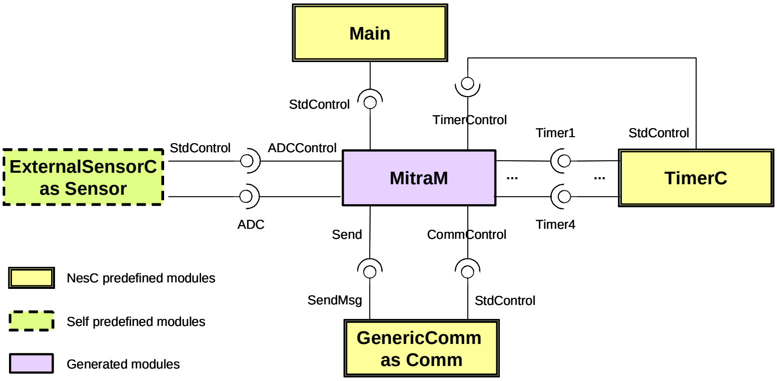


Fig. 5. NesC component model for the sap monitoring nodes

NesC models, described according to the implemented NesC meta-model, only enable the description of components and their interconnections, and do not include any of the behavioural information described in the more abstract models. As explained in the following section, this requires using NesC models and PIMs in order to generate the final application code. Some improvements in the NesC meta-model and in the ATL transformation from PIMs to NesC models are currently being developed to address this limitation, as later commented in the conclusions.

6 Code Generation

The final step of the proposed MDE methodology for WSN system development is to obtain the final application code from the previous models. In this case, the transformation is not Model-to-Model (M2M) but Model-to-Text (M2T). M2T transformations define how model elements are converted into text patterns (in this particular case, into code), while M2M transformations define meta-model mappings, that is, how the elements (concepts and relationships between them) included in one of the meta-models are transformed into elements included in the other.

The final NesC M2T transformation has been implemented using the Eclipse MOFScript [13] plug-in. This tool enables the definition of M2T transformations for MOF-based models. It provides, among others, the following facilities: model checking, parsing and querying, output file management, code completion, etc.

In order to obtain the final application code, the NesC models obtained in the previous step are not sufficient, since they do not contain complete information about component behaviour. Thus, the MOFScript transformation has been designed to use both, NesC models and intermediate PIM models (containing component behaviour specifications obtained from the initial DSL models). This limitation is currently being addressed as described in the conclusions and future research section.

Some of the rules included in the MOFScript transformation, are outlined next:

- The NesC component model enables the generation of a list of provided and required interfaces (`provides` and `uses` clauses, respectively) included at the beginning of the NesC file.
- The module implementation starts with a variable declaration section. One variable is defined for each <<`Store`>> `Activity` defined in the PIM with a different name, and also for each message the `Component` sends (receives) to (from) other components via wireless.
- For each provided `Interface` in the NesC model, all its `Commands` must be implemented. Conversely, for each required `Interface`, all its `Events` must be handled. The sequence of NesC primitives associated to these `Commands` and `Events` is extracted from the `Activities` defined in the PIM.

An excerpt of the implemented MOFScript transformation can be found in Appendix A, and a piece of the generated NesC code in Appendix B, both of them included at the end of the paper. The generated solution, obtained from the corresponding PIM intermediate model (see Fig. 4) and the NesC component model (see Fig. 5), has been successfully compiled and deployed on the MITRA WSN system, demonstrating satisfactory results.

7 Related Work

Different approaches to WSN application development can be found in the literature. Some of them offer a set of predefined components, built on top of a certain operative system, and allow designers to build new ones by configuring and combining them. This is the case of TinyDB [14], which defines a database of TinyOS [4] components that can be distributed and run in different nodes of a WSN. The information obtained by each node can be accessed from an external PC by means of SQL-like queries. Also in this line, TinyCubus [15] offers a framework which enables to dynamically select and interconnect predefined TinyOS components. These two proposals, as most others, are totally focused on the implementation of platform-specific WSN applications. These approaches require a deep knowledge of the target platform and, in most cases, the resulting designs are too platform-dependent to be reused.

Trying to address this problem, and in the line of this paper, some proposals have focused their attention on the model-driven approach. GRATIS II [16] offers a graphical modelling tool for designing component-based NesC applications. The underlying NesC meta-model has been defined using the Generic Modelling Environment (GME) [17], a toolkit for defining domain-specific modelling and programming languages. GRATIS II offers a solution similar to the one offered by the final step of our proposal, that is, the one including our NesC meta-model and NesC model-to-code transformation. Currently we do not offer a graphical modelling tool for depicting NesC component models since these are automatically generated from the higher level meta-model, and not depicted by the user like in GRATIS II. Our

NesC meta-model is also simpler than the one offered by GRATIS II, which covers a wider range of TinyOS-based target platforms and configuration modes. However, our proposal is not TinyOS (or any other platform) dependent, enabling higher level WSN application designs. Currently we support TinyOS-NesC code generation like GRATIS II, although the proposal could be easily extended to different target platforms, as stated in the introduction (see Fig. 1).

The Abstract Task Graph (ATaG) [18] offers a DSL for graphically describing WSN applications in terms of the tasks they must perform and the data their nodes must collect. The data-driven diagrams depicted using this DSL provide a platform-independent model of the system under development (nodes, tasks, data types, etc. are abstract to keep this platform independence). These models are similar to the activity diagrams included in our intermediate component-based architectural models. However, ATaG is architecture-independent and thus, no design information can be included within its models. Furthermore, the ATaG code generation requires the user to provide the code of each abstract task included in the model for the current target platform, offering only a semi-automated solution.

Finally, CADENA [19] offers a very complete and sophisticated environment for general purpose application development. CADENA provides designers with an end-to-end model-driven environment which supports the entire application development life cycle. This tool offers, among others, a NesC plug-in which provides a graphical modelling tool (similar to the one offered by GRATIS II), and automatic NesC code generation facilities. WSN applications can also be modelled at a higher level of abstraction using CADENA general-purpose artefacts “adapted” to the WSN domain. However, adapting a general-purpose language or tool to a specific domain can be a hard work and the result could be never as good (in terms of precision, efficiency, etc.) as the one obtained by defining a DSL.

8 Conclusions and Future Research

The work presented in this paper offers a new model-driven approach to WSN application development. The proposal presents a high level of abstraction domain-specific language, which allows designers to model their systems in a platform-independent way, obtaining more flexible and reusable designs. Two additional abstraction levels have been defined which deal with the system architecture from a platform-independent and platform-specific point of view. Automatic model transformations from the initial domain-specific models to the final application code have been implemented using the Model-Driven Engineering facilities provided by the free open-source Eclipse platform. Both the proposed approach and the tools implemented to support it have been tested on a real WSN system related to *precision agriculture* with successful results. The re-engineered application is fully functional and, although it is not code-optimized, the effort and the time-to-market to produce the new solution have been sensibly reduced. Actually, different solutions have been effortlessly generated thanks to the implemented infrastructure, allowing us to test new sensors and different network topologies and communication protocols.

Currently, we are working on improving the NesC meta-model and the ATL transformation from PIMs to NesC models in order to keep all the behavioural information during the whole development process. This will allow us to simplify the final code generation step using only NesC models as input. We are also working on the integration of requirements from the very early stages of the proposed MDE approach. Actually, we have developed a Requirements Engineering Meta-Model (REMM) and a graphical requirements modelling tool aimed at defining reusable requirements catalogues [20]. Currently we are building a catalogue of functional and non-functional WSN requirements together with a tracing tool. We are also very interested in proving the benefits of our intermediate meta-model for different target platforms and domain applications. Thus, we plan to define new DSLs for other domains in which our research team has also some experience, such as computer vision and robotics. The wide variety of platforms currently available for this kind of systems, and the fact that some applications incorporate concepts from both domains (i.e. industrial inspection), make this future research both challenging and promising.

References

1. Akyildiz, I.F., Kasimoglu, I.H.: Wireless Sensor and Actor Networks: research challenges. Ad Hoc Networks 2, 351–367 (2004)
2. Römer, K., Mattern, F.: The design space of wireless sensor networks. IEEE Wireless Communications 11, 54–61 (2004)
3. Huang, G.T.: Casting the Wireless Sensor Net. MIT’s Magazine of Innovation, 51–56 (2003)
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System Architecture Directions for Networked Sensors, vol. 34. ACM Press, New York (2000)
5. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Network Embedded Systems. In: ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, California, USA, pp. 1–11 (2003)
6. Kent, S.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, Springer, Heidelberg (2002)
7. Unified Modeling Language: Superstructure v 2.0. The Object Management Group (2005)
8. Meta-Object Facility Specification v2.0: The Object Management Group (2004)
9. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modelling Framework. Addison-Wesley Professional, Reading (2003)
10. The Eclipse Graphical Modelling Framework, available at: http://www.eclipse.org/gmf
11. The Eclipse Modedelling Framework Technologies (EMFT) Projects, available at: http://www.eclipse.org/emft/projects/
12. The Atlas Transformation Language (ATL) Project, available at: http://www.eclipse.org/m2m/atl/
13. The Eclipse MOFScript subproject, available at: http://www.eclipse.org/gmt/mofscript/
14. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An Acqusitional Query Processing System for Sensor Networks. ACM Transactions on Database Systems 30, 122–173 (2005)
15. Marrón, P.J., Minder, D., Lachenmann, A., Rothermel, K.: TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks. Information Technology 47, 87–97 (2005)
16. GRATIS II: Institute for Software Integrated Systems. Vanderbilt University, Tennessee, USA, available at: http://www.isis.vanderbilt.edu/projects/nest/gratis

17. Lédeczi, Á., Maróti, M., Völgyesi, P.: The Generic Modeling Environment (GME). Institute for Software Integrated Systems, Vanderbilt University, Tennessee, USA, available at: http://www.isis.vanderbilt.edu/Projects/gme
18. Bakshi, A., Prasanna, V.K., Reich, J., Larner, D.: The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems. In: EESR'05. Proc. Workshop on End-to-End, Sense-and-Respond systems, applications and services, Seattle, Washington, pp. 19–24 (2005)
19. The Cadena 2.0 Project: Kansas State University, USA, available at: http://cadena.projects.cis.ksu.edu/
20. Vicente-Chicote, C., Moros, B., Toval, A.: REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting. Journal of Object Technology, Special Issue TOOLS EUROPE 2007 (2007)

Appendix A: Excerpt of the MOFScript M2T Transformation

This excerpt of the implemented MOFScript transformation is the module in charge of generating the code for all the NesC interfaces defined in NesC models.

```
texttransformation NesCM2T (in mm:"NesC") {
...
module::createInterfaces(){
   self.objectsOfType(mm.Interface)->forEach(i:mm.Interface){
      file f ( i.name + ".nc" )
      f.println ( "interface " + i.name + "{" )
      var count:integer
      // Adds the interface command prototypes
      i.prototypes->forEach( p:mm.CommandPrototype ){
         if ( p.isAsynchronous==true ) f.print ("async command ")
         else f.print ( "command " )
         f.print ( p.returnType + " " + p.name + "(" )
         count = p.arguments.size()
         p.arguments->forEach ( v:mm.Variable) {
            f.print ( v.type + " " + v.name )
            if ( count > 1 ) {f.print (", ") count=count-1}
         } f.println ( ");" )
      }  // Event prototypes are similarly added …
}}
```

Appendix B: Excerpt of the Generated NesC Code

This excerpt of the generated NesC code corresponds to the MitraM module defined in the NesC component model (see Fig. 5).

```
includes MitraMsg;
module MitraM {
   uses {
      interface StdControl as CommControl;
      interface StdControl as TimerControl;
      interface StdControl as ADCControl;
      interface SendMsg as Send;
      interface ADC;
      interface Timer as Timer1;
      interface Timer as Timer2;
      interface Timer as Timer3;
      interface Timer as Timer4; }
   provides {
      interface StdControl; }
}
implementation {
   TOS_Msg msg_global;
   uint8_t counter=1;
   uint16_t reading1, reading2;
   command result_t StdControl.start() {
      call ADCControl.start();
      call TimerControl.start();
      call CommControl.start();
      call Timer4.start(TIMER_REPEAT, 1024*60*60*3);
      initialize_cycle();
      return SUCCESS; }
   …
   event result_t Timer1.fired() {
      SENSOR_SEND_PULSE();
      call Timer2.start(TIMER_ONE_SHOT, 1024*60*15);
      return SUCCESS;}
   …
   event result_t Timer4.fired(){
      // Builds a message containing sap measures and sends it to the sink node via wireless.
      struct MitraMsg *message =
                     (struct MitraMsg *)msg_global.data;
      message->RAM1 = reading1;
      message->RAM2 = reading2;
      call Send.send(TOS_BCAST_ADDR,
                     sizeof(struct MitraMsg),&msg_global));
      return SUCCESS;}
}
```

A Distributed Staged Architecture for Multimodal Applications*

Alessandro Costa Pereira[1,2], Falk Hartmann[1,2], and Kay Kadner[1,2]

[1] Department of Computer Science
Technische Universität Dresden
Dresden, Germany
{alessandro.pereira,falk.hartmann,kay.kadner}@tu-dresden.de
[2] SAP Research CEC Dresden
SAP AG
Dresden, Germany
{alessandro.costa.pereira,falk.hartmann,kay.kadner}@sap.com

Abstract. Most of the research in the area of multimodality discusses either the usability aspect of multimodality or the multimodality support given by or missing in certain markup languages. The overall architectural side of large multimodal systems is unfortunately not adequately represented in today's literature. This report shows some results obtained during the implementation of such a system, e.g., the use of the multimodal interaction framework and how a staged architecture can be combined with this framework to achieve domain independence.

1 Introduction

The *Services for Nomadic Workers* project (SNOW, [2]) aims at enabling the widespread use of multimodal documentation for mobile operations in an industrial environment. One of the use cases covered by the project is that of a service worker in the aircraft maintenance domain. In order to get rid of the currently used paper-based maintenance documentation, the worker should get electronic access to this information. The documentation consists of so-called *procedures*. Because of the requirements (hands-free operation) and characteristics (noisy, changing light conditions, restricted) of the working environment, access to procedures must be multimodal, e.g., by allowing the use of speech for navigation.

In Section 2, the requirements within the SNOW project and the overall architecture are explained. As there are (from the archictural point of view) two major requirements, i.e., support for multimodal access and the demand for domain independence, the article has two main parts. These requirements and their consequences for the architecture are explained in more detail in Section 3 and 4. Lastly, related work is discussed in Section 5 and conclusions together with an outlook are given in Section 6.

* An extended abstract of this paper has been published at the SE 2007 Conference on Software Engineering [1].

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 195–206, 2007.

2 Architectural Requirements

As already mentioned, the first major requirement of the SNOW project was that the resulting software had to be accessible in a multimodal fashion. In the standard use case this includes speech input and output as well as gesture recognition as input, but beyond that, the architecture should not restrict the number or type of usable modalities.

The second major requirement was to design an architecture that is as domain-neutral as possible, i.e., the number of parts to be exchanged when switching to another domain had to be minimized. A second domain that has been considered during the design of the SNOW architecture was the area of healthcare, where hands-free operation also plays an important role.

In addition to these major requirements, some minor issues had to be considered. First, the number of available devices that are usable in a harsh environment and capable of delivering input for gesture recognition (via built-in or extra camera) were limited. Moreover, the processing power of available devices is restricted, forcing gesture and voice recognition components to be located on a server with extensive processing capabilities.

Lastly, it was required that the documentation is always *at least as good as paper*, which means that even with interruptions of the network connection, the application's user must have access to (prefetched) procedures. The missing network connection might thereby affect accessibility of the application by restricting the use of modalities due to their server-based processing.

The SNOW project was finished in September 2006. The architecture shown in Figure 1 has proven to be quite stable and mature. Some details, such as components dealing with prefetching of procedures have been omitted for better comprehensibility. The notation used for Figure 1 is a block diagram as defined by the F-M-C (Fundamental Modeling Concepts, see [6], [7]). This kind of diagram gives a concise and easy to understand overview of architectural structures.

Nigay et al. defined the CARE properties for assessing capabilities of multimodal systems with regard to the multimodal interaction [8], which are complementarity, assignment, redundancy and equivalence. An extended version aligns the CARE properties to devices, users and languages [9]. The extended version is only partially applicable to the SNOW architecture, since we do not have explicit interaction languages. Because modalities are tightly bound to devices and our approach was system-centric, applying the CARE properties only makes sense on a modality level as it was proposed in [8].

The multimodal interaction in SNOW was designed to be robust and easy to use. Therefore, relations between modalities were omitted, which means that the modalitites cannot be used *complementary*. In most situations, the user can arbitrary choose between the available modalities, therefore the modalities are *equivalent*. Our system does not support *redundant* modalities, because multimodal input immediately leads to the new dialog state. Multiple inputs in parallel leading to the same new state is not possible. Modalities can be enforced by the system (e.g., for not using voice for password entry), which fulfills the *assignment* property.

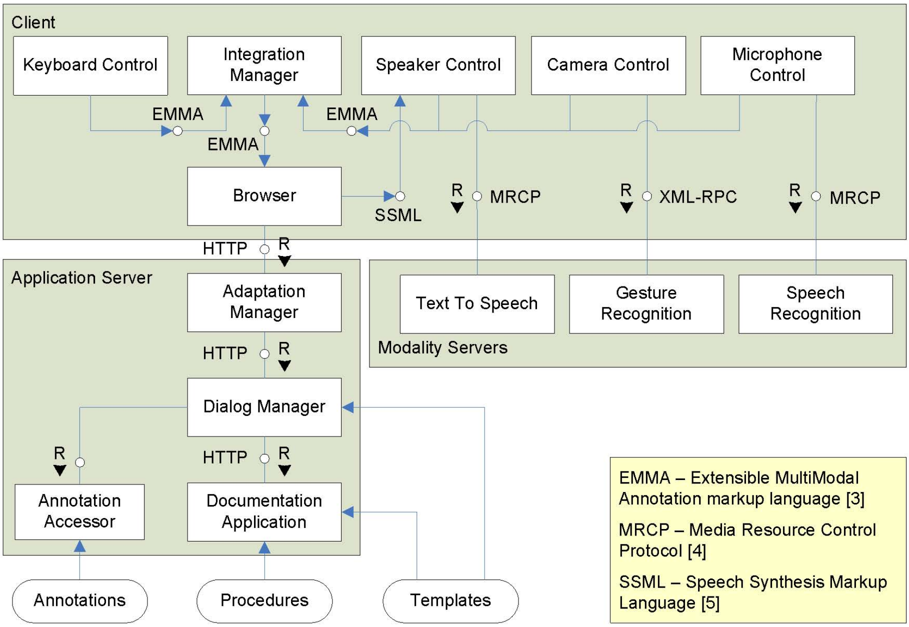


Fig. 1. Overall SNOW Architecture (as an F-M-C block diagram)

3 Enabling Multimodal Access

The demand for enabling multimodal access to procedures influenced the architecture within the SNOW project. Its consequences are explained in the following together with an analysis of the nature of the underlying, preexisting framework.

3.1 The Multimodal Interaction Framework

As a base for the development of the multimodal application, the SNOW consortium decided to use the *MultiModal Interaction Framework* (MMI-F, [10]), a specification under development by the W3C in its Multimodal Interaction Activity. This specification defines basic building blocks of a multimodal application, their responsibilities and collaboration partners.

The MMI-F specification defines six types of *basic components*. The relationships between the components described below are shown in Figure 2. The *input* components are responsible for handling of input modalities like audio, handwriting etc. The *output* components are used for handling of output channels like text, speech etc. The *interaction manager* is a component that coordinates the data and execution flow from and to the input and output components.

The *application functions* component is the “blackbox”, which implements the application functionality to be exposed in a multimodal fashion. The *session component* provides an interface used by the interaction manager to support

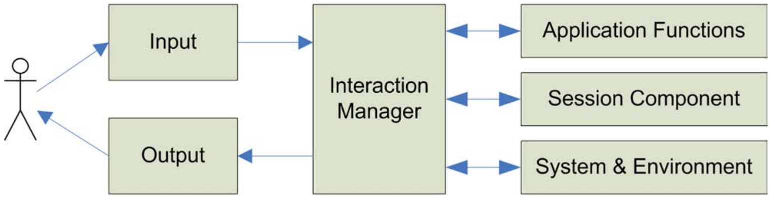


Fig. 2. The W3C Multimodal Interaction Framework (as shown in [10])

(transient or persistent) session management. The *system and environment* component is used by the interaction manager to detect changes of available devices, device capabilities or user preferences.

Input components are supposed to be built out of three smaller components: a *recognition* component that captures the input from the user, an *interpretation* component that tries to identify the semantics of the recognized input, and an *integration* component that combines the results of multiple interpretation components. This integration component is the part of the architecture that performs the modality fusion process [11] in which the logic for composite multimodal input [12] is located (e.g., to implement features like the well known "Put that there!" example [13]).

Likewise, output components are recommended to consist of a *generation* component for generating an intermediate output format, a *styling* component for adding layout information and a *rendering* component that finally outputs the information to the user. Typically, the generation component is performing the modality fission process [11].

However, the MMI-F is only influencing architectures with respect to multimodality. The specification explicitly states: "The multimodal interaction framework is not an architecture." During the execution of the SNOW project, it became clear that the MMI-F is a *role model* that could be used to introduce the multimodality aspect to existing architectures.

In [14], a role model is defined as "the description of a (possibly infinite) set of object collaborations using role types". A *role type* describes one objects view of another, while the *role* is the runtime property of an object conforming to a

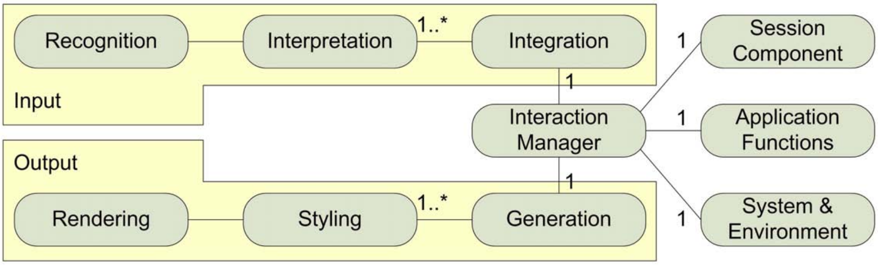


Fig. 3. The MMI-F as a role model

role type. Therefore, it can be stated that an object *plays* a role defined by a role type. Role models can also be used to describe and compose design patterns [15] as the participating classes are role types. In the following, the terms role and role type are applied to components instead of objects, a view that is more appropriate when considering architectures.

Understanding the MMI-F as a role model makes it easier to incorporate it into a concrete architecture as the one developed for the SNOW project. The same effect makes a design pattern as an abstraction more usable: by documenting the pattern, it becomes applicable to a variety of scenarios by more users. Figure 3 shows an interpretation of the MMI-F as a role model, where the input and output component have been refined into their subcomponents (cf. Figure 2).

3.2 Applying the Role Model

The role model extracted from the MMI-F has been mapped onto the SNOW architecture by applying the role assignments described below. The result of this mapping is illustrated in Figure 4.

The recognition role is played by the three input control components: keyboard, camera and microphone control. The interpretation role is fulfilled by the speech and gesture recognition servers and by the keyboard control component. Thus, the keyboard control component plays two roles, as the introduction of a special component for the interpretation of the keyboard input is unnecessary.

The interaction manager role is played by the dialog manager, which also fulfills the generation role. Two components, the annotation accessor and the documentation application play the application functions role.

The styling role is fulfilled by both the adaptation manager and the text-to-speech server. An interesting point to note here is that the direct association between the generation and the styling role fulfilled by the text-to-speech server is actually routed via the adaptation manager, browser and speaker control components. Lastly, the rendering role is played by the speaker control and the browser component.

The session component and system and environment roles are fulfilled by the application server's infrastructure.

Generally it is important to note that role associations do not always map directly to physical connections in the architecture, as it has been mentioned above for the association between generation and styling role. This kind of denormalization that happens to the role model might have several reasons: performance improvements, special requirements and/or deficiencies in the role model.

Probably, the MMI-F role model has such an deficiency because it does not consider a client/server split as it appears in the SNOW architecture. For such an architecture, it would be more appropriate to have an additional fission role [11], which represents a subset of the current generation role: in this case, only one output document would be created (on the server) which is then divided into multiple documents (on the client). In addition, this role could be responsible for synchronizing the output modalities, as it is necessary in advanced cases [16].

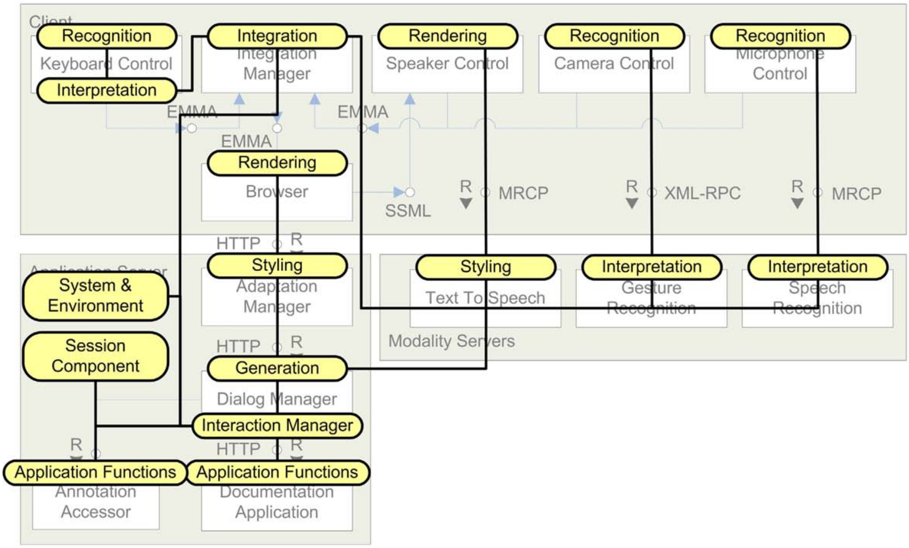


Fig. 4. The MMI-F Role Model mapped onto the SNOW Architecture

4 Achieving Domain Independence

The goal of keeping the SNOW architecture as domain-neutral as possible had consequences for both the architecture and the implementation, which are shown in this section. First, it is shown how a staged architecture could help keeping most of the components domain independent. Second, it is shown how a template engine was used to efficiently implement the described staged architecture.

4.1 Staged Architectures

Extensive domain independence has been reached within the SNOW project using a *staged architecture*. The term staged architecture has been borrowed from [17], where it is defined as a sequence of n subarchitectures, in which each stage produces the (data for the) next stage.

A staged architecture can be used very well to implement separation of concerns in an architectural sense: in SNOW, the stages separate the implementation of domain-specific from domain-neutral components *as well as* the implementation of device-specific from device-independent components. In some way, a staged architecture can be seen as an implementation of a “multi-level” separation of concerns (as opposed to the multi-dimensional separation, e.g., in [18]).

Figure 5 shows the subset of the components from Figure 1 which form a staged architecture leading to the creation of a document that is passed to a client, containing the stages represented by the following components:

The *Documentation Application* represents the domain-specific, device-independent stage, where the procedures are loaded from a special database,

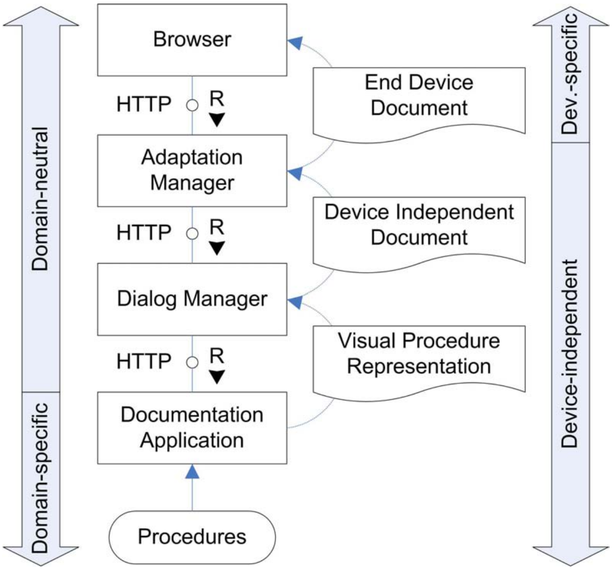


Fig. 5. Achieving Separation of Concerns with a Staged Architecture

which stores the procedures in a special topic map format called XTM-P (XML Topic Maps for Procedures, see [19]). These procedures are transformed into XML documents in a language called D3ML (Device Independent Multimodal Markup Language, see [20]).

In the next stage, the *Dialog Manager* implements a domain-neutral, device-independent dialog model. It interprets requests for documents (which might be maintenance procedures), fetches the documents from the Documentation Application and transforms them together with dialog model specific content (e.g., navigation bars) into a new D3ML document.

Afterwards, the *Adaptation Manager* transforms the D3ML document retrieved from the Dialog Manager into a device-specific document (e.g., an XHTML page, [21]) . As a component, the Adaptation Manager is still device-independent, but the documents produced by it are suitable only for particular device and hence device-specific.

For adapting the architecture to a different domain, only a limited number of components need to be exchanged. If the target domain is document-oriented like the maintenance domain, the only component that needs to be exchanged is the Documentation Application. The described architecture has also been used to implement a sample use case from the WearIT@Work project [22], this validation has shown that the architecture is indeed adaptable to different domains, in this case to a healthcare scenario.

In addition to this flexibility given by the staged architecture, the architecture allows to introduce components at the least specific level possible. E.g., the *Annotation Accessor* shown in Figure 1 has been developed in a way that is independent of a specific domain. It is a generic component that can be used to annotate all documents that expose identifiable anchors for the annotations.

SNOW's staged architecture resembles the invasive software composition approach from [23], as it allows documents to be created invasively by interweaving information obtained from one or multiple sources.

4.2 Implementation Using a Template Engine

An effect that might be seen as a drawback of staged architectures is that they might become arbitrary complex and thus inefficient and hard to maintain. This is only partially true for the following reasons.

First, a staged architecture might become inefficient if the stages are implemented as standalone applications, which are connected via standard networks. In the SNOW project, the stages have been implemented this way—the drawback of the necessary network communication is compensated by the possibility to deploy the architecture in a variety of combinations to multiple machines. If this deployment option is not necessary, it is also possible to collocate components in larger applications to remove the communication overhead. The variable deployment of the components also made the testing of the application easier as the responsibilities for the components has been distributed among the SNOW partners.

Second, the implementation of the various stages might be quite similar: in the SNOW project, the Documentation Application and the Dialog Manager are built on top of a template engine that has been developed as part of the project. This way, the implementation effort of the components is quite small.

The so-called XTL (*X*ML *T*emplate *L*anguage, [24]) template engine used in the SNOW project has some features that make it especially suitable for use in a staged architecture. The XTL engine is capable of fetching the instantiation data from multiple data sources. This is achieved by a *plug-in* architecture that allows the template engine to use data source specific languages within the template. E.g., in the SNOW project, plug-ins for the access to the XTM-P data layer, to XML data sources and to the Annotation Accessor component have been implemented. A single template might even use different data sources.

The XTL engine currently supports tags for the creation of attributes and text, the conditional and repeated inclusion of document fragments, for the inclusion as well as the definition and use of macros (for the prevention of duplicate fragments within the templates). The evaluation of these tags replaces them with data fetched from the instantiation data sources using the plug-ins, thereby invasively transforming the template into an instantiated document.

In addition to the plug-in concept, the XTL engine has a feature that allows templates to defer the instantiation of designated template parts. As there are multiple XTL engines embedded in the staged architecture, sometimes it is necessary to insert XTL language elements that are not intended to be evaluated when the template passes the first engine invocation, but rather are to be processed by the second (or third a.s.o.) engine invocation – as it is the case when necessary instantiation data is not yet available. It is even possible that the arguments for the XTL language elements are created by the evaluation of other XTL commands. This feature is called *generational bypassing*, as it allows to bypass the next n engines.

The template engine is used in two components. The Documentation Application uses the template engine just once to instantiate a domain-specific template with data from the procedures stored as XTM-P files. The utilization of the template engine by the Dialog Manager is more interesting as it is used two times: first, a presentation template is transformed, augmenting the output from the Documentation Application with presentational content (like links for navigation); second, the obtained intermediate document still has some evaluateable XTL tags (which bypassed the first transformation) for evaluation with data from the Annotation Accessor.

5 Related Work

Existing multimodal architectures are often based on the paradigm of software agents. One example is the QuickSet system [25], which is a distributed multi-agent architecture (namely the Open Agent Architecture$^{\text{TM}}$) to integrate various user interface components as well as a collection of distributed applications. Major design goal of QuickSet was to provide the same user input capabilities for handheld, desktop and wallsized terminal hardware, which were believed to be voice and gesture-based interaction, whereas gesture refers to 2D drawings on a screen. The QuickSet architecture groups all components around a central facilitator, which is responsible for routing, dispatching and triggering of all agent messages. Most of the QuickSet agents can be mapped to roles of the MMI-F, e.g., the agents for *TTS* or *User interface* act as rendering components, the *Multimodal integration* agent plays the role of the integration component and the *Application bridge* agent is responsible for application functions. As the QuickSet system was developed for a particular application domain (military simulation) using pre-defined devices (handheld PDA), component reuse by maximizing domain and device independence was no design criteria and thus not regarded during development.

The focus of Embassi [26] is on providing natural interfaces to people unexperienced with computers and the control of consumer electronics. The Embassi system supports the dynamic change of available modality analyzer or renderer components. The architecture is based on a considerable set of agents grouped into several layers. It implements an agent for merging input (Polymodal Input Module) and one for generating output (Polymodal Output Module), which can be associated to the integration and generation roles of the MMI-F. Moreover, the component sets named I (recognition), F (interpretation), O (rendering), R (generation), and D (interaction manager) are related to the MMI-F. The remaining MMI-F roles are distributed among the A, X, G, and C components. Embassi was developed for easing the interaction with consumer electronics, therefore including other applications involves considerable changes to the architecture.

Nightingale [27] is a ubiquitous system architecture, which supports multimodal and context aware applications spanning multiple devices. The context management includes an inference approach based on ontologies. The goal of Nightingale was to decouple multimodal applications from particular devices. Therefore, the architecture allows users to approach any device in her surroundings and immediately use those devices for the multimodal dialog. The multimodal architecture is based on a multi-agent system with agents for input recognition, application processing and output generation. Information sent from input to application agents is encoded in EMMA [3], which might be due to the overall orientation on the MMI-F. The Nightingale system is extendable by application writers, which only have to supply application/modality specific grammar and interpreter to input agents as well as application-specific style sheets to output agents. Therefore, the system can be considered as domain and device independent. Since the Nightingale system relies on the agent programming paradigm, new applications must also adhere to that, which restricts extension possibilities. Moreover, the application writer has do define input grammars and output style sheets. Together with the application itself, this causes a considerable development effort for writing new Nightingale applications.

The ICARE approach is a component-based system for supporting multimodal applications [28]. ICARE focusses on input processing, which results in different types of components for elementary and composite input processing. Application authors are supported in creating new ICARE applications by a graphical environment, called the ICARE platform. As the system focusses on multimodal input processing, it does not make any assumptions about domain or devices at this abstract level. However, the system seems to be quite immature, since its output capabilities were not tested as the authors admitted.

The work described in [29] describes a system, which allows the creation of applications independent of concrete user interfaces. Applications are described using GIML (Generalized Interface Markup Language). The interface toolkit is called GITK (Generalized Interface ToolKit), which allows new renderers to be plugged in later on (especially after creating a particular application). These renderers generate the user interface at runtime, resulting in different user interfaces according to different renderers. The system focusses on output of information, whereas input is at least of equal importance. Mechanisms for input integration were not considered. It is likely that automatically generated user interfaces are less user-friendly than hand-crafted user interfaces.

6 Conclusion

We have shown that the MMI-F is in fact a role model. In our opinion, implementors of the MMI-F specification can benefit from this knowledge, hence the specification should state this explicitly. A precise role model in the notation from [14] should be included to make the specification easier to understand. In addition to this, we recognized that a staged architecture is a powerful architectural style to implement multi-level separation of concerns. The predicted disadvantages could be circumvented by the means shown above.

A best architectural style for a multimodal application does not seem to exist. It is more likely that future multimodal applications will be built by using the architectural style best suitable for the application domain. In this case, overlaying the MMI-F role model will become a standard way of enabling multimodality in an architecture.

Acknowledgements

This work has been partially supported by the European Union within the FP6 IST STREP SNOW (FP6-511587). We would like to thank Henrik Lochmann and the anonymous reviewers for their very valuable comments on previous versions of this paper.

References

1. Pereira, A.C., Hartmann, F., Kadner, K.: A Distributed Staged Architecture for Multimodal Applications (Extended Abstract). In: SE 2007. Software Engineering 2007. Lecture Notes in Informatics (LNI), vol. 105, Köllen Verlag, Bonn (2007)
2. The SNOW Consortium: SNOW Project Homepage (2005) (visited May 10, 2006), http://www.snow-project.org
3. The World Wide Web Consortium: EMMA: Extensible MultiModal Annotation markup language (2005) (visited June 2, 2006), http://www.w3.org/TR/emma/
4. The Internet Engineering Task Force: A Media Resource Control Protocol (MRCP) (2006) (visited October 4, 2006), http://www.apps.ietf.org/rfc/rfc4463.html
5. The World Wide Web Consortium: Speech Synthesis Markup Language (SSML) Version 1.0 (2004) (visited October 4, 2006), http://www.w3.org/TR/speech-synthesis/
6. The Fundamental Modeling Concepts Consortium: Fundamental Modeling Concepts (2003) (visited May 29, 2006), http://www.f-m-c.org/
7. Knöpfel, A.: FMC quick introduction. FMC Publication (2003), http://www.f-m-c.org/
8. Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., Young, R.M.: Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE Properties. In: Proceedings of INTERACT'95, pp. 115–120 (1995)
9. Nigay, L., Coutaz, J.: Multifeature Systems: The CARE Properties and Their Impact on Software Design. AAAI Press, Stanford, California, USA (1997)
10. The World Wide Web Consortium: Multimodal Interaction Framework (2003) (visited May 29, 2006), http://www.w3.org/TR/2003/NOTE-mmi-framework-20030506/
11. Wahlster, W. (ed.): SmartKom: Foundations of Multimodal Dialogue Systems. Cognitive Technologies. Springer, Heidelberg (2006)
12. The World Wide Web Consortium: Multimodal Interaction Requirements (2003) (visited May 31, 2006), http://www.w3.org/TR/mmi-reqs/
13. Bolt, R.A.: Voice and gesture at the graphics interface. In: SIGGRAPH '80. Proceedings of the 7th annual conference on Computer graphics and interactive techniques, pp. 262–270. ACM Press, New York (1980)
14. Riehle, D., Gross, T.: Role model based framework design and integration, pp. 117–133. ACM Press, New York (1998)

15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
16. Kadner, K.: A flexible architecture for multimodal applications using federated devices. Visual Languages and Human-Centric Computing, 236–237 (2006)
17. Aßmann, U.: Architectural styles for active documents. Science of Computer Programming 56, 79–98 (2005)
18. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, Dordrecht (2000)
19. Kadner, K., Roussel, D.: Documentation for aircraft maintenance based on topic maps. In: Maicher, L., Sigel, A., Garshol, L.M. (eds.) TMRA 2006. LNCS (LNAI), vol. 4438, pp. 56–61. Springer, Heidelberg (2007)
20. Göbel, S., Hartmann, F., Kadner, K., Pohl, C.: A device-independent multimodal mark-up language. In: INFORMATIK 2006: Informatik für Menschen, Band 2, pp. 170–177 (2006)
21. World Wide Web Consortium: XHTML^{TM}1.0 The Extensible HyperText Markup Language, 2nd edn. (2000) (visited May14, 2006), http://www.w3.org/TR/xhtml1/
22. Carlsson, V., Klug, T., Ziegert, T., Zinnen, A.: Wearable computers in clinical ward rounds. In: IFAWC 2006. Proceedings of the third International Forum on Applied Wearable Computing, pp. 45–53 (2006)
23. Aßmann, U.: Invasive Software Composition. Springer, Heidelberg (2003)
24. Hartmann, F.: An architecture for an xml-template engine enabling safe authoring. In: DEXA 2006, Proceedings of the 17th International Conference on Database and Expert Systems Applications. pp. 502–507. IEEE Computer Society, Washington, DC, USA (2006)
25. Cohen, P.R., Johnston, M., McGee, D., Oviatt, S., Pittman, J., Smith, I., Chen, L., Clow, J.: Quickset: Multimodal interaction for distributed applications. In: Proceedings of ACM Multimedia 1997, pp. 31–40. ACM Press, New York (1997)
26. Elting, C., Rapp, S., Möhler, G., Strube, M.: Architecture and implementation of multimodal plug and play. In: ICMI '03. Proceedings of the 5th international conference on Multimodal interfaces, pp. 93–100. ACM Press, New York (2003)
27. West, D., Apted, T., Quigley, A.: A context inference and multi-modal approach to mobile information access. In: Artificial Intelligence in Mobile Systems, Nottingham, England, pp. 28–35 (2004)
28. Bouchet, J., Nigay, L., Ganille, T.: Icare software components for rapidly developing multimodal interfaces. In: ICMI '04. Proceedings of the 6th international conference on Multimodal interfaces, pp. 251–258. ACM Press, New York (2004)
29. Kost, S.: Dynamically generated multi-modal application interfaces. PhD thesis, TU Dresden (2006)

On the Modularity of Software Architectures: A Concern-Driven Measurement Framework

Cláudio Sant'Anna[1,2], Eduardo Figueiredo[2], Alessandro Garcia[2],
and Carlos J.P. Lucena[1]

[1] Computer Science Department, PUC-Rio, Brazil
[2] Computing Department, Lancaster University, UK
{claudios,lucena}@inf.puc-rio.br,
{a.garcia,e.figueiredo}@lancaster.ac.uk

Abstract. Much of the complexity of software architecture design is derived from the inadequate modularization of key broadly-scoped concerns, such as exception handling, distribution, and persistence. However, conventional architecture metrics are not sensitive to the driving architectural concerns, thereby leading a number of false positives and false negatives in the design assessment process. Therefore, there is a need for assessment techniques that support a more effective identification of early design modularity anomalies relative to crosscutting concerns. In this context, this paper proposes a concern-driven measurement framework for assessing architecture modularity. It encompasses a mechanism for documenting architectural concerns, and a suite of concern-oriented architecture metrics. We evaluated the usefulness of the proposed framework while comparing the modularity of architecture design alternatives in three different case studies.

1 Introduction

Modularity has been playing a pervasive role in early design stages even before the emergence of the software architecture discipline [13]. Software engineers consider modularity as a key principle when comparing architecture alternatives and analysing architecture degeneration [9]. In fact, software engineers are fostered to design good architectures by relying on a plethora of modularity mechanisms available in: (i) architecture description languages (ADLs), such as ACME [8], (ii) catalogues of architectural styles [2, 13], and (iii) well-know high-level design principles, such as narrow component interfaces, reduced architectural coupling, and the like.

However, strict reliance on these architecture mechanisms is not enough to achieve truly modular architecture designs [1, 3, 10]. Moreover, assessment and improvement of early design modularity is even more challenging. Quantitative assessment techniques are needed for evaluating and controlling architecture alternatives. Software metrics are a powerful means to provide modularity indicators of architecture design [3]. The software metrics community has consistently used notions as module coupling and cohesion to derive measures of software architecture quality [1, 10, 16].

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 207–224, 2007.

In fact, the conception of the right architecture decomposition is still a deep bottleneck to the software design process. A number of widely-scoped architectural concerns that emerge in early development phases need to be simultaneously modularized. An archi- tectural concern is some important part of the problem that we want to treat in a modular way at the architecture specification [4, 5]. Much of the complexity of software archi- tecture design is derived from the inadequate modularization of architectural concerns, such as graphical user interface (GUI), exception handling, and persistence. Architects tend to naturally give priority or focus on the modularization of certain concerns, while choosing a certain combination of existing architecture styles or relying on a particular way for architecture decomposition. As a result, a number of concerns end up having a crosscutting impact on the system architectural decomposition, and systematically affecting the boundaries of several architectural elements, such as components and their interfaces [5, 14].

Although typical architecture modularity problems are related to the inadequate modularization of concerns, most of the current quantitative assessment approaches do not explicitly consider concern as a measurement abstraction. A number of architecture quantitative assessment methods are targeted at guiding decisions related to modularity, without calibrating the measurement outcomes to the driving architectural concerns. It imposes certain shortcomings, such as the ineffective identification of desirable and undesirable couplings. Also, this necessity becomes more apparent in an age that a number of different forms of architecture decompositions are emerging. For example, aspect-oriented software architectures and feature-oriented product-line architectures [15] support different composition mechanisms for enhancing the separation of certain concerns. A number of case studies have pointed out that detection of certain design flaws can be observed in early design stages [14, 17, 23].

Software architects need, therefore, quantitative assessment approaches that support them to identify modularity anomalies related to inadequate modularization of architectural concerns. We believe that concern-driven quantitative assessment improves the architecture modularity analysis, because it makes more evident the overall influence of the (in)adequate modularization of widely-scoped design concerns. In this context, the contributions of this paper are threefold: (i) a list of shortcomings associated with conventional architecture modularity measurement, (ii) a concern-driven framework for assessing architecture modularity, and (iii) a systematic evaluation of the proposed framework. In addition to a concern-sensitive metrics suite, the framework includes a mechanism for documenting concerns in architecture descriptions. We evaluated the usefulness of the measurement framework while comparing the modularity of architecture design alternatives in the context of three case studies.

The remainder of this paper is organised as follow. Section 2 motivates the proposed measurement framework while discussing the limitations of conventional architecture metrics. Section 3 defines the framework terminology and metrics. Section 4 discusses the evaluation of the framework in the case studies. Finally, Section 5 presents the concluding remarks comparing our work to related research.

2 Why Concern-Driven Architecture Assessment?

Current quantitative architecture assessment approaches [1, 10, 16] usually rely on traditional abstractions such as component (or module) and its interfaces in order to undertake the measurements. Based on these abstractions, they define and use metrics for quantifying attributes such as coupling between components, component cohesion, interface complexity, and so forth. Figure 1 depicts an architecture that will serve as a running example throughout this paper, and as an illustration for the limitations of conventional architecture metrics. It shows a partial, simplified graphical representation of the architecture description of a real Web-based information system, called Health Watcher [4, 17]. The design is structured mainly following the layer architectural style [2].

In an architecture specification, a concern is addressed (or realized) by a number of architecture elements, such as components, interfaces and operations. The gray boxes in Figure 1 represent the concerns addressed by the Health Watcher architecture elements. For instance, the business concern is addressed by the Business_Manager component and its interfaces, except the useTransaction interface, which addresses the persistence concern; persistence is also an architectural concern which is realized by the Data_Manager and Transaction_Manager components, by the useTransaction interface of the Business_Manager component and by the transactionExceptionEvent and repositoryExceptionEvent events raised or captured in a number of interfaces, such as savingService. The exception handling concern is reified by the transactionExceptionEvent, repositoryExceptionEvent and communicationExceptionEvent events. Besides, in the example shown in Figure 1, the Business_Manager component, for instance, is connected to three other components. The distributedSavingService interface has four operations and events. In the light of this example, the next subsections discuss the limitations of current architecture metrics.

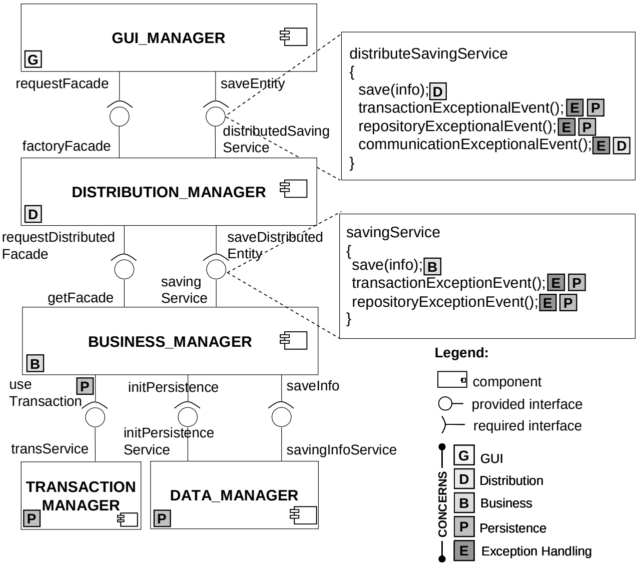


Fig. 1. Architecture of the Health Watcher system

2.1 Inaccuracy on Modularity Analysis

When choosing certain architecture abstractions, styles and mechanisms for decomposition, architects may leave some concerns non-modularized. In other words, these concerns are not satisfactorily captured in separate modular units in the architecture description, i.e. only localized within one or a few components with well defined interfaces. In the architecture shown in Figure 1, the exception handling concern is partially addressed by abnormal events, such as transactionExceptionalEvent and repositoryExceptionalEvent, exposed in several interfaces of the components. In this system, it is important to well modularize the global exception detection strategies because they are similarly applied over several key services defined by the system architecture.

Current architecture metrics are not able to highlight that this concern has a wide impact on several interfaces. The main problem is that they do not rely on the identification of the architectural elements related to each concern, thereby causing a number of false negatives in the architecture assessment process. The reason is that typical architecture metrics are not able to explicitly capture this kind of modularity-related impairment as, for instance, exception handling is an architectural property typically diffused all over the architecture elements. Hence, this implies that existing metrics are often inaccurate to support the identification of non-modularized architectural concerns.

2.2 Impairments on Change Management

The dependence between system concerns is a pivotal information for software architects in order to support design change management. Changes on a concern may impact concerns that depend on it. However, current coupling metrics are inaccurate to identify architectural inter-concern dependencies. Coupling architecture metrics quantify the dependence between components, assessing, in this way, the dependence between only the primary concerns modularized within components.

In Figure 1, the Business_Manager component depends on two other components, namely Transaction_Manager and Data_Manager, and one component (Distribution_Manager) depends on it. However, concerns which are not entirely modularized by the architecture abstractions do not have modular boundaries, in the sense that their boundaries are not well defined by component interfaces. Hence the dependence between such non-modularized concerns or even between non-modularized and modularized concerns cannot be measured by traditional measures. Therefore, these metrics cannot support the assessment of the impact of non-modularized concerns on other architectural concerns. In Figure 1, the distribution concern depends on the persistence concern because the Distribution_Manager component includes persist- ence-related exception events, which are not modularized by any component. Moreover, the exception handling concern interacts with the persistence concern because the transactionExceptionEvent and repositoryExceptionEvent events are related to both concerns.

2.3 Inaccuracy on Identifying Instabilities

Conventional metrics are also inaccurate to identify potential unstable architecture elements. Architecture stability is usually measured by the dependence between components [16]. A component that depends upon nothing at all is considered to be very stable, since a change in any other component is unlikely to ripple up to them and cause them to change. Therefore, current architecture metrics evaluate the stability of a component by measuring its coupling with other components [16].

In Figure 1, the Distribution_Manager component depends upon the Business_ Manager component, so that changes in the latter can be propagated to the former. However, since some concerns are not totally modularized within components and cut across several components, the stability of a component has also to do with the number of concerns that affect it. The more concerns affect a component the more unstable that component is, because it can be changed due to changes related to those concerns. In Figure 1, the Distribution_Manager component is affected by the exception handling and persistence concerns, once its interface encompasses exception events related to persistence – transactionExceptionEvent and repositoryExceptionEvent. This same reasoning is also valid for interface stability. That is, the more concerns affect a interface the more unstable that interface is, because it can be modified due to changes related to those concerns. Since the current metrics do not take into account the number of concerns that affect the architecture elements, they are not able to capture this dimension of architecture stability.

2.4 The Tyranny of Dominant Modularity Attributes

Software architecture measurement also suffers from what we call the tyranny of the dominant architectural modularity principles. The fact that the assessment of certain principles, such as low coupling and narrow interfaces, are overemphasized, other equally important design principles, such as separation of concerns, have been neglected in architecture measurement processes. It might hamper trade-off analysis in scenarios where priorities need to be given, for instance, either to the separation of a specific concern or to the degree of coupling of architectural components involved. More importantly, it is not possible to understand how the separation of certain architectural concerns influence other modularity attributes, such as coupling.

As shown in Figure 1, the exception handling concern affects the distributedSavingService and savingService interfaces, since they have to expose exceptional events. These events contribute to increase the complexity of those interfaces. Even though the traditional metrics can provide information about the interface complexity, they do not support the architects to reason about the fact that the exception handling concern is the main contributor for that complexity. Hence, the identification of the impact of architectural concerns on traditional attributes is hindered.

3 A Concern-Driven Measurement Framework

This section is targeted at defining a concern-sensitive measurement framework for assessing architecture modularity. It complements existing architecture metrics by explicitly promoting concern as a measurement abstraction. The framework aims at supporting the software engineers to: (i) anticipate modularity problems caused by architecturally-relevant concerns, and (ii) compare alternatives of architecture design solutions with respect to their ability to modularize distinct sets of prioritized concerns.

Our framework mainly relies on evaluating the modularization of architectural concerns. Therefore it includes metrics for quantifying separation of concerns and their interactions. For instance, it quantifies the diffusion of a concern realization within architecture specification elements, such as components and interfaces. The metrics suite also evaluates how a particular concern realization affects traditional attributes such as coupling, cohesion and interface complexity. Hence it includes metrics for assessing these attributes.

Our concern-oriented metrics focus on the evaluation of software architecture representations, such as UML-based or ADL specifications. Also, in order to consider concern as an abstraction in the measurement process, there is a need to explicitly document the concerns in the architecture. Therefore, our approach also includes a notation to support the architect with the documentation of the driving architectural concerns (Section 3.6). Using this notation, the architect can assign every architecture element (components, interfaces, and operations) to one or more concerns.

Table 1 presents a summary of the architecture metrics suite with a brief definition for each of the metrics and their association with distinct modularity attributes they measure. The metrics definition is agnostic to specific ADLs. Therefore, in order to apply the metrics, it is necessary to adapt their definition to the specific abstractions of the architecture description approach in use. The following subsections present the terminology considered in the definition of the metrics (Section 3.1), and the detailed definition of each metric of the framework (Section 3.2 – 3.5).

Table 1. Suite of Concern-Driven Architectural Metrics

Attribute	Metric	Definition
Concern Diffusion	Concern Diffusion over Architectural Components (CDAC)	It counts the number of architectural components which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Interfaces (CDAI)	It counts the number of interfaces which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Operations (CDAO)	It counts the number of operations which contributes to the realization of a certain concern.
Coupling Between Architectural Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which the assessed concerns share at least a component.
	Interface-level Interlacing Between Concerns (IIBC)	It counts the number of other concerns with which the assessed concerns share at least an interface.
	Operation-level Overlapping Between Concerns (OOBC)	It counts the number of other concerns with which the assessed concerns share at least an operation.
Coupling Between Components	Afferent Coupling Between Components (AC)	It counts the number of components which require service from the assessed component.
	Efferent Coupling Between Components (EC)	It counts the number of components from which the assessed component requires service.
Component Cohesion	Lack of Concern-based Cohesion (LCC)	It counts the number of concerns addressed by the assessed component.
Interface Complexity	Number of Interfaces	It counts the number of interfaces of each component.
	Number of Operations	It counts the number of operations in the interfaces of each component.

Looking again to the architecture in Figure 1, using the proposed metrics (Table 1) we can now quantify the effects of the exception handling concern in the architecture.

After documenting that the exception events are related to the exception handling concern, we can compute the concern-driven metrics. The results will show that the exception handling concern is spread over several components and interfaces. Moreover, the results of the Lack of Concern-based Cohesion metric for the GUI_Manager, Distribution_Manager and Business_Manager components will show that there is more than one concern present in each of those components. In this way, the architect will be warned that in the Business_Manager component, for instance, besides the business concern, there are other concerns contributing for the complexity of the component.

3.1 Terminology

In order to define the concern-oriented architectural metrics, we present here the fundamental terminology for architecture elements we have used.

3.1.1 System Architecture

The measurement framework is basically rooted at the component-and-connector models [21] of the system architecture, as they subsume the core abstractions in architecting processes, such as components, interfaces, and their relationships. They are also the models supported by a plethora of ADLs and, more notably, by UML. The examples based on the component-and-connector views are described in this paper using UML 2.0.

Definition 1: Component-and-Connector View. A component-and-connector view of the system architecture consists of a set of components. Each component has provided and required interfaces. Each interface encompasses one or more operations and/or events (herein referred only as operations). A required interface can be linked to a provided interface by means of a connector. The architecture in Figure 1 consists of five components – GUI_Manager, Distribution_Manager, Business_Manager, Transaction_Manager, and Data_Manager. The Distribution_Manager component has a provided interface called distributed SavingService, which is connected to a required interface of the GUI_Manager component, called saveEntity. The distributedSavingService interface has four operations.

3.1.2 Interaction Between Components

Components interact with each other by means of their interfaces. These interactions are used to define metrics capturing the coupling between components.

Definition 2: Afferent Relationship. A component *A* has an afferent relationship with a component *B* if a required interface of *B* is linked to a provided interface of *A*, which means that *B* requires services from *A* and, therefore, depends upon *A*. In Figure 1, the Distribution_Manager component has an afferent relationship with the GUI_Manager component.

Definition 3: Efferent Relationship. A component *A* has an efferent relationship with a component *B* if a provided interface of *B* is linked to a required interface of *A*, which means that *B* provides services to *A* and, therefore, is depended upon by *A*. In Figure 1, the Distribution_Manager component has an efferent relationship with the Business_Manager component.

3.1.3 Architectural Concern

The architecture description of a system encompasses several concerns which come directly from the system requirements or emerge during the architecting process. In the example of the Health Watcher architecture (Figure 1), there are several concerns such as GUI, Business, Distribution, Persistence and Exception Handling. As mentioned before (Section 2), in an architecture specifi- cation, a concern is realized by a set of architecture elements, such as components, interfaces and operations. Our measurement framework focuses on architectural concerns that eventually evolve into concrete pieces of code and contribute directly to the functionality of the system, such as the aforementioned ones. Our metrics do not rely on concerns that: (i) influence how the system is built but do not trace to any specific piece of code, such as performance, or (ii) influence the software process and are not observable when the system executes, such as maintainability.

Definition 4: Architectural Concern. A concern *C1* consists of a list of architecture elements assigned to it which can be components, interfaces and operations. If a component *A* is assigned to *C1,* all the interfaces of *A* are considered also assigned to *C1,* except those which are explicitly assigned to other concern. The same reasoning is applied to interfaces, that is, if a interface *I* is assigned to *C1*, all the operations in *I* are considered also assigned to *C1,* except those which are explicitly assigned to other concern.

The architect has the role of assigning every architecture element to one or more system concerns. We provide a mechanism for supporting the architect on documenting the mapping of concerns to architecture elements (Section 3.6). It is out of the scope of this paper providing an approach to support the architect on deciding which elements are related to a concern. However, we are aware that different selections of concerns and their assignment to architecture elements may imply different results in the measurement process. Therefore, in the future, we plan to specify guidelines for systematically identifying concerns in the architecture. For now, in our case studies, in order to make the identification of the architectural elements related to the concerns simpler and more systematic, we follow a specific guideline: assign a concern to an architectural element (component, interface or operation) if the complete removal of the concern requires with certainty the removal of the element. This guideline is inspired on the guidelines proposed by Eaddy et al [24]. There are some works (e.g. [18]) that aim at supporting the identification of concerns in software artefacts. Hence, these approaches and their tools can be used for identifying concerns in the architecture. Also, we have consistently observed in our case studies that, as expected, the mapping to architecture elements is easier and less time-consuming than the mapping to elements of source code due to the much more coarse-grained level of abstraction in architecture descriptions.

3.1.4 Concern Interactions

As stated in the previous section, the business concern is concretized by the Business_Manager component in the architecture shown in Figure 1. Nevertheless, note that even though the main purpose of Business_Manager component is to address the business concern, it also includes elements related to other concerns such as the useTransaction interface (persistence concern) and exception handling events. Note also that some elements, such as the repositoryExceptionEvent event, are related to more than one concern (exception handling and persistence). This occurs because some concerns are not well modularized and, as a consequence, are not totally localized in components whose only purpose is to address them. As a result, concerns interact to each other not only by means of the relationship between components, but also because sometimes more than one concern is present in the same architecture element. Some of the metrics in our framework target at assessing the interaction between concerns. In order to define them, we first define here three forms of concern interaction which they take into account.

Definition 5: Component-level Interlacing. A concern *C1* is interlaced at the component level with another concern *C2* if *C1* and *C2* have one or more components in common, but distinct parts of that component are assign to each concern. We consider it in two ways: (i) a component is assigned to *C1*, and one or more interfaces of the same component are assigned to *C2*, or (ii) one or more interfaces of a component are assigned to *C1*, and one or more distinct interfaces of the same component are assigned to *C2*. In Figure 1, the business concern is interlaced at the component level with the persistence concern, once the Business_Manager component is assigned to it, but also has one interface (useTransaction) assigned to the persistence concern.

Definition 6: Interface-level Interlacing. A concern *C1* is interlaced at the interface level with another concern *C2* if *C1* and *C2* have one or more interface in common. It can happen in two ways: (i) an interface is assigned to *C1*, and one or more operations of the same interface are assigned to *C2*, or (ii) one or more operations of an interface are assigned to *C1*, and one or more distinct operations of the same interface are assigned to *C2*. In the example of Figure 1, the business concern is interlace with the exception handling concern at the interface level since the business-related interface savingService includes two operations assign to the exception handling concern.

Definition 7: Operation-level Overlapping. A concern *C1* is overlapped at the operation level with a concern *C2* if one or more operations are assigned to both *C1* and *C2*. This interaction is different from the previous one because here at least one same operation is entirely assigned to both concerns. In the architecture shown on Figure 1, the persistence concern is overlapped with the exception handling because the repositoryExceptionalEvent and the transactionExceptionalEvent operations are assigned to both concerns.

3.2 Metrics for Concern Diffusion

We next define our measures for concern diffusion based on the terminology presented in Section 3.1. These metrics are defined on counting, for each architectural concern, the number of architecture elements assigned to it. They are devoted to calculate the degree to which a single concern in the system maps to distinct architectural elements.

Definition 8: Concern Diffusion over Architectural Components (CDAC). CDAC for a concern *C1* counts the number of components in the system architecture entirely assigned to *C1*. The counting also includes the number of components where there is at least one interface assigned to C1, and the number of components where there is at least one operation assigned to *C1*. According Figure 1, the value of CDAC for the persistence concern is 4 since it is present in: (i) the Transaction_Manager and Data_Manager components, (ii) the useTransaction interface of the Business_ Manager component, and (iii) the two operations of the distributed Saving Service interface of the Distribution_Manager component. Therefore, the persistence component is spread over four components.

Definition 9: Concern Diffusion over Architectural Interfaces (CDAI). CDAI for a concern *C1* counts the number of interfaces in the system architecture entirely assigned to *C1* (which includes the interfaces of components entirely assigned to *C1*), plus the number of interfaces where there is at least one operation assigned to *C1*. According Figure 1, the CDAI value for the persistence concern is 6 since four interfaces are entirely assigned to it – transService, initPersistenceService, savingInfoService and useTransaction, and there are also operations assigned to it in the saveDistributedEntity and distributedSavingService interfaces.

Definition 10: Concern Diffusion over Architectural Operations (CDAO). CDAO for a concern *C1* counts the number of operations in the system architecture assigned to *C1* (which includes the operations of interfaces entirely assigned to *C1*). In Figure 1, CDAO for the persistence concern counts all the operations in the interfaces of the Data_Manager and Transaction_Manager components, plus all the operations in the useTransaction operation, and plus the repositoryExceptionalEvent and the transaction ExceptionalEvent in the savingService and distributedSavingService interfaces.

3.3 Metrics for Coupling Between Concerns

The measures for coupling between concerns are defined based on the kinds of concern interactions defined in Section 3.1.4. These metrics are targeting at assessing concern dependences caused by concerns that are not well modularized. In other words, they deal with coupling between concerns which are not introduced by the dependence between components.

Definition 11: Component-level Interlacing Between Concerns (CIBC). CIBC for a concern *C1* counts the number of other concerns with which *C1* is interlaced at the component level (Component-level Interlacing – see Definition 5). In Figure 1, the CIBC value for the business concern is 1 because it is interlaced with the persistence concern at the component level, since the Business_Manager component include an interface entirely dedicated to persistence (useTransaction interface).

Definition 12: Interface-level Interlacing Between Concerns (IIBC). IIBC for a concern *C1* counts the number of other concerns with which *C1* is interlaced at the interface level (Interface-level Interlacing – see Definition 6). In Figure 1, the IIBC value for the business concern is 2, since there are operations assigned to the persistence and exception handling concerns in the saveDistributedEntity interface of the Business_Manager component.

Definition 13: Operation-level Overlapping Between Concerns (OOBC). OOBC for a concern *C1* counts the number of other concerns with which *C1* is overlapped at the operation level (Operation-level Overlapping – see Definition 7). In Figure 1, the OOBC value for the exception handling concern is 2 because there are operations that, besides being assigned to the persistence concerns, are also assigned to the distribution (communicationExceptionalEvent) and persistence (transactionExceptional Event and repositoryExceptionalEvent) concerns.

3.4 Component Cohesion Metric

Here we define one concern-sensitive metric for cohesion. This metric also rests on the mapping of the system concerns to the architecture elements. However, differently from the metrics presented in Sections 3.2 and 3.3, it is measured from the component point of view. In other words, the results of this metric is obtained per component, and not per concern as in the metrics defined in the previous sections. Our cohesion metric are defined on counting, for each component, the number of concerns it addresses.

Definition 14: Lack of Concern-based Cohesion (LCC). LCC for a component *A* counts the number of concerns which *A* is assigned to, plus the number of distinct concerns which the interfaces of *A* are assigned to, plus the number of distinct concerns which the operations in the interfaces of *A* are assigned to. In the architecture in Figure 1, the LCC value for the Business_Manager component is 3 because it is assigned to three concerns: (i) the entire component is assigned to the business concern, (ii) its useTransaction interface is assigned to the persistence concern, and (iii) two operations in one of its interfaces are assigned to the exception handling concern.

3.5 Metrics for Coupling and Interface Complexity

Our measurement framework also includes metrics for quantifying coupling between components and interface complexity. These metrics are based on traditional metrics already defined [10, 16]. We just adapted them to comply with our terminology. The coupling metrics (Definitions 15 and 16) are based on the definitions presented in Section 3.1.2.

Definition 15: Afferent Coupling Between Components (AC). AC for a component *A* is the number of distinct components with which *A* has afferent relationship (Definition 2 – see Section 3.1.2).

Definition 16: Efferent Coupling Between Components (EC). EC for a component *A* is the number of distinct components with which *A* has efferent relationship (Definition 3 – see Section 3.1.2).

Definition 17: Number of Interfaces *(NI).* NI for a component *A* counts the number of interfaces of *A*.

Definition 18: Number of Operations (NO). NO for a component *A* counts the number of operations in all interfaces of *A*.

3.6 Concern Templates

In order to support the application of the concern-driven metrics, we defined what we call as *concern template*. A concern template captures the architecture elements associated with key concerns, letting the architects to represent all the architectural implications related to a concern in a single place. The template includes the following information: (i) *name* of the concern; (ii) *architecture elements*, such as components, interfaces, relationships, operations, events and so forth, related to the concern; and (iii) *composition rules* to describe how the elements with respect to a concern affect architectural elements related to other concerns.

Concern: Exception Handling

Architecture Elements:

    transactionExceptionalEvent() ;
    repositoryExceptionalEvent() ;
    communicationExceptionalEvent() ;

Composition Rules :

Add event communicationExceptionalEvent() to
    interface distributedSavingService;
SavingSet = distributedSavingService, savingService;
Forall I in SavingSet
  Add event transactionExceptionalEvent() to I;
  Add event repositoryExceptionalEvent() to I;

Fig. 2. Concern Template

Figure 2 shows how to use the template to support the description of architectural designs relative to the exception handling concern in the Health Watcher system architecture (Figure 1). The template shows the three operations assigned to the persistence concern and how they are composed with the other elements in the architecture, that is, in which interfaces they are present. Based on this information is it possible to compute all the concern-driven metrics we defined.

4 Evaluation and Discussion

We undertook three case studies in order to carry out an evaluation of our measurement framework. We have used the framework to perform modularity comparisons of two different architecture alternatives for each of three systems. The first alternative is always an aspect-oriented (AO) architecture. The second version is based on one or more specific architectural styles [2, 13], herein referred as non-aspectual (non-AO) architecture. We have chosen to compare the modularity of AO and non-AO architectures because aspect-oriented abstractions [4, 15] are claimed to modularize certain concerns in a superior manner in contrast with conventional architectural decompositions. Nevertheless, since our metrics are paradigm independent, they can be used to assess and compare architectures which follow any style. They can also be useful to compare different architectures alternatives in the same paradigm, for instance, two aspect-oriented alternatives or two object-oriented alternatives.

Aspect-oriented software development has initially emerged as a programming technique [15], however some work on aspect-oriented architecture specification have been developed recently [4]. Based on these works, we consider here that, in addition to components, interfaces, connectors and operations, the component-and-connector view of an aspect-oriented architecture is depicted using two new abstractions: aspectual component and aspectual connector [4]. An *aspectual component* represents a component which will be concretized by at least one aspect at the implementation level [4]. An *aspectual connector* [4] represents a relationship between an aspect and other components. For further information about the definitions of these abstractions refer to [4].

The first of our series of case studies was a multi-agent system framework, called AspectT [5]. The evaluation included an architecture mainly based on the Mediator pattern [19], in addition to an AO architecture. The second study encompassed publisher-subscriber [2] and aspect-oriented versions of the MobiGrid architecture [20], a framework used to develop mobile agents in Grid environments. The third case study involved the AO and non-AO layered architecture of the Heath Watcher system, presented in Section 2. The systems were ideal for our evaluation study because the chosen systems have stringent modularity requirements due to the demand for producing reusable, adaptable and evolvable architectures. Moreover, they are realistic systems that involve emphasis on different concerns and their distinct compositions. In addition, previous studies [14, 17] have interestingly figured out that architecture degeneration would be avoided whether the use of aspects was planned since the design outset in certain parts of the Health Watcher architecture.

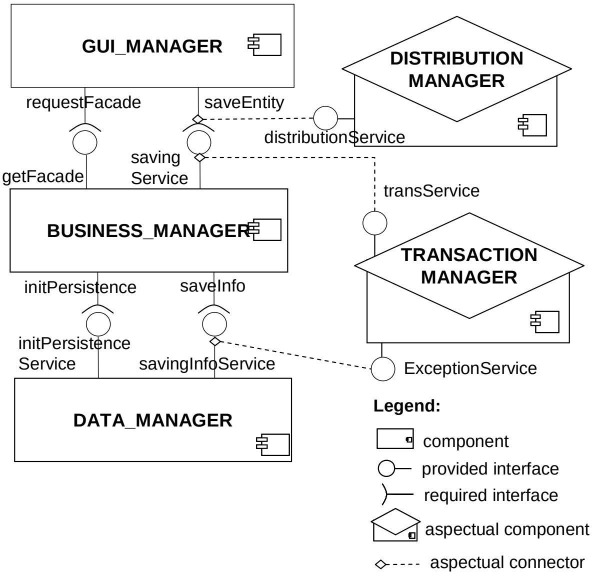


Fig. 3. AO Architecture of Health Watcher System

Figure 3 shows a partial view of the AO architecture of the Health Watcher system. In the following we partially present the data gathered with the application of our metrics in the architectures of the three systems and discuss how they support overcoming the limitations of conventional metrics (Section 2). Due to space limitation, we will give more attention to the results for the Health Watcher architecture, since it is our running example in this paper. The complete description of the gathered data is reported elsewhere [7].

Identification of non-modularized concerns. Table 2 presents the measures for concern diffusion (Section 3.2) and coupling between concerns (Section 3.3). The presented data is relative to the AO and non-AO versions of the Health Watcher architectures. The results for the concern diffusion metrics (CDAC, CDAI and CDAO) show that the persistence and exception handling concerns are spread over more architecture elements in the non-AO solution. For instance, in the non-AO architecture, the persistence affects more components (CDAC metric) – 5 vs. 2, more interfaces (CDAI metric) – 22 vs. 9 – and more operations (CDAO metric) – 154 vs. 45. This occurs mainly because in the AO solution the persistence-specific exceptional events are modularized within the Transaction_Manager aspectual component and, as a consequence, do not need to be addressed by the interfaces of Business_Manager, Distribution_Manager and GUI_Manager components as in the non-AO solution.

Our framework was also useful to analyse the differences on the concern diffusion between the AO and the mediator-based architectures of the AspectT (first case study). The results on Table 3 show that the AspectT concerns are spread over more components, interfaces and operations in the non-AO architecture (mediator-based). It happens because the component which plays the mediator role needs to inevitably embody functionalities from the different concerns; therefore it includes interfaces and operations related to them.

Table 2. Health Watcher: Concern Diffusion and Coupling between Concerns measures

Concerns	**Concern Diffusion over Architectural Components (CDAC)**		**Concern Diffusion over Architectural Interfaces (CDAI)**		**Concern Diffusion over Architectural Operations (CDAO)**		**Component-level Interlacing Between Concerns (CIBC)**		**Interface-level Interlacing Between Concerns (IIBC)**		**Operation-level Overlapping Between Concerns (OOBC)**	
	non-AO	*AO*	*non-AO*	*AO*	*non-AO*	*AO*	*non-AO*	*AO*	*non-AO*	*AO*	*non-AO*	*AO*
GUI	1	1	2	2	14	14	0	0	3	0	0	0
Distribution	2	1	5	1	51	16	0	0	3	1	**1**	**1**
Business	1	1	8	9	57	57	**2**	**0**	2	0	0	0
Persistence	**5**	**2**	**22**	**9**	**154**	**45**	2	0	**4**	**1**	**1**	**1**
Concurrency	2	1	2	2	4	4	2	0	0	0	0	0
Exception Handling	**5**	**2**	**24**	**8**	**156**	**52**	0	0	3	2	**2**	**2**

Identification of dependence between architectural concerns. Table 2 also presents the outcomes for coupling between concerns (CIBC, IIBC and OOBC metrics). The results show that modularizing some concerns with aspectual components eliminates the interlacing between concerns at the component level in the AO architecture. Note, for instance, that, in the non-AO architecture, the business concern is interlaced with two concerns at the component level (CIBC metric). Checking the architecture description and its concern templates, we can see that one of these two concerns is the persistence concern. This interlacing is caused by the useTransaction required interface (Fig. 1), which is related to the persistence concern, in the Business_Manager component. This interface is not necessary in the AO architecture, because the Transaction_Manager aspectual component provides the transaction control service by capturing the requisitions to the saving services directly in the savingService provided interface (Fig. 3).

The coupling related to interface-level interlacing is also lower in the AO solution. For instance, the persistence concern is interlaced with 4 other concerns at the interface level in the non-AO architecture, against only one concern in the AO solution (IIBC metric). This is due the fact that persistence-specific exceptional events, such as transactionExceptionEvent and repositoryExceptionEvent, are spread over interfaces of the Business_Manager, Distribution_Manager and GUI_Manager components in the non-AO architecture. On the other hand, in the AO solution, these events are handled by the Transaction_Manager aspectual component which capture them directly in the provided interfaces of the Data_Manager component, represented in Figure 3 by the saveInfoService interface (in fact, in the real architecture of the Health Watcher, the Data_Manager component has seven interfaces).

Table 3. AspectT: Concern Diffusion Measures

Concerns	Concern Diffusion over Architectural Components (CDAC)		Concern Diffusion over Architectural Interfaces (CDAI)		Concern Diffusion over Architectural Operations (CDAO)	
	Non-AO	*AO*	*Non-AO*	*AO*	*Non-AO*	*AO*
Interaction	2	1	9	3	22	10
Adaptation	2	1	6	2	34	5
Autonomy	2	1	7	3	80	31
Collaboration	2	1	6	4	87	37
Mobility	2	1	3	3	35	20
Learning	2	1	4	2	16	6

Finally, the results regarding the metric for operation-level overlappings show that the AO solution for the Health Watcher architecture was not able to improve this kind of coupling. In both AO and non-AO architectures the Exception Handling concern are overlapped with the Persistence and Distribution concerns due to the exceptional events specific to these two concerns. Even though these two concerns are modularized within aspectual components, the interfaces of these components still have to include the exceptional events.

Breaking the tyranny of the dominant architectural modularity attributes. Table 4 presents the results for three metrics: Lack of Concern-based Cohesion (LCC), Number of Interfaces (NI) and Number of Operations (NO). Note that unlike the results for the metrics on Tables 2 and 3, the results for the metrics on Table 4 are gathered per component. Analysing the results of the Number of Interfaces (NI) and Number of Operations (NO) metrics, which are conventional metrics, we can see that three components, namely GUI_Manager, Distribution_Manager and Business_Manager have more complex interfaces in the non-AO architecture. For instance, the Business_Manager component has more interfaces (11 vs. 9) and more operations (64 vs. 57) in the non-AO architecture.

Although this information is important, it does not give any clue about the reasons for that. In this way, the results for the Lack of Concern-based Cohesion complement this information in the sense that it shows that those components which have more complex interfaces also has more concerns affecting them in the non-AO solution. Therefore, these concerns can be one of the causes for the higher interface complexity.

Table 4. Health Watcher: Component Cohesion and Interface Complexity Measures

Components	**Lack of Concern-based Cohesion (LCC)**		**Number of Interface (NI)**		**Number of Operations (NO)**	
	non-AO	*AO*	*non-AO*	*AO*	*non-AO*	*AO*
GUI_Manager	4	1	2	2	**17**	**14**
Distribution_Manager	**3**	**2**	**3**	**1**	**34**	**16**
Concurrency_Manager	1	1	1	2	2	4
Business_Manager	**4**	**1**	**11**	**9**	**64**	**57**
Transaction_Manager	2	2	2	2	5	5
Data_Manager	2	2	7	7	40	40

In our second case study which was about the MobiGrid architecture, the Afferent Coupling Between Components (AC) and Efferent Coupling Between Components (EC) metrics showed values 50% higher for the non-AO architecture. This difference was due to the bidirectional coupling between several components. The concern-driven metrics highlighted that this bidirectional coupling was caused by the propagation of events related to the Mobility concern over several components. In the AO architecture the Mobility concern was totally modularized within an aspectual component, eliminating, as a consequence, the bidirectional coupling between the other components.

5 Concluding Remarks

Up to date, to the best of our knowledge no concern-oriented metrics have been developed to evaluate software architecture modularity. There are only initial works on concern-based metrics [11, 24]. Di Stefano et al [11] propose a metric for coupling between concerns based on the coupling between the classes that implement the concern. Eaddy et al [24] propose two metrics: degree of scattering which measures the distribution of a concern’s implementation; and degree of focus which measures the degree to which a class’s implementation relates to multiple concerns. Nevertheless both works are concerned with assessing the separation of concerns only at the implementation level. The works concerning quantitative architecture assessment are all about the definition of metrics based on conventional abstractions and, therefore, suffer from the limitations previously discussed in this paper. For instance, Briand et at [1] define architecture metrics for coupling, cohesion and visibility based on the module abstraction. Martin [16] defines coupling metrics resting on the object-oriented package abstraction.

Modularity occupies a pivotal position in the design of good system architectures. Yet the task of considering the multi-dimensional facets of modularity remains a deep challenge to architects. Building modular architectures is a challenging task mainly because the architects need to reason and make decisions with respect to a number of crosscutting architectural concerns. In this way, the main contribution of our work is to promote concern as an explicit architecture measurement abstraction.

This paper proposed a measurement framework consisting the following: (i) a suite of concern-oriented metrics for evaluating architecture modularity, and (ii) a technique for documenting the concerns in the architecture. This paper also systematically discussed limitations of the conventional architecture metrics. In the future, we plan to develop a tool for automating both the architectural concerns documentation and the metrics application. We intend to extend our tool [22], which supports the application of some similar concern-driven metrics at the detailed design and implementation level. We also plan to develop guidelines for the systematic identification of concerns in the architecture and a catalogue of concerns which are candidate to be considered in the architecture evaluation. Moreover, we plan to undertake new case studies to: (i) investigate how our framework can support the detection of unstable architecture elements while making changes in the systems (Section 2.3), (ii) analyse how the use of our framework to control the modularization of concerns at the architecture-level impacts in the implementation artefacts, since they simply represent projections of the architecture design, and (iii) validate the metrics in terms of their impact on external quality attributes, such as maintainability and reusability.

Acknowledgements. This work is partially supported by European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe). Claudio is supported by CAPES-Brazil under Grant No. 3651/05-3.

References

[1] Briand, L., Morasca, S., Basili, V.: Measuring and Assessing Maintainability at the End of High Level Design. In: Proc. IEEE Conf. Software Maintenance (1993)

[2] Buschmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley, Chichester (1996)

[3] Dobrica, L., Niemela, E.: A Survey on Software Architecture Analysis Methods. IEEE Trans. on Soft. Eng. 28(7), 638–653 (2002)

[4] Garcia, A., et al.: On the Modular Representation of Architectural Aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, Springer, Heidelberg (2006)

[5] Garcia, A., Lucena, C.: Taming Heterogeneous Agent Architectures with Aspects. Comm. of the ACM, July 2006 (accepted to appear)

[6] Garcia, A., et al.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: AOSD'05, pp. 3–14 (2005)

[7] Concern-Driven Measurement Framework for Assessing Architecture Modularity. URL: http://www.lancs.ac.uk/postgrad/figueire/co_metrics

[8] Garlan, D., et al.: ACME: An Architecture Description Interchange Language. In: Proc. CASCON'97 (November 1997)

[9] Lindvall, M., et al.: Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture. In: Proc. of the Intl. Symposium on Software Metrics, USA, p. 77 (2002)

[10] Lung, C., Kalaichelvan, K.: An Approach to Quantitative Software Architecture Sensitivity Analysis. In: Proc. of the Int'l Conf. on SW Eng & Knowledge Eng., pp. 185–192 (1998)

[11] Di Stefano, A., et al.: Metrics for Evaluating Concern Separation and Composition. In: SAC 2005, pp. 1381–1382. ACM Press, New York (2005)

[12] Sant'Anna, C., et al.: On the Quantitative Assessment of Modular Multi-Agent System Architectures. NetObjectDays (MASSA) (2006)

[13] Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs (1996)

[14] Kulesza, U., et al.: Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In: Proc. of ICSM'06, Philadelphia, USA (September 2006)

[15] Filman, R., et al. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Reading (2005)

[16] Martin, R.: Stability, C++ Report (February 1997)

[17] Soares, S., et al.: Implementing Distribution and Persistence Aspects with AspectJ. In: Proc. of OOPSLA'02, pp. 174–190 (2002)

[18] Robillard, M., Murphy, G.: Concern Graphs: Finding and describing concerns using structural program dependencies. In: Proc. of the ICSE'2002, pp. 406–416 (May 2002)

[19] Gamma, E., et al.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1996)

[20] Barbosa, R., Goldman, A.: MobiGrid. In: Karmouch, A., Korba, L., Madeira, E.R.M. (eds.) MATA 2004. LNCS, vol. 3284, pp. 147–157. Springer, Heidelberg (2004)

[21] Bass, L., et al.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading (2003)

[22] Figueiredo, E., Garcia, A., Lucena, C.: AJATO: an AspectJ Assessment Tool. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, Springer, Heidelberg (2006)

[23] Filho, F., et al.: Exceptions and Aspects: The Devil is in the Details. In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, Springer, Heidelberg (2006)

[24] Eaddy, M., et al.: Identifying, Assigning, and Quantifying Crosscutting Concerns. In: ACoM.07 at ICSE'07. 1st Workshop on Assessment of Contemporary Modularization Techniques (2007)

Lightweight Web Services for High Performace Computing*

Adrián Santos, Francisco Almeida, and Vicente Blanco

Dpto. Estadística, .I.O. y Computación Universidad de La Laguna, Spain
asmarre@ull.es

Abstract. Web Services-based technologies have emerged as a technological alternative for computational web portals. Facilitating access to distributed resources through web interfaces while simultaneously ensuring security is one of the main goals in most of the currently existing manifold tools and frameworks. OpenCF, the Open Source Computational Framework that we have developed, shares these objectives and adds others, like enforced portability, genericity, modularity and compatibility with a wide range of High Performance Computing Systems. OpenCF has been implemented using lightweight technologies (Apache + PHP), resulting in a robust framework ready to run out of the box that is compatible with standard security requirements.

1 Introduction

A widespread drawback among many scientists who wish to use the potential computational power provided by High Performance Computer Systems (HPCS) is the significant barrier (technological and learning) these users face when trying to access such services.

Parallel machines usually run UNIX-based Operating Systems which users access through terminal-based connections. Once the code is compiled by the user, it is run as a batch code through the queue system. Typical researchers, who at one time may have been familiar with the code or library used to run experiments in their field of expertise, are having to confront their inexperience when using terminals and UNIX/Linux commands. An additional source of complexity is introduced by queue systems that vary from system to system and which use their own commands and manipulation rules. The type of user we are dealing with is not particularly interested in learning new tools or knowing the details of parallel computing, and yet the effort required in these cases from researchers who are only interested in faster processing times is still high.

Solution strategies founded on Client/Server-based applications have emerged as alternatives to overcome this barrier. A friendly graphical interface enables access to non-local resources where parallel codes can be executed remotely in geographically distributed parallel machines, or where heterogeneous devices

* This work has been partially supported by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2005-09037-C02).

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 225–236, 2007.

can interact with each other to solve a problem in parallel. The approach allows the use of the system by a larger number of users, including those without any knowledge of sequential or parallel programming. Details of the parallel architecture are transparent to the user and parallel codes can be efficiently executed in the parallel system. A paradigmatic example of this approach is found in [1] or in many other Client/Server applications that even use web interfaces in the clients [2,3,4,5].

As Web Services (WS) progress toward an industry standard, they have attracted the attention of the parallel computing community and proven to be a technological alternative for implementing this kind of system. WS technology eases the implementation of a web-accessed computing system. It also makes for better interoperability between the different systems and tools that comprise a High Performance Computing environment.

The Open Computational Framework (OpenCF) presented here uses WS technologies to achieve these objectives with a lightweight and portable implementation that is compatible with the security requirements of many HPCS sites. OpenCF is freely downloadable from[6], licensed under GPL, and has been successfully used by the Pelican project [7], from where some snapshots have been extracted for illustrative purposes in this paper.

This paper is structured as follows. In section 2 we introduce the background on Web Services for computing environments and the main motivation for this work. Section 3 describes the software architecture and technology used in the modular design of the OpenCF system. Several security-related issues involving WS and HPCS are discussed in section 4 where a description of our implementation in OpenCF can be found. Finally in section 5 we present some concluding remarks and future lines of work.

2 Motivation and Related Work

Web Service (WS) technology facilitates the development of remote access tools which have the ability to communicate through any common firewall security measure without requiring changes to the firewall filtering rules. The solutions presented vary from static ad-hoc WS for specific applications on specific parallel machines [8], to more general and complete solutions.

For example, Gateway [9,10,11] implements a WS portal based on Java and CORBA technologies. In the GridSam [12] project we can find a WS infrastructure that is also implemented using Java, and includes JSDL descriptions for job submissions and a command-line interpreter for access to the GridSAM system. OpenDSP [13] (DRMAA Service Provider) is an open architecture implementation of SOAP Web Service for multi-user access and policy-based job control routines by various Distributed Resource Management (DRM) systems. This API follows the DRMAA [14] specification for the submission and control of jobs to one or more DRM systems. The scope of this specification is the high-level functionality necessary for an application to consign a job to a DRM system, including common operations for jobs like termination or suspension.

In these environments the user is provided with web interfaces or other software infrastructures that supply easy and secure access to several resources, including different architectures and applications. However, the use of frameworks based on Java (JavaBeans, JavaServlets, ...) and CORBA is sometimes criticized for delivering weighted solutions.

The Open Computational Framework (OpenCF) that we present here may be enclosed in this group of projects that provide generic solutions using Web Services-based technologies. While ideas, concepts and goals are shared by these projects, in practice, the differences between them are highlighted by their specific implementations. In our case, the OpenCF is based on PHP technology, providing a lightweight and portable implementation.

Although in many of the cases the development effort has been technologically impressive and the demand for this kind of tool is still high, none of the generic frameworks based on computational web services has achieved enough popularity in the wider parallel programming community to be accepted as a standard framework enjoying widespread use. In many cases they have not been used beyond the institutions where they were developed. Some of the reasons for this are as follows:

- The technology used by itself can be a handicap when the effort demanded from the system managers is high.
- The compatibility with the security policies and resource managers of the HPCS sites.
- Many of the solutions provided are non-standard, non-generic (based on specific technologies), non-portable and non-open source.

WithOpenCFweintendtobuildagenericOpenSourceComputationalFramework based on WS technologies implemented according to the W3C (World Wide Web Consortium) recommendations for WS development [15]:

- OpenCF is compact and highly efficient since it is based on a modular design that has been implemented using the HTTP Apache server, a set of PHP libraries and a Perl script. This combination results in a lightweight package with low resource requirements.
- The system is easy to install, requiring from the system manager no more knowledge than that needed to install a web mail application.
- The genericity is enforced through the use of portable technologies that are compatible with the resource management systems running on most parallel systems, and the security requirements have been implemented in such a way that they are independent and compatible with the security measures of HPCS sites. That means that OpenCF, out of the box and without any modifications, meets the security requirements of HPCS sites.
- The flexibility to enlarge the service with new computational proposals was an important issue considered in our design strategy.

Although we are not using new concepts, our main design effort has been to bring all these ideas together and make them work as a successful tool that we hope will be of great use to the scientific community.

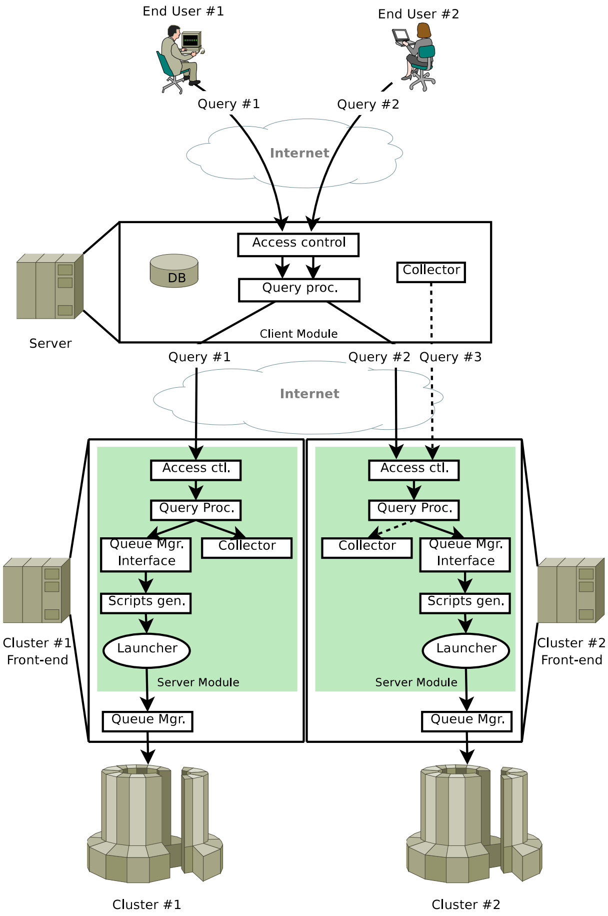


Fig. 1. The OpenCF architecture

3 The OpenCF Architecture

For OpenCF, we identified the following core services to be implemented in the portal: secure identification and authorization for users and inter-communication services, information services for accessing descriptions of available host computers, applications and users, job submission and monitoring through the queue system, file transfer, and facilities for user and resource management.

The above services are typically implemented in many computational web portals; however, we also list some of the design features that are not always included in this kind of tool and which, in our opinion, are required to successfully implement a web computing framework:

- Modular design, whereby modules can be easily added, replaced, updated or independently used with minimum development and management effort.

- Genericity so that the requirements of as many users, applications and HPCS as possible can be met.
- Portability.
- The use of standard technology so as to reach a widespread community.
- Independence from hardware requirements.
- Lightweight.
- Open Source as a requirement to allow the project to be enhanced in the future with new ideas and developments.
- Ease of use and installation both for end users and system managers.
- Ready to use.
- No modifications required in the original source code by the end user.

Figure 1 depicts the OpenCF software architecture. In keeping with a modular design, two modules make up the OpenCF package, namely, the server (side) module and the client (side) module. Modules can be independently extended or even replaced to provide new functions without altering the other system components. The client and the server implement the three lower-level layers of a WS stack: Service Description, XML Messaging, and Service Transport. The fourth level, Service Discovery, has not been implemented for security reasons. Thus, system managers still control the client services accessing the clusters via traditional authentication techniques.

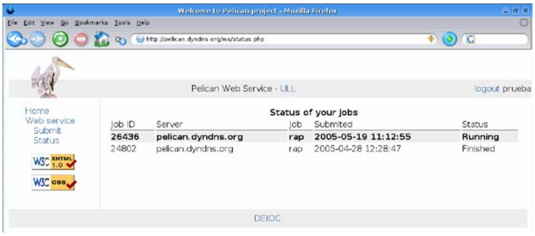
The client provides an interface for the end user and translates the requests into queries for the servers. The server receives queries from authenticated clients and transforms them into jobs for the queue manager. These modules, in turn, are also modularized. `Access Control`, `Query Processors` and `Collector` components can be found on both the server and client sides. The client also holds a database to better manage the information generated by the system. The server includes elements for the scripting and launching of jobs under the queue system.

The following subsections describe the functionalities and technology used in each of the aforementioned modules, except for access control modules and security issues, which will be detailed in a separate section.

3.1 The Client

The client is the interface between the end user and the system. Users are registered in the system through a form. Some of the information requested is used for security purposes, while the rest is needed for job management. This information is stored in the client's database. Next is a listing of the client module submodules.

- The client `DataBase` stores information on users, servers, jobs, input/output files, etc. It has been implemented as a relational MySQL database and it is accessed through PHP scripts.
- The client `Query Processor` consists of a web interface through which the user can access lists of available applications. Each entry in the lists shows a brief description of the routine. Tasks are grouped according to the servers


Welcome to Pelican project - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://pelican.dyndns.org/ws/status.php
logout prueba
Home
Web service
Submit
Status
W3C XHTML 1.0
W3C CSS
DEIOC

Pelican Web Service - ULL

Status of your jobs

Job ID	Server	Job	Submited	Status
26436	**pelican.dyndns.org**	**rap**	**2005-05-19 11:12:55**	**Running**
24802	pelican.dyndns.org	rap	2005-04-28 12:28:47	Finished

Fig. 2. Job status under the Pelican project

supporting them. When execution of a routine is requested, the target platform is implicitly selected. An XHTML form for inputting parameters is dynamically generated according to the job description.

- The client `Collector` manages the job's server-generated output. The service notifies the user via e-mail when the job is finished. The state of the jobs submitted for execution can also be checked through a web interface (see Fig. 2), and the results stored into disk files. These files are periodically downloaded by the `Collector` according to the list of pending jobs and can be accessed via the web. Once the end user collects the output files, they can be removed from the client module and, optionally, a copy of the files can be left so any other user can freely access them in the future.

  Different input/output interfaces can be added in the future to provide the system with new functionalities. Automatic server allocation in terms of the resources available at the servers is also left for future development.

3.2 The Server

The server manages all job-related issues, making them available at the service and controlling their state and execution. When Apache catches a new query from the client, it allocates a new independent execution thread to create a new instance of the server module.

- The `Query Processor` module consists of a set of PHP scripts that is responsible for distributing the job among the different components. Queries addressed for the computational system are dispatched to the Queue Manager Interface, and the rest of the queries are served by the `Query Processor`. The web service is also generated and served by this module. The service description document (WSDL) is automatically updated by the NuSOAP PHP class. This class also handles the SOAP messaging encapsulation of the packets.
- The `Queue Manager Interface` handles the interaction with the HPCS Queue System. The server needs to know how a job will be executed and how to query the status of a job being executed on the server supporting it. To do so, two PHP class methods, the class `OpenCFJob`, have to be overwritten. These methods enable the job to be executed (under the queue system) and

Listing 1.1. The main routine for the Hello World job

```
void hello(int greetings_num, char *name);

int main(int argc, char *argv[]) {
   int num;                        /* Greetings */
   char *name;                     /* Name */

   if (argc != 3) {
      printf("Usage:\n");
      printf("\thello <GREETINGS_NUM> <NAME>\n");
      return 1;
   }
   num = atoi(argv[1]);
   name = argv[2];
   hello(num, name);
   return 0;
}
```

the job status to be checked. In addition, an XML description of each available routine is needed to specify the job. As an example, listing 1.2 shows the XML description for a simple "Hello World" code. The tag <name> holds a representative identifier for the routine and the tag <binary> is the path to the executable file. Then, the problem description and the routine arguments, along with their data types, are introduced. Once the user submits a job request, the server executes the associated binary code with the arguments supplied by the user. The binary is a pre-compiled file of the code in listing 1.1. Thus, new services can be easily incorporated into the service just by adding the XML description file with the pre-compiled code to the server.

– The Scripts Generator produces the scripts needed for the job execution under the various queue systems. The Script Generator is composed of a set of templates plus a processing engine to create the script. A different template is needed for each of the queue managers supported. The template is instantiated into a functional script by the processing engine by the substitution of a fixed set of fields. These fields are obtained from the input data arguments for the job, from the XML job description document, and from the user data stored by the client DataBase. Currently the Scripts Generator uses the template library Smarty[16], a lightweight PHP library found in any UNIX/Linux system. Research is ongoing into making this module compatible with the Job Submission Description Language (JSDL) GridForum proposal as an alternative to the XML document currently used to describe a job.
– The Launcher is the interface between OpenCF and the operating system; it forks the process to be executed, returns its identification and unlocks the thread handling the client query. The implementation is Perl-based to be independent from the architecture. In future versions of the OpenCF accounting system, this module will be responsible for collecting and reporting on individual and group use of system resources.

Listing 1.2. Describing a Problem

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="job.xsl"?>

<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNameSpaceSchemaLocation="job.xsd">
        <name>Hello World</name>
        <service_name>hello</service_name>
        <binary>bin/hello</binary>
        <description>Simple Hello World application</description>
        <argument type="integer">
                <name>greetings_num</name>
                <sdesc>Number of greetings</sdesc>
                <ldesc>The number of greetings to output.</ldesc>
        </argument>
        <argument type="string">
                <name>name</name>
                <sdesc>Person to greeting</sdesc>
                <ldesc>The name of the person to greet.</ldesc>
        </argument>
</job>
```

– The `Collector` is the client interface which delivers the output data produced by an executed job. Once a job has finished, the queue system automatically sends an e-mail to the user, and moves the output data files to a temporary directory until they are downloaded from the client `Collector`. The server `Collector`, implemented as a PHP script, cleans up files from the temporary directories on the server once they have been safely downloaded from the client.

4 Security in OpenCF

Typically, in web services technologies, two levels of security are required: client-server and server-backend transactions must be authenticated, authorized, and encrypted.

In our approach, since the architecture in OpenCF has two independent modules, a WS consumer, the client module, and a WS provider, the server module, a third level of security must be introduced, user-client security, for the transactions between the end users and the client module.

Furthermore, in our particular case, the WS security requirements must be compatible with those of the computational resource. HPCS services are usually restricted-access resources due to their high complexity and capacity, especially considering many of these systems are used in the data centers of government institutions and companies with strong security controls.

The common requirements are:

- User authentication to access the system through the (username, password) pair.
- The authorization to control access to the resources, usually based on services provided by the operating system and the queue managers.

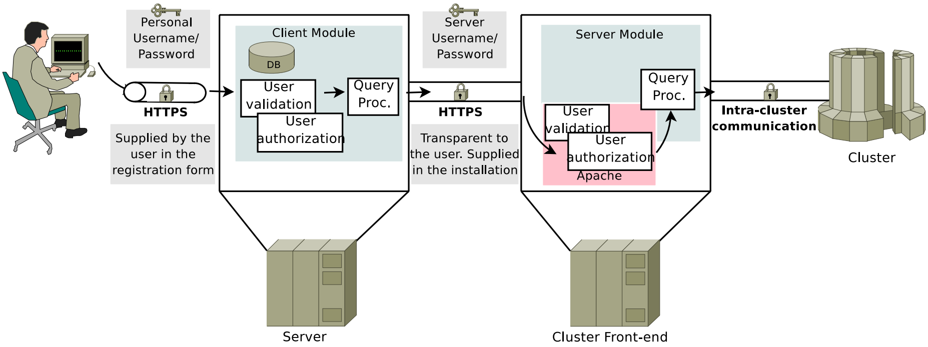

Personal Username/ Password
HTTPS
Supplied by the user in the registration form
Client Module
DB
User validation
User authorization
Query Proc.
Server Username/ Password
HTTPS
Transparent to the user. Supplied in the installation
Server Module
User validation
User authorization
Apache
Query Proc.
Intra-cluster communication
Cluster
Server
Cluster Front-end

Fig. 3. The OpenCF security implementation

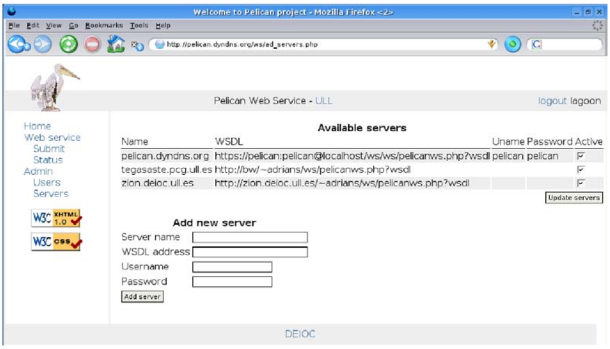
– The privacy typically provided through secure connections like SSH for remote transactions of the end users.

Higher levels of security can be found in computational sites where more specific access controls are needed, for example using Kerberos.

The WSS[17] document provides a standard mechanism for authenticating and authorizing the sender of a SOAP message, and also for encrypting the message. Some of the supported technologies are: Kerberos (both for the authentication and the encryption), X.509 certificate, SAML (Security Assertion Markup Language, developed by OASIS for the interchange of data for authentication and authorization between security domains using XML documents), SSL, etc.

The OpenCF approach to security (fig. 3) is based on implementations of the standard technologies present in time-tested scientific and commercial applications. We adopt simple solutions to comply with the above-stated security requirements. The end users register at the client web page by using a form. The information required consists of an e-mail address where results and system notifications are sent, the username and password to be authenticated at the client and some information relative to the user's organization to help the system manager decide whether the registration was successful or not. Once this information has been submitted, the account will not be activated until the organization, through a privileged user (the system manager), decides whether the request is accepted or not. At that time the user is notified of the resolution. The client database module stores information about the users registered. The password is encrypted for security reasons and a PHP module is used to authenticate end users when logging into the system. Users with management privileges can be added. These users may access the management options at the client (fig. 4) and are responsible for adding and removing users and servers.

Clients are also authenticated at the server using a username and password provided by the system manager of the HPCS where the server is installed. Apache enforces authentication between client and server through the *mod_auth* module that controls the access to resources through the (username, password)


Welcome to Pelican project - Mozilla Firefox <2>
File Edit View Go Bookmarks Tools Help
http://pelican.dyndns.org/ws/ad_servers.php
Pelican Web Service - ULL
logout lagoon
Home
Web service
Submit
Status
Admin
Users
Servers
W3C XHTML 1.0
W3C CSS
Update servers
Add server
DEIOC

Available servers

Name	WSDL	Uname	Password	Active
pelican.dyndns.org	https://pelican:pelican@localhost/ws/ws/pelicanws.php?wsdl	pelican	pelican	☑
tegasaste.pcg.ull.es	http://bw/~adrians/ws/pelicanws.php?wsdl			☑
zion.deioc.ull.es	http://zion.deioc.ull.es/~adrians/ws/pelicanws.php?wsdl			☑

Add new server

Server name ______

WSDL address ______

Username ______

Password ______

Fig. 4. Servers management option

pair. This pair is generated by the server manager using Apache tools when installing the system. The authentication between the server and the cluster is enforced through a `cluster access account`. This approach is compatible with the recommendation of the WSS committee.

To make OpenCF compatible with the security requirements on the cluster where the server will be running, OpenCF security measures are implemented using the security technologies of the cluster system. For instance, in most cases, the `cluster access account` is an unprivileged user created for that purpose.

Once the end user has been authenticated she may access all the services provided by the client. The authenticated client is also authorized to launch all the services provided by the server.

The system manager at the HPCS where the server is installed authorizes the `cluster access account` to submit the jobs, launched by the WS, to the queue manager according to her access control policy.

In keeping with the OASIS recommendation, privacy between end user-client and client-server is enforced through the use of HTTPS to encrypt messages. Privacy between the server and the cluster is delegated to the cluster's privacy facilities so as to comply with compatibility requirements inside the cluster.

Finally, we validate the input data provided by the end user in the form. Appropriate treatment must be performed to avoid special characters that could be used to trick the parser and execute malicious code.

5 Conclusion and Future Work

We have presented an open source computational framework based on web services technologies. An important design consideration was the ability to provide the community with a modular, lightweight, standard tool ready to run out of the box. Security in OpenCF was implemented according to established standards which are compatible with the security policies of many HPCS sites. A valuable feature is the simplicity of the approach.

The robustness of the OpenCF was successfully tested and validated in the Pelican[7] project. This project is a comprehensive collection of parallel routines based in the GNU Scientific Library (GSL, [18]) tailored to parallel architectures. OpenCF performed very well in that scenario. Some shortcomings have been detected during the production phase of the project. For example, the limits in the size of the attached files, these limits have to be raised for the successful use of OpenCF when required. One source of difficulties was the addition to the WS of all these routines from Pelican. Currently, writing the description of a service is made by hand, that is not a problem for few routines but typically hundred of service routines will be demanded. Currently, we are developing automatic mechanisms to fulfill this objective.

For the near future, we also intend to extend OpenCF with more functionalities. We are planning to add the discovery layer of the Web Service stack, that will certainly provide new functionalities in terms of interoperability, as well as it will present new challenges in terms of security.

References

1. NetSolve project web site, `http://icl.cs.utk.edu/netsolve`
2. Klotz, G.A., Harvey, N., Stacey, D.A.: Connecting researchers to hpcs through web services. In: HPCS, p. 14. IEEE Computer Society, Los Alamitos (2006)
3. Thomas, M., Mock, S., Boisseau, J.: Development of web toolkits for computational science portals: The npaci hotpage. hpdc 00, 308 (2000)
4. Petcu, D.: Between web and grid-based mathematical services. iccgi 0, 41 (2006)
5. Yang, X., Hayes, M., Usher, A., Spivack, M.: Developing web services in a computational grid environment. scc 00, 600–603 (2004)
6. OpenCF project webpage, `http://opencf.pcg.ull.es`
7. Pelican project webpage, `http://pelican.pcg.ull.es`
8. e-HTPX E–Science Resource for High Throughput Protein Crystallography, `http://clyde.dl.ac.uk/e-htpx/index.htm`
9. Pierce, M.E., Youn, C.H., Fox, G.: The gateway computational web portal. Concurrency and Computation: Practice and Experience 14(13-15), 1411–1426 (2002)
10. Pierce, M.E., Youn, C.H., Fox, G.: The gateway computational web portal: Developing web services for high performance computing. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science - ICCS 2002. LNCS, vol. 2329, pp. 503–512. Springer, Heidelberg (2002)
11. Youn, C.: Web Service Based Architecture in Computational Web Portals. PhD thesis, Graduate School of Syracuse University (2003)
12. GridSAM project web site, `http://gridsam.sourceforge.net/`
13. OpenDSP: DRMAA Service Provider open implementation, `http://sourceforge.net/projects/opendsp`
14. DRMAA: Distributed Resource Management Application API, `http://www.drmaa.org/`
15. World wide web consortium, `http://www.w3.org/`
16. Smarty template engine, `http://smarty.php.net/`

17. OASIS Web Services Security (WSS) TC, http://www.oasis-open.org/committees/wss/
18. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M., Rossi, F.: GNU scientific library reference manual. Ed. 1.2, for GSL Version 1.2 (2002)

The Art and Science of Software Architecture

Alan W. Brown[1] and John A. McDermid[2]

[1] IBM Software Group
Raleigh, NC, USA
awbrown@us.ibm.com
[2] University of York
Heslington, York, UK
John.McDermid@cs.york.ac.uk

Abstract. The past 20 years has seen significant investments in the theory and practice of software architecture. However, architectural deficiencies are frequently cited as a key factor in the shortcomings and failures that lead to unpredictable delivery of complex operational systems. Here, we consider the art and science of software architecture: we explore the current state of software architecture, identify key architectural trends and directions in academia and industry, and highlight some of the architectural research challenges which need to be addressed. The paper proposes an agenda of research activities to be carried out by a partnership between academia and industry. While challenges exist in many domains, for this paper we draw examples from one area of particular concern: safety-critical systems.

Keywords: Software architecture, Software engineering, Systems engineering.

1 Introduction

Experience developing software-intensive solutions in many domains leads one to conclude that a very significant shift is taking place in delivery of complex software systems[1]. This shift is being driven by several convergent factors, including [1, 2]:

- *End user expectations* – End users want information to be available everywhere, on demand, with no downtime.
- *Cost to create solutions* – To be competitive it is essential that IT organizations take advantage of lower labor rates around the world, integrate components from a variety of suppliers, and reuse across product lines and solution families.
- *Auditing and Compliance* – Increased regulations and oversight are placing additional requirements of adoption of well-documented best practices with mandated control points and delivered artifacts to aid auditing.
- *Speed of change* – The fast pace of business change demands that IT systems can be reconfigured quickly as those needs change.

[1] We refer to “complex software systems” as a shorthand for “large-scale development and support of software-intensive solutions in domains such as defense, aerospace, telecommunications, banking, insurance, healthcare, retail, etc.”.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 237 – 256, 2007.

- *Adaptable business platforms* – Today's distributed solutions platforms must allow optimization around current business needs, and support reconfiguration as those needs change.

As IT organizations reexamine their IT systems and rethink their practices and tools, the subject of systems and software architecture is frequently at the core. In fact, we observe more categorically that architecture is pivotal in the development of complex software systems; the Royal Academy of Engineering and British Computer Society (RAE/BCS) report on Complex IT Systems [3], states that:

> *"Systems architecture is one of the most significant technical factors in ensuring project success."*

Unfortunately, as a central element in a project's success, the problems too often associated with delivery of high quality systems on-time and to budget can also be frequently traced to issues of systems and software architecture. As the RAE/BCS report identifies:

> *"At present, definition of architecture is relatively ad hoc and, whilst there is good practice, much more needs to be done to codify, validate and communicate the experience and skills of the best systems architects."*

Such concerns raise a number of deep questions concerning the art of software architecture as practiced and supported in industry, and the science of software architecture as studied and taught in academia. This provides the motivation for our paper; to see how to gain benefit by combining industrial experience with academic work on software architecture.

We start by considering the role of software architecture in the software development process then outline what we see as the "state-of-the-practice" in industry/ commerce and the "state-of-the-art" in academia. To focus this analysis – we believe without loss of generality – we consider the challenges of UK Defence systems, including safety-critical systems. We use this analysis to identify a research agenda – to strengthen industrial/commercial practice, and to focus academic research.

2 State-of-the-Practice in Software Engineering

There is not a "hard" distinction between academic and industrial/commercial research in software architecture – in fact, we shall argue that there is a need for greater links between academic and commercial endeavors – but there is a discernible difference in emphasis. Our aim below is to outline some of the main thrusts in these areas –space does not permit a full exposition of ongoing work, and we focus more on research relevant to safety critical and embedded systems, as this enables us to illustrate some successful industrial-academic collaboration.

2.1 Academic

Taken broadly, the academic view has been to try to bring rigor to architectural definition – either building new approaches or seeking to "strengthen" techniques used in industry. We distinguish here the formal and analytical approaches from those more pragmatic approaches which seek more to underpin the development process.

2.1.1 Analytical Approaches

Widely used software architecture description notations such as UML are often viewed by academic researchers as having weak semantics; some early work on “precise UML” (pUML[2]), sought to provide a sound semantic basis for the notation. In some research programs, the aim was to support analysis of models but in many cases it was just an attempt to remove ambiguity. Arguably such work is bound to be of limited value as the tools effectively define semantics, and the tools evolve faster than the modeling research.

More recently, the focus has been on defining new notations with better defined semantics (see below) or on adapting popular notations to make them analyzable. A common approach has been to identify parts, or aspects of notations, which can be subjected to formal analysis. Perhaps one of the most common has been to use model checking on the state machines in UML see, for example [7], or in StateMate. This sub-problem has been chosen as it is amenable to automated analysis, but it still does not solve the basic problem of semantic ambiguity[3]. Further, there is an issue of scalability; although model checking is becoming more powerful, there are as yet few examples of the being used on (large-scale) real-world problems ([8] is one of the few published examples). Perhaps more critically, such approaches do not address the full expressive power of notations such as UML.

In safety critical systems, where there is a strong motivation to verify programs, the SPARK notation and toolset [9] has become successful [10]. The tools incorporate the long-established principles of program verification by extending a subset of Ada with annotations which allow the expression of pre- and post- conditions. The SPARK Examiner tool is capable of discharging some “routine” proof obligations, e.g. freedom from run-time exceptions, largely automatically, as well as showing correctness against the pre- and post- conditions.

Some work has “lifted” this idea to the architectural level, by defining a state machine subset and adding annotations, e.g. on assumptions of rate of change of external variables when moving between states [11] (this was done on Stateflow, the state machine element of Matlab-Simulink-Stateflow (MSS), which is widely used for modeling control systems). Related work also addressed the rest of the MSS notation, viz control law diagrams, proposing a variant of WP-calculus which deals with differentials (and integrals) as found in control systems [12]. This work has some technical merit, but there are problems of scaling it to complex systems – especially in the interaction of the state machine and control law elements (there is a “state explosion” which hinders automated analysis).

The above discussion only considers the functional properties of programs. Most safety-critical systems are also real-time systems, and it is necessary to be able to demonstrate that they meet (specified) timing properties. This involves both determining the worst case execution time (WCET) of code fragments, and the overall timing properties of the program, which involves analysis of scheduling. There have been mature theories for scheduling for some time, and these have been applied to real-world systems, e.g. [13].

[2] There is still a webpage at www.cs.york.ac.uk/puml/ but the site is now moribund.

[3] Work by Mikk in the late 1990s identified over 100 semantic models of StateMate statecharts.

Some time ago, it was possible to determine WECT analytically (formally), by a combination of program static analysis and instruction counting. Again this work reached a level of maturity which allowed it to be applied on real systems, e.g. [14]. Since the mid 1990s, timing analysis has become harder as processors have been optimized for the best average case performance, and worst case program timing is influenced by the interaction of cache and pipeline with the program data. More recently, therefore, research has moved away from pure analytical techniques and is using a combination of static and dynamic (testing) techniques. Work at Rapita[4] shows that it is still possible, however, to produce techniques which are applicable to industrial scale problems and which can estimate timing properties at the architectural level.

In summary, a major thrust of much academic research which can be viewed as being at the "architectural level" has been focused on the use of formal, analytical, techniques. There have been some successes, both intellectual and practical, but there are also significant limitations on the techniques. The formal analysis of functional properties has not proven to scale effectively. Model checking techniques are advancing in power faster than Moore's law (i.e. algorithms are improving as well as the processing hardware), but dealing with all aspects of large-scale complex control systems remains problematic. Approaches to timing properties have been more successful, in impacting real-world systems, but to cope with modern processors it has been necessary to move away from a purely analytical approach to using a mixture of static and dynamic techniques. This must be considered a success, not a failure, but it highlights the difficulties of taking analytical approaches to large-scale problems.

2.1.2 Pragmatic Research

There is less of a clear-cut distinction between what is industrial research and academic research in the more pragmatic areas – indeed there is effective collaboration between Universities and industry – and we focus here on those developments which are clearly influenced by both an academic and an industrial perspective. We consider two aspects of work: that which expands on notations to make them more relevant to their application domains, and that which focuses on making development processes more rigorous and repeatable.

Languages such as UML, despite being multi-faceted, do not address all the properties of interest for real-time safety critical systems; for example they do not express timing properties within the core language. There have been approaches to dealing with these limitations for some time, e.g. real-time extension of UML [15], but these have not been widely taken up. The reason seems to be mainly that the extensions are not sufficiently rich – and thus do not do enough to enable the techniques to solve all the problems encountered in practice.

Some academic work has tried to address this problem by developing wide-spectrum specification languages, and supporting contracts which enable aspects of systems to be specified and built independently, e.g. AIM [16]. Whilst mainly pragmatic, this work has also embraced formal analysis of failure properties, a key aspect of safety-critical software designs, with a technique known as FPTC [17]. Associated work has also considered the use of automated techniques for model transformation (we defer such issues to section 3.2 below).

[4] See http://www.rapitasystems.com/

Perhaps the most significant aspect of this work has been to influence domain-specific languages such as the Architecture Analysis and Description Language (AADL) which started out as a research program funded by the US Army Avionics Command, and which is now an SAE standard[5]. AADL incorporates contracts and notions of fault propagation which are influenced by AIM and FPTC. Whilst AADL was originally an "independent" notation it is now available as a UML profile.

There are many approaches to improving development processes which build on architectural representations; we consider two: product line research and test automation.

Many systems, e.g. airplane engines, evolve as product lines, i.e. the features and properties of one product bear a strong relationship to those of others in the family – and the relationships can be utilized to simplify the design process. Specifically the common parts can be represented in the architecture – which also has explicit representation of variations (alternatives) and options in the system. These concepts have been studied in academia [18], but are also used in industry [19] and are supported commercially[6]. Further work has considered the process for deriving requirements for product lines and architectural flexibility [20] in the context of embedded control systems (aircraft engine controllers). Generally work in this area is quite mature. It is not as widely adopted as might be expected, but that is largely a socio-technical issue, which is outside the scope of this paper.

Perhaps one of the reasons for limited take-up of such ideas is that the real cost of producing critical systems arises from the verification activities, not the development activities. If code generation costs, say, 10% of the development cost and verification is 50% of the cost, then halving code production time is of little benefit if the verification cost is not also reduced. Some tools, e.g. Reactis[7], have been developed to automate some of the verification tasks when using model-based development, but generally these do not consider issues such as product lines. Some research work, e.g. [21], has addressed product lines but there remains much more to be done.

Interestingly, because of the cost of testing, there have been a number of attempts to substitute formal analysis for testing. One of the most interesting developments is ClawZ which verifies Ada against MSS models [22]. This approach has been used on practical applications, e.g. the Eurofighter Typhoon flying control system.

For these approaches – test automation and formal analysis – there are many challenges, including ensuring that the certification authorities accept their validity. Again, space does not permit a full treatment of these issues, but [23] indicates one way of overcoming these difficulties, by providing a logical argument why the techniques proposed are an effective substitute for those mandated by the standards.

2.1.3 Observations

The work outlined in the two previous sections can (perhaps simplistically) be characterized as "science driven" and "problem driven" (or an engineering approach) respectively.

[5] See http://www.aadl.info/

[6] See, for example http://www.biglever.com/index.html

[7] See http://www.reactive-systems.com

The “science driven” approaches have potential, but often fall a long way short of solving industrial problems. It is likely that this is where radical changes to processes will come but the success rate is likely to be low, as the problems of scaling the “science” to industrial scale problems are very great, and perhaps need more investment than can be found in academic projects.

The “problem driven” approaches have often involved interaction between academia and industry – focusing academic research on issues which cause difficulty in practice. Whilst this work may not always produce full solutions it often ameliorates the problems, and produces useful incremental improvements to processes.

Both forms of research are needed – see the later discussions for our views on the balance.

2.2 Industrial/Commercial

Architectural concerns are important to all industrial systems. In any development project there is significant focus on the architecture as a central artifact. However, practical considerations require efficient techniques and tools that provide clear value to the project. All too frequently this leads to shortcuts being taken that have consequences on the long term maintenance and evolution of the system.

Here, we explore several areas of industrial software architecture as practiced today, discuss current practices in use, and highlight several key gaps and challenges.

2.2.1 Architectural Design

Today, a majority of software developers still take a code-focused approach to development, and do not use separately defined abstract (architectural) models at all. They rely almost entirely on the code they write, and they express their model of the system they are building directly in a third-generation programming language (3GL) such as Java, C++, or C# within an Integrated Development Environment (IDE), e.g. the IBM Rational Application Developer, Eclipse, or Microsoft VisualStudio. Any separate modeling of architectural designs is informal and intuitive, and lives on whiteboards, in Microsoft PowerPoint slides, or in the developers’ heads. While this may be adequate for individuals and very small teams, this approach makes it difficult to understand key characteristics of the system among the details of the implementation of the business logic. Furthermore, it becomes much more difficult to manage the evolution of these solutions as their scale and complexity increases, as the system evolves over time, or when the original members of the design team are not directly accessible to the team maintaining the system.

The UML is the most frequently used language for visualizing static and dynamic aspects of software-intensive systems [5]. Users of UML for architectural design are supported by well-established methods that offer a set of best practices for creating, evolving, and refining models described in UML. One of the most well-known is the IBM Rational Unified Process (RUP). The RUP describes a development process that helps organizations successfully apply a Model-driven Development (MDD) approach [24].

The UML is one element of a broader initiative aimed at encouraging a model-driven approach. This is most clearly seen in OMG’s Model Driven Architecture

(MDA) approach, a set of technologies intended to provide an open, vendor-neutral approach to the challenge of business and technology change [26]. Based upon OMG's established standards such as the Meta-Object Facility (MOF), UML, and XMI, the MDA separates business and application logic from underlying platform technology. The goal of MDA is to offer a conceptual framework for using models and applying transformations between models as part of controlled, efficient software development process [27]. Several commercial and open source software products claim support for part or all of an MDA approach, and are used in many architectural domains most notably in real-time and embedded systems.[8]

In spite of these improvements in model-driven development support, many developers use the UML notation informally as a way to "sketch" the design of new and existing systems [28]. Architectural design can then occur through use of domain-specific languages (DSLs) that embed many architectural patterns and assumptions into the notation and its transformation into a solution specific to the domain. Many such DSLs are available focusing on business domains such as banking, automotive, and telecommunications systems.

The approach to creating and applying DSLs has received additional attention recently as it is one of the cornerstones of Microsoft's Software Factories approach [29]. A Software Factory is a collection of technologies that introduces product line thinking around an application architecture that is defined and refined through domain specific languages. Tooling for this approach is part of Microsoft's latest releases of its VisualStudio product line.

2.2.2 Architectural Analysis

State-of-the-art transformation techniques used in MDA generally cannot be "steered" by dependability issues, and have not been widely applied to architectural models with dependability attributes. Integrating MDA with mechanisms for building dependable systems requires deep and applied knowledge of dependability as well as MDA standards, tools, and techniques.

Models of enterprise architecture (EA) are rarely used to develop assets used downstream. There are several reasons for this. Downstream assets (such as code, documentation for review, and deployment models) are difficult to derive from models using the current state-of-the-art transformation tools, which are typically targeted at single diagrams. An EA model typically provides information spanning several meta-models, and most transformation tools are applicable to a single source meta-model. Deriving downstream assets from EA models may be more feasible – and demonstrably more useful – with mechanisms for integrating different views (from different meta-models) – the results of which could then be applied to transformation. Another difficulty encountered is dealing with non-functional attributes, such as quality-of-service constraints. While mechanisms like the UML Quality-of-Service (QoS) profile can help to enrich EA models with QoS and dependability characteristics, current generation transformation and integration tools must be made aware of these characteristics during the generation and integration process.

[8] See www.omg.org/mda for more details and examples.

Automated architectural analysis techniques often have to be tailored specifically to modeling languages and tool infrastructure; moreover, adapting existing analysis implementations (e.g., fault and failure analysis, timing analysis) to specific architectural modeling languages and tools often requires much repeated work. As a result, much of the currently practiced architectural analysis remains informal and based on inspection techniques supported by architectural heuristics gathered from experiences across several domains.

2.2.3 Architectural Frameworks

Support for a consistent approach to software architecture requires a set of guidelines that identify the key artifacts to be delivered, and constrains the method by which those artifacts are produced. For many organizations, a primary objective is to produce a standardized blueprint describing a complex systems architecture so that decision-makers can then use this report to compare the architecture of alternative system designs and manage the evolution of existing systems. The blueprint must describe a system's architecture well enough to enable detailed analysis, justify procurement to the project's sponsors, and support ongoing management of the in-flight project.

To assist in this, several organizations have formalized such guidelines in the form of a so-called "architectural framework". These guidelines typically are specialized to a specific technology or business domain (e.g., defense systems, or telecommunications systems), or else promote specific architectural views of the system to highlight particular forms of communication and analysis (e.g., business/IT communication, or operational systems management). In the first category we have efforts such as the US Department of Defense Architectural Framework (DODAF) and the UK Ministry of Defence Architectural Framework (MODAF). In the second category we have efforts such as the Zachmann Framework and The Open Group Architectural Framework (TOGAF).

For illustrative purposes, we shall consider MODAF[9]. The MODAF V1.0 set of baseline documentation was published in 2005 [30]. Many of the MOD stakeholders have started to adopt MODAF. There have been a number of notable areas of successful adoption and several MOD organizations have successfully modeled aspects of their architecture using MODAF (e.g., The Director of Equipment Capability (DEC) Command, Control and Information Infrastructure (CC&II), The DEC Intelligence Surveillance Target Acquisition and Reconnaissance (ISTAR), and The Logistics Coherence Information Architecture (LCIA) within the Defence Logistics Organisation (DLO[10])). In addition, many MoD Invitation to Tender (ITT) documents have required MODAF views to be produced as part of the technical proposal response. Consequently, a number of the tool vendors have developed MODAF specific configurations of their modeling tools, including Salamander, Telelogic, Troux Technologies, Artisan Software and IBM.

However, in spite of this success, the use of architectural frameworks such as MODAF is inconsistent and sporadic. For example, MODAF as a standard is not

[9] We believe that to a large degree our experience with other architectural frameworks mirrors the status of MODAF.

[10] The MoD has recently been reorganised, and the DLO is now part of Defence Engineering and Support (DE&S).

completely mature. The MODAF M3 Meta Model is in advance and inconsistent with the MODAF Technical Handbook [31]. Further, initial use and adoption of MODAF has identified a number of areas that need simplification or clarification. In particular, the MODAF Tool Certification Plan [32] needs to be both extended and implemented. Most importantly, despite the fact that different MOD stakeholders have started to model using MODAF, very little architectural analysis is being performed using these models; it is used simply as a documentation approach for architectural designs.

2.2.4 Architectural Styles

Much practical work has taken place over the past few years to understand how various repeating patterns of development can be understood, categorized, and used as the basis for future systems development. As a result, a set of architectural styles has emerged that are used by many organizations as the basis for design decisions, and supported by vendors in their commercial tools, methods and infrastructure offerings [33, 34].

In particular, the need to respond to changing business demands with flexible IT solutions has led many businesses to employ Service Oriented Architectures (SOAs). SOA is an architectural style aimed at more directly representing business processes through choreographed sequences of services realized through reusable components [35, 36, 37]. The service design layer (or “service architecture”) is explicitly independent of applications and the computing platforms on which they run. Solutions are designed as assemblies of services in which the description of the assembly is a managed, first-class aspect of the system, and hence amenable to analysis, change, and evolution.

SOAs provide the flexibility to treat elements of business processes and the underlying IT infrastructure as secure, standardized components – typically Web services – that can be reused and combined to address changing business priorities. They enable businesses to address key challenges and pursue significant opportunities quickly and efficiently.

However, as with many such initiatives, the interest in SOA as an architectural style has its challenges. While the application of SOA in commercial business domains is well advanced, issues around performance, reliability, and predictability of such a loosely coupled architectural style remain. Most notably, this is a result of a lack of practical architectural design approaches and verification techniques that address key system properties such as availability, performance, resource usage, timing properties, and so on. Similarly, much of the available commercial infrastructure supporting SOA is largely untested in the kinds of demanding contexts typical of military systems. Some interesting research has been carried out, leading to proof-of-concept demonstrations and prototypes. However, much work remains to see this applied more extensively in commercial practice.

3 A Software Architecture Research Agenda

The breadth of areas of concern to software architecture is daunting. Hence, to be successful, the work in software architecture must focus on key issues which are essential to make progress, and directly bridge the academic-to-commercial gap.

We propose research in software architecture is organized into three areas:

- Management.
- Dependability and Properties.
- Assembly and Integration.

We briefly discuss each area. To be effective, and to enable technology transition, collaborative research between industry and academia needs to produce guidance, such as process guides, and handbooks of best practice, which we generically refer to as Statements of Best Practice.

3.1 Management

Within the area of management, we have identified three research strands. We see these strands as ongoing throughout the life of any software architecture program:

- Acquisition processes.
- Measurement.
- Software development processes.

The acquisition strand is intended to address current acquisition practice to ascertain where it might be changed to better address the challenges of acquiring complex software-intensive equipment, systems and systems-of-systems. This strand provides the greatest potential for early impact. Key research issues include:

- Management of risks and uncertainties associated with the requirements and the technical solution.
- The management of changes to the requirements and / or the technical solution.
- Incremental and evolutionary procurement (closely related to the previous items).
- Through Life Capability Management.
- Estimation of cost and timescale in relation to earned value and other ROI measures.

In order to take effect, any improved acquisition processes will need to be adopted by industry. It is our experience that adoption is difficult to achieve. Accordingly, the research effort should be constructed so as to maximize the chances of successful adoption. In particular,

- The research will be focused on areas where the stakeholders agree that there is a need for change and is willing to change.
- Any changes must be matured, through research and pilot studies, before being rolled out generally.

A corollary is that researchers must seek to engage with the stakeholders to discuss the changes that it is prepared to make; and the research program must be sufficiently flexible to absorb the results of that negotiation, which may alter during the life of the program.

The second priority is measurement: this reflects the importance that we attach to taking an evidence-based approach generally. The measurement strand consists of several activities involving data gathering from projects n industry. This must:

- Provide an improved picture of industry practice, as the basis for additional research activities.
- Assist in the identification and assessment of potential improvements to the processes used within industry.
- Assist in monitoring the effect of any such changes in a well-defined feedback loop.

In addition, the measurement activities must include research into techniques for monitoring project health. This aspect of the research effort will concentrate on metrics that are useful to all stakeholders but which are currently under-researched, such as metrics related to safety or security.

The final strand deals with the processes that are used for software development. Here, we note

- The existing literature is very large. It includes: the CMMI, ISO9001:2000 and TickIT, several methodologies such as RUP, and proprietary processes developed within industry.
- Despite this mountain of advice, many problems on projects arise because of the non-application of known good practice. From the RAE/BCS report on the challenges of complex IT systems [3]:

> *"A significant percentage of IT project failures, perhaps most, could have been avoided using techniques we already know how to apply. For shame, we can do better than this."*

Accordingly, we see the adoption of good practice itself as the central issue and this must be the focus of research. Previous research consists of several studies in the business IT sector, in which researchers have studied software teams in action. This research must be expanded into other domains, determining which practices are readily followed “at the coalface”. The outputs from this strand will consist of improved advice regarding the construction of software development procedures within industry.

3.2 Dependability and Properties

Dependability and properties, especially so-called non-functional requirements (NFRs), are cognate issues which need to be treated together in software architecture, especially in trade-studies and practical architectural analysis.

Currently, individual high-integrity systems (HIS) achieve acceptable levels of safety, albeit at high cost. In TLCM, the crucial issues for HIS are reducing the timescales and costs of procurement and extending capabilities to a full range of properties, including other aspects of dependability, e.g. security. In particular, rapid, low-overhead, assurance techniques are required to demonstrate that a given HIS achieves a specific safety or security level, without the current substantial delays in carrying out labor-intensive evaluation processes. The emphasis of new research should focus on automation of assessment – although any such research will need to draw on other work to address this issue fully.

Particular attention of research in this area should be on Systems-of-systems (SoS) issues, where the problems are much more open-ended. Approaches to safety and security which have been used for HIS don't scale (or don't apply) to SoS. For example, safety and security assessment normally starts by defining the boundary of the system of interest – for SoS the boundary is not known in advance, and may change dynamically, especially in open systems. Conditions of use may vary dynamically (e.g. the changing topology of ad hoc networks of users communicating over different categories of secured links, and changing access rights of those users), and the SoS must remain demonstrably safe/secure. This requires new approaches to dealing with safety and security, with more emphasis on the dynamic management of safety and security issues, i.e. sensing and responding to them at run-time. Further, safety and security are not disjoint – for example compromise of a communications link to a UAV could lead to an unsafe weapon release; new approaches to integrated safety and security analysis are needed. Also, there is a view that all elements in a SoS will require some level of assurance – say SIL 2 in safety terminology; emphasis is needed on developing effective assurance techniques for such levels of integrity.

The scope of the work will be different for the two areas. In HIS, the focus of research should be mainly on improving software engineering economics, for example by developing and justifying "agile high integrity" processes. For SoS, the emphasis of research must be more on identifying means by which dependability can be "engineered in" to products, and dependability can be preserved through dynamic changes in the membership of the SoS. The boundaries are the genuine systems engineering issues, e.g. evaluating the dependability of different system concepts.

3.3 Assembly and Integration

To be effective, architectures need to be multi-faceted and concerned with data flow, control flow, module decomposition, and properties such as throughput, resilience and recovery, and so on. Current architectural design methods are inadequate as they do not deal with the range of properties of interest, allow for the prediction of implementation properties, and so on. For HIS, the Architecture Analysis and Design Language (AADL) and work in the Defense and Aerospace Research Partnership (DARP) form starting points, but much needs to be done e.g. on analysis and prediction of properties, effective interface definition to allow ease of integration, and so on. Issues which research needs to address include:

- Architecture evolution.
- Assigning values (figures of merit) to architectures to enable trade-offs.
- Migration of application architectures between computing platforms.
- Scalability in terms of numbers of nodes, and data volumes.
- Open and modular systems (especially SoS), including interface definition.
- Robustness (resilience), recovery and reconfiguration.
- Definition and control of emergent properties.

In practice, however, existing systems often have a brittle, complex structure that does not easily enable a move to the kinds of flexibility required. The IT drivers found by customers for undertaking such a migration include the need for operational and systems agility, interoperability, reuse, streamlining architectures and technology solutions and leveraging legacy systems and existing capability.

The emergence of techniques and technologies for service-oriented architectures (SOA) are directly aimed at providing a design framework to support this kind of flexibility and agility. There remain many research challenges to their widespread use in embedded systems. Most notably, this is a result of a lack of practical architectural design approaches and verification techniques that address key system properties such as availability, performance, resource usage, timing properties, and so on. Similarly, much of the available commercial infrastructure supporting SOA is largely untested in the kinds of demanding contexts typical of military systems. Some interesting research has been carried out, leading to proof-of-concept demonstrations and prototypes. However, much additional research work remains.

Transformation of IT systems of itself will reduce application maintenance and operational cost. However, the bottom-up only approach carries the risk that the new services will embed old ways of doing business, providing flexible systems, but inflexible enterprises. In order for enterprises to become flexible, parallel organizational transformation needs to take place and the complementary supporting governance embedded. Only in this way can the benefits to Operations can be realized (e.g., efficient global footprints, economies of scale, agility - rapid change, regulatory compliance and process optimization).

Integration is concerned with construction of systems from components – whilst minimizing risk and surprises. Several research activities would be valuable. The concerns are, in many ways, the same as for architecture, but from the viewpoint of validating the properties and behavior of the composed system. Issues include:

- Planning and management of integration, including risk management.
- Architectural evaluation (determining figures of merit).
- Testing strategies, especially for SoS which may be configured in an ad hoc and dynamic fashion, maximising what can be predicted from the minimum of testing.
- Configuration control, and evaluation and assessment of differing system configurations.

4 Discussion

We discuss the role of academia and industry in addressing the art and science of software architecture. Simply stated, we believe software architecture holds the pivotal position in helping:

- Academia to educate the next generation of architects in software engineering.
- Industry to produce more robust, scaleable methods and tools that meet users' needs.
- Academia and industry to form a closer partnership to advance the state-of-the-art in software engineering.

4.1 What Can Academia Do?

Software engineers need to understand key aspects of computer science (CS) fundamentals – the analogy is with the need for understanding of physics in the "physical" engineering disciplines, such as mechanical engineering and electronics. Beyond this, they need knowledge and skills in five areas – albeit in different amounts depending on their role in a project.[11] These are:

- Architecture
- Good practice
- Domain knowledge
- Management
- Soft skills

Architecture: Software is an intellectual artifact – producing software is essentially a "pure design" activity. Thus the core of what software engineers need to know is how to architect software systems, and how to tell good architecture from bad. Often this involves understanding the non-functional properties of the architecture – e.g. will it perform fast enough to process all the data, will it be secure (against anticipated attacks), will it manage hardware failures to preserve system safety, and will it be useable by the general public?

Good Practice: All engineers should use good practice, but this is sadly all too rare in software engineering – according to Fred Brooks "in no other discipline is the gulf between typical practice and best practice so large" [40]. Further, good practice is not static, indeed technology moves apace. However, not all the new technology is useful, or stands the test of time. Thus, software engineers need to understand *principles* so they can assimilate new practices and, to some degree, sort out the genuine advances from the "mere fads".

Domain Knowledge: Software engineers need domain knowledge. Programs have a role in the world – either as the control and monitoring element in some embedded system, e.g. in an aircraft engine controller, or as a key enabler in a business or organization, e.g. providing electronic access to patient health records. Most requirements for software systems are incorrect – at least initially. The users or procurers do not fully understand what they want, and also don't write down what is "obvious" – at least to them. To defend against this, and to produce something useful and useable, software engineers need to understand the application domain to be able to validate and complete requirements. It is not normally possible for software engineers to gain domain knowledge in many, disparate, areas – there is simply too much to understand to be expert in say, car braking system design, and on-line reselling. Software may be ubiquitous, but the domain knowledge is not.

Software engineers must obtain domain knowledge to work effectively – computer science knowledge is necessary, but not sufficient, to work in a given domain. An *automotive software engineer* of our acquaintance informs us that he has 20 books on his shelf which he uses regularly – 8 of these are sources of domain knowledge, e.g. the Diesel Engine Handbook.

[11] These ideas are inspired by the work of David Parnas [38], and explored in more detail by one of the authors [39].

Management: Modern software systems are amongst the most complex artifacts produced by man. Developing them requires teams, which need management. The RAE/BCS study found that management was a key success factor, but also a major problem area, for software projects. Managers need to understand what they are managing – otherwise how can they make informed decisions? However they also need to understand management skills and techniques. Traditional engineering distinguishes *repeat* design – making something very similar to what was produced before – from *novel* design, i.e. producing something which is largely unprecedented. Most software projects involve novel, not repeat, design – thus they undertake work for which there is no precedent. Management strategies for dealing with uncertainty, such as incremental development, and approaches to software (project) risk management are therefore keys to success. Software architecture is a control point for managing risk, and a cornerstone of the measures and metrics appropriate for every successful software-based management technique.

Soft Skills: Software engineers may spend much of their day working with computers – but they need to talk to customers, other engineers designing the embedding system, e.g. an aircraft engine, nurses who might use the medical records system, and so on. They need to write manuals, or on-line help, from the *users*' perspective not an internal (design) perspective. They have to work in teams with other software engineers, to ensure that they produce an effective whole. Thus soft skills are crucial to project success. Success of a project is frequently predicated on how well an architect communicates key qualities and properties of the architecture to the rest of the team, and to the broader project stakeholders.

4.2 What Can Industry Do?

Industry needs to produce collections of tools and methods that are more readily consumable by practicing software engineers. While many valuable technologies are available, they need to be more tightly integrated, open to customization, and focused on specific domains and architectural styles.

A collection of tools, practices, guidance, heuristics, and specialized content (patterns, frameworks, domain models, etc.) must be assembled to support an organization as it designs, develops, deploys, manages, and evolves its solutions. Informally we can refer to this as an "architecture-centric workbench". In practice, we can distill certain characteristics in the approaches and capabilities of these workbenches.

As illustrated in Figure 1, such a workbench typically consists of elements in several categories. Here, for illustration purposes we focus on a workbench optimized for SOA architectural styles of solution in business domains such as insurance and banking. The underlying platform for the workbench is a collection of commercial tools acquired from one or more vendors, and integrated through standard techniques such as shared meta-data, import/export across Application Portability Interfaces (APIs), or use of a common plug-in framework such as the Eclipse technology[41]. In the case of many IBM customers it is the IBM Rational Software Development Platform that provides the core set of technologies [42].

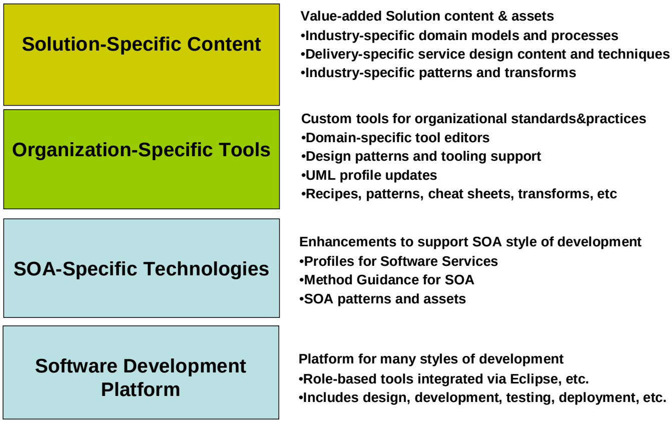

Solution-Specific Content
Value-added Solution content & assets
•Industry-specific domain models and processes
•Delivery-specific service design content and techniques
•Industry-specific patterns and transforms
Organization-Specific Tools
Custom tools for organizational standards&practices
•Domain-specific tool editors
•Design patterns and tooling support
•UML profile updates
•Recipes, patterns, cheat sheets, transforms, etc
SOA-Specific Technologies
Enhancements to support SOA style of development
•Profiles for Software Services
•Method Guidance for SOA
•SOA patterns and assets
Software Development Platform
Platform for many styles of development
•Role-based tools integrated via Eclipse, etc.
•Includes design, development, testing, deployment, etc.

Fig. 1. A Workbench for Service-Oriented Solutions

The core technologies support a wide collection of practices and architectural styles. Extension and customization allows this core to be adapted. In particular, SOA-specific technologies are applied as encoded in patterns and templates, method guides, and profiles for service design that extend the core tooling base.

Then, solution delivery teams within an organization, or external systems integrators and partners, further customize the platform with their own techniques, technologies, patterns, transforms, and so on. These are specific to an organization's ways of working, and relevant to their particular business practices and domain. For example, IBM Global Business Services has defined a set of proprietary practices to provide consistency in the way it delivers services to its clients. It has a collection of SOA-based design techniques, such as the Service-Oriented Modeling and Architecture (SOMA) method [43], that have been distilled from experiences of practitioners on a wide variety of projects. Customized tooling for SOMA has been created as an extension to the IBM Rational Software Development Platform specifically to support those practitioners by automating many of the SOMA techniques.

Finally, domain content is provided to populate the workbench to improve efficiency of delivering solutions, and offer some measure of consistency across solutions in the same domain. Typically, the tools are augmented with domain models and libraries of common patterns in areas such as retail banking, insurance healthcare, and so on. An example of such a domain model is the Insurance Application Architecture (IAA), a detailed set of content models for several aspects of insurance [44].

So this set of technologies, this layering of capabilities, contains tooling, methods, and content. It provides organizations with a domain-specific platform that can be used to deliver service-oriented solutions specific to their business. With some variation, this approach is being used in many organizations to assemble a technology platform appropriate for developing, delivering, and evolving service-oriented solutions.

4.3 What Can Academia/Industry Do Together?

In practice, by working, together academia and industry can add a significant focus on transition into practice, leading to important improvements and up-scaling in the skills, processes, and practices in use. This will be supported in appropriate industry standards and technologies, and a recognized transition path for promising software systems engineering research into relevant programs.

These objectives will be achieved through a focus on 4 key areas; Advancing key technologies, leveraging the community, transitioning into practice, and learning from experience.

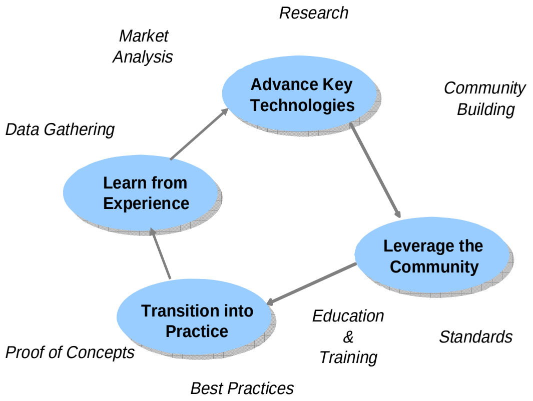


Fig. 2. Academic/Industrial Collaboration

Advance Key Technologies: Identify and advance emerging technologies to address significant and pervasive software systems engineering problems, and develop these technologies to improve software engineering practices in industry. Work with the research community to help create and identify new and improved practices by creating cooperative research and development agreements with industry and academia prove out new and emerging technologies.

Leverage the Community: Work closely with the broad software systems engineering community to seek out best practice and to raise the quality of acquired and delivered software-intensive systems. Work through the global community of software systems engineers to amplify the impact of the new and improved practices by encouraging and supporting their widespread adoption, with the aim of raising the "average" practices much closer to best practices. In addition, this will be supported through the packaging and deliver of a variety of education and training courses and technology practices based on matured, validated, and documented solutions.

Transition into Practice: Work with leading-edge software developers and acquirers to apply and validate new and improved practices. Assist the stakeholders to address specific software systems engineering and acquisition challenges, e.g. clearance of critical aircraft systems, by applying these practices. Transition activities will be primarily funded through contracted additional services with a range of different stakeholders.

Learn from Experience: Build a base of empirical data by undertaking studies, analysis and surveys aimed at establishing a realistic understanding of the current state-of-the-practice for software systems engineering within the systems and software supplier community. In addition to the value of this base data to the wider software systems engineering community, it will also be used to establish appropriate measures for assessing impact of technologies as they transition into practice, and for adjusting the research activities undertaken by in response to that feedback. Where practical this data will also be used to help compare practices with those in other sectors, and to identify opportunities to learn from experience in these other sectors.

5 Summary and Conclusions

It is tempting to believe that software development is easy. You gain an understand-ding of the problem that needs to be addressed by talking with people familiar with that domain, and then design a solution to meet those needs and deploy it in the customer's environment. Unfortunately, complexity and scale get in the way to make the task of software development a lot more challenging.

Software engineers turn to software architecture as a cornerstone for managing complexity and scale in software development. An architecture is an abstraction of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones. All forms of engineering rely on architectures as essential to understanding complex real world systems. Architectures are used in many ways: predicting system qualities, reasoning about specific properties when aspects of the system are changed, and communicating key system characteristics to its various stakeholders.

Here, we have considered the current state of software architecture, identified key architectural trends and directions in academia and industry, and highlighted some of the architectural research challenges which need to be addressed. The paper has proposed a detailed agenda of research activities to be carried out by a partnership between academia and industry. It is our firm belief that this combination of strengths from the two communities will be the basis for future progress in the art and science of software architecture.

References

1. Friedman, R.: The World is Flat: A Brief History of the 21st Century, Farrar, Straus and Giroux (2005)
2. Bhagwati, J.: In Defence of Globalization. Oxford University Press, Oxford (2004)
3. The Challenges of Complex IT Projects: The Royal Academy of Engineering, and British Computer Society (April 2004)
4. Computer Weekly Article (2004)
5. Rumbaugh, J., Booch, G., Jacobsen, I.: The UML Reference Manual. Addison-Wesley, Reading (2004)

6. Ministry of Defense: Defense Technology Strategy for the Demands of the 21st Century, UK MoD (2006), www.science.mod.uk
7. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)
8. Damm, W., et al.: Formal Verification of an Avionics Application using Abstraction and Model Checking. In: Redmill, F., Anderson, T. (eds.) Towards System Safety, Springer, Heidelberg (1999)
9. Barnes, J.G.P.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, Reading (2003)
10. King, S., Hammond, J., Chapman, R., Pryor, A.: Is Proof more Cost-Effective than Testing? IEEE Transactions on Software Engineering 26(8) (2000)
11. Iwu, F., Galloway, A., McDermid, J.A., Toyn, I.: Integrating Safety and Formal Analysis using UML and PFS, RE&SS (2007)
12. Blow, J.R.: Use of Formal Methods in the Development of Safety-critical Control Software. DPhil thesis, Dept of Computer Science, University of York. YCST-2003-08 (2003)
13. Bate, I.J., Burns, A., Audsley, N.C.: Putting Fixed Priority Scheduling Theory into Engineering Practice for Safety Critical Applications. In: Proceedings of 2nd Real-Time Applications Symposium (1996)
14. Eccles, M.A.: STAMP Tool Assessment. BAe-WSC-RP-R&D-0031, BAe Warton (July 1995)
15. Douglass, B.P.: Real-Time UML: Developing Embedded Objects for Embedded Systems. Addison-Wesley, Reading (1998)
16. Radjenovic, A., Paige, R.F.: Architecture Description Languages for High Integrity Real-Time Systems. IEEE Software 23(2), 71–79 (2006)
17. Wallace, M.: Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In: FESCA'05. Formal Foundations of Embedded Systems and Component-Based Software Architectures (2005)
18. Bosch, J.: Design and Use of Software Architectures. Addison-Wesley, Reading (2000)
19. Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., MacGregor, J.: Configuration in Industrial Product Families: The ConIPF Methodology (2006), see http://www.conipf.org
20. Stephenson, Z., McDermid, J.A.: Deriving Architectural Flexibility Requirements in Safety-Critical Systems. IEE Proceedings on Software 154(4) (August 2005)
21. Stephenson, Z., Zhan, Y., Clark, J., McDermid, J.: Test Data Generation for Product Lines - A Mutation Testing Approach. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, Springer, Heidelberg (2004)
22. Arthan, R., Caseley, P., O'Halloran, C., Smith, A.: ClawZ: Control Laws in Z. In: Liu, S., McDermid, J.A., Hinchey, M.G. (eds.) Proceedings of ICFEM 2000, IEEE Computer Society, Los Alamitos (2000)
23. Galloway, A., Paige, R.F., Tudor, N.J., Weaver, R.A., Toyn, I., McDermid, J.A.: Proof versus testing in the context of Safety Standards. In: 24th Digital Avionics Systems Conference (2005)
24. Kruchten, P.: Rational Unified Process. Addison Wesley, Reading (2002)
25. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20 (September 2003)
26. OMG: MDA Guide, Version 1.0.1 (2003), www.omg.org
27. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley Press, Chichester (2003)

28. Fowler, M.: Comments on UML sketching (2005), www.fowler.com
29. Greenfield, J., Short, S., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
30. MODAF Tools Policy Statement of Position: MoD (2006)
31. MODAF M3 Meta Model V1.0: MoD (April 2006)
32. MODAF Tool Certification Plan: V1.0, MoD (April 2006)
33. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Addison-Wesley, Reading (1998)
34. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2001)
35. Krafzig, D., Banke, K., Slama, D.: Enterprise SOA. Prentice-Hall, Englewood Cliffs (2005)
36. Bieberstein, N., et al.: Service-Oriented Architecture (SOA) Compass: Business Value, Planning, and Enterprise Roadmap. IBM Press (2005)
37. Herzum, P., Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. Prentice-Hall, Englewood Cliffs (2002)
38. Parnas, D.L.: Software Engineering Programmes are not Computer Science Programmes. IEEE Software (November/December 1999)
39. McDermid, J.A.: Tailoring Software Engineering Education: One Size Does Not Fit All. Ingenia, RAEng, pp. 50–54 (2004)
40. Brooks, F.: The Mythical Man-Month: 20th Anniversary edn. Addison-Wesley, Reading (2004)
41. Carlson, D.: Eclipse Distilled. Addison-Wesley, Reading (2005)
42. Brown, A.W., Delbaere, M., Eeles, P., Johnston, S., Weaver, R.: Realizing Service oriented Solutions with the IBM Software Development Platform. IBM Systems Journal 44(4), 727–752 (2005)
43. Johnston, S.K., Brown, A.W.: A Model-driven Development Approach to Creating Service-oriented Solutions. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 624–636. Springer, Heidelberg (2006)
44. IBM Insurance Application Architecture. http://www.ibm.com/industries/financialservices/doc/content/solution/278918103.html

Issues in Applying Empirical Software Engineering to Software Architecture

Davide Falessi[1], Philippe Kruchten[2], and Giovanni Cantone[1]

[1] University of Rome "Tor Vergata", DISP, Rome, Italy
falessi@ing.uniroma2.it, cantone@uniroma2.it
[2] University of British Columbia, ECE, Vancouver, Canada
pbk@ece.ubc.ca

Abstract. Empirical software engineering focuses on the evaluation of software engineering technologies, such as processes and tools, by comparing related sets of data. It has contributed a valuable body of knowledge in several areas such as Software Economics and Software Quality, which in turn drove important advances in related tools and techniques. Unfortunately this is not (yet) the case for software architecture, where empirical studies are still few. Such a condition demands for further empirical research efforts on the topic of software architecture and suggests specific areas of improvement. In this paper we discuss several essential, innovative, and maybe provocative, questions such as: Why do we have so few applications of empirical software engineering on software architecture? Which are the main difficulties? What can we do?

Keywords: Software architecture, empirical software engineering.

1 Introduction

One of the objectives of Empirical Software Engineering (ESE) is to develop and utilize evidence to advance software engineering methods, processes, techniques, and tools. ESE is now an advancing discipline; this is demonstrated by the growing consensus from the software community [21], and a number of specialized conferences [8], journal [2], and books [10] [20]. ESE has achieved considerable results in building valuable bodies of knowledge, which in turn drove important advances in different areas; for instance, the effective application of empiricism in software engineering gave excellent results in the area of Software Economics [5] and value-based Software Engineering [4], and improved the techniques for inspecting software artifacts for defects detection [17]. Sharing architectural knowledge seems to be one of the most promising next steps for advancing research and improving knowledge about Software Architecture. This is attested by a number of workshops [14] and publications in software architecture conferences [13]. Although ESE would seem to be a valid means to improve software architecture maturity and the sharing of knowledge, the literature exhibits very few empirical studies on software architecture and we wonder why? Is it so difficult? Too difficult? What can we do? We will discuss these questions in the remaining of this brief paper: Section 2 describes some of the challenges in applying ESE on SA. Section 3 concludes in describing what we could do.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 257 – 262, 2007.

2 Challenges in Applying ESE to Software Architecture

2.1 How to Define Software Architecture "Goodness"?

According to Bass *et al.* [3] analyzing an architecture without knowing the **exact criteria for goodness** is like beginning a trip without a destination in mind." On the same vein, Booch writes "one architectural style might be deemed **better** than another for that domain because it better resolves those forces. In that sense, there's a **goodness of fit**—not necessarily a perfect fit, but **good enough**." [6] The concept of "good enough" may vary according to some aspects, including:

- **Bounded rationality:** the level of goodness heavily depends on the amount of available knowledge at the time of the evaluation [18]. Software architecture is an artifact that software development lifecycle delivers very early; this implies that software architecture decisions are often made based on unstable and quite vague system requirements. Hence software architecture goodness depends on the existent level of risk, which is difficult to describe.
- **Other decisions:** Design decisions are made based on the characteristics of the relationships that they have with the other decisions outside of the reach of the architect, called "pericrises" in [12]. The few available tools for handling software architecture design decisions are still in infancy [7].
- **ROI:** In general, in every system development, the optimal set of decisions is the one which maximizes the return of investment (ROI). For example, an existing architecture is more valuable than the best one achievable by some modifications because the latter would impose some delay and typically delay implies financial losses. Therefore, the ROI should be considered as the main driver to define the goodness of software architecture while in empirical software engineering, correct is quite always defined as the most frequent output. The difficulties in describing the ROI are a fundamental barrier to the definition of the empirical measures.
- **Social factors:** Social issues such as business strategy, culture, size of the development team, etc. crucially influence every kind of design decisions. The difficulties in describing social issues that influence design decisions constitute a barrier when trying to measure and/or control at constant level related empirical variables (i.e. "you cannot control what you cannot measure," as DeMarco said).

2.2 How to Describe the Software Architecture Evaluation Technique?

In the absence of enough evidence, we should assume that different evaluation techniques might lead to different results. Based on the work by Ali Babar *et al.* [1], we propose the following set of attributes to characterize an software architecture evaluation technique and hence its results: level of maturity of the evaluation technique, definition of SA, objective (risk identification, change impact analysis, trade-off analysis), techniques employed for evaluation (e.g., questionnaire, checklist, scenarios, metrics, simulation, mathematical modeling, experience-based reasoning), quality attributes taken into account (see below "SA evaluation result description"), stage of the software process lifecycle (initial, mid-course, post-deployment), type of architectural description (formal, informal, specific ADL, …), type of evaluation (quantitative, qualitative), non technical issues (e.g., organizational structure), required effort, size of the evaluation team, kind of involved stakeholders, and number of scenarios, if any. This set of attributes represents only a basic frame of reference for starting a discussion on how to arrive at an agreement for the description of a software architecture evaluation technique. The difficulty in describing this evaluation step is a barrier for a valid empirical data analysis.

2.3 How to Select the Software Architecture Evaluation Input?

Similarly, we assume that different types of input to evaluate the software architecture may lead to different results. Following Obbink *et al.* [15] we propose the following set of input which influences the result of a software architecture evaluation: objectives (e.g., certifying conformance to some standard, assessing the quality of the architecture, identifying opportunities for improvement, improving communication between stakeholders), scope (i.e. what exactly will be reviewed), architectural artifacts, supporting evidence (e.g. feasibility studies, prototypes, minutes, notes, white papers, measurements), architectural significant requirements, product strategy and planning, standards and constraints, quality assurance policies, risk assessments. As in Section 2.2, this is only a starting point to achieve a consensus on what constitutes a reasonable set of input for a software architecture evaluation process, and this represents another key barrier for ESE to overcome in the field of SA.

2.4 How to Describe the Software Architecture Evaluation Scenarios?

“There are also some attribute model-based methods and quantitative models for software architecture evaluation but, these methods are still being validated and are considered complementary techniques to scenario-based methods.” [1] In fact, different scenarios lead to different conclusions, therefore the researchers should focus on two main issues when doing empirical studies of scenarios: i) apply all the treatments on each scenario ii) describe each scenario in the report, otherwise the readers would not be able to understand to what extent the empirical results are applicable to their own context. While we have quasi standard rules for describing our experimental apparatus (e.g., type of subjects, their incentives [9]), there are no standards for describing the scenarios used for evaluating an architecture. Lacking these, researchers describe them in an *ad hoc* manner, which eventually decreases the effectiveness of the evaluation and hence its validity. The difficulty in completely describing the evaluation scenarios is a barrier for a valid empirical data analysis.

2.5 How to Describe the Software Architecture Evaluation Results?

According to [3], software architecture quality can be defined in terms of ease of creation, conformance, manufacturability, environment impact, suitability, interoperability, security, and a long list of other “-ilities”. The contribution of such a list is threefold: (i) to provide a basic frame of reference for defining the goodness of an architecture (by including, refining, or omitting the proposed attributes), (ii) to stress that most of the attributes can be measured, so that we can describe software architecture “goodness” in an objective way, (iii) to highlight that some of the "ilities" to evaluate a software architecture can only be analyzed once the system has been developed.

2.6 How to Sustain the Cost of Professional Review?

According to [3], a professional architecture review costs around 50 staff-days, and this cost alone is a barrier for empirical studies.

2.7 How to Adopt Sophisticated System?

A main feature of software architecture is to provide “intellectual control over a sophisticated system enormous complexity” [11]. Hence software architecture is really useful only for large software system whose complexity would not be manageable otherwise. The use of software architecture artifacts for small or simple systems, as empirical objects adopted in a typical academic study with students, would be not representative of the state of the practice and moreover the study would neglect phenomena characterizing complex systems. This is a barrier to the construction of valid *artificial* empirical objects; note that this does not imply only a low generality of the results, but also that something is wrong with the results.

2.8 How to Define the Boundaries of Software Architecture?

There is no clear agreement on the definition of SA [19] [15]. Software architecture comprehends the set of decisions that have an impact on the system behavior and not just parts of it. Hence, an element is architecturally relevant based on the locality of its impacts rather than on where or when it was developed. The difficulty in specifying the boundaries between software architecture and the rest of the design is a barrier to the selection of empirical objects to study.

2.9 How to Adopt Subjects with Experience?

In the absence of systematic methods, making the right software architectural decisions requires a high level of experience. Hence the use of empirical subjects with little experience, usually students, is not representative of the state of the practice. Again, this challenge is a barrier to the selection of valid empirical subjects.

The challenges we’ve identified can be categorized in two types: theoretic and practical as described in Table 1. Table 2 shows which types of challenges negatively affect which ESE issues. In particular, practical challenges affect the construct validity, the experimental strategy and both quantitative and qualitative types of analysis. Theoretic challenges affect the internal and conclusion validity, all the available empirical strategy, and the quantitative analysis.

Table 1. Practical and theoretic challenges in applying ESE on SA

Practical	Cost of professional review	Size of the system	Hazy boundaries	Level of experience
Theoretic	SA goodness	SA evaluation technique	SA evaluation scenario	SA evaluation result description

Table 2. ESE issues negatively affected by different types of challenges

Type of challenges	Spoiled Issues		
	Validity	Empirical strategy	Data analysis
Practical	Construct	Experiment	Both
Theoretic	Internal and Conclusion	All	Mainly quantitative

3 Conclusion

Mulla Nasruddin was a 13th century Sufi visionary who wrote the parable known as “searching the keys under the lamppost.” He describes a man who lost his keys in a dark place and who then tried to recover them not were he lost them but far away, under a lamp, because this was the only place with enough light for searching. In this context the dark areas are the aforementioned challenges. Most software architecture research seems to occur under the lamppost. There are many dark areas and it is the main reason of the current mismatch between ESE and SA. If we focus on these dark areas we could: 1) search in the most enlightened part as possible (i.e., we already have standard metrics hence we should avoid to use other subjective dependent variables), 2) consider studies that searched in the dark as valid instead of invalid if there were no light available in the surroundings, but also 3) we may install new lamps in key dark areas (i.e., face the challenges we described above).

Future works include an a characterization of the available empirical methods, based on the level of impact of the above mentioned challenges.

References

1. Babar, M.A., Zhu, L., Jeffery, R.: A Framework for Classifying and Comparing Software Architecture Evaluation Methods. In: Australian Software Engineering Conference (2004)
2. Basili, V., Briand, L.C.: Empirical Software Engineering: An International Journal, ISSN: 1382-3256
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Massachusetts (2003)
4. Biffl, S., Aurum, A., Bohem, B., Erdogmus, H., Grünbacher, P.: Value-Based Software Engineering. Springer, Heidelberg (2005)
5. Boehm, B.W.: Software Engineering Economics. Prentice Hall PTR, NJ (1981)
6. Booch, G.: Goodness of Fit. IEEE Software 23(6), 14–15 (2006)
7. Capilla, R., Nava, F., Perez, S., Dueñas, J.C.: A web-based tool for managing architectural design decisions. SIGSOFT Softw. Eng. Notes 31(4) (November 2006)
8. ESEM: International Symposium on Empirical Software Engineering and Measurement (2007), http://www.esem-conferences.org/
9. Host, M., Wohlin, C., Thelin, T.: Experimental context classification: incentives and experience of subjects. In: Proceedings of the 27th international Conference on Software Engineering, St. Louis, MO, USA, pp. 470–478 (2005)
10. Juristo, N., Moreno, A.: Basics of Software Engineering Experimentation. Springer, Heidelberg (2001)
11. Kruchten, P., Obbink, H., Stafford, J.: The Past, Present, and Future for Software Architecture. IEEE Software 23(2), 22–30 (2006)

12. Kruchten, P.: A Taxonomy of Architecture Design Decisions in Software-Intensive Systems. In: 2nd Groningen Workshop on Software Variability Management, Groningen, NL (2004)
13. Kruchten, P., Lago, P., van Vliet, H.: Building up and Reasoning about Architectural Knowledge. In: 2nd International Conference on the Quality of Software Architectures, Vaesteras, Sweden (June 2006)
14. Lago, P., Avgeriou, P.: First ACM Workshop on SHAring and Reusing architectural Knowledge (SHARK). SIGSOFT Software Engineering Notes 31(5), 32–36 (2006)
15. Obbink, H., et al.: Report on Software Architecture Review and Assessment (SARA). Version 1.0 (2002), available from: http://philippe.kruchten.com/architecture/SARAv1.pdf
16. SEI: Published Software Architecture Definitions (2007), http://www.sei.cmu.edu/architecture/published_definitions.html
17. Shull, F., Seaman, C., Zelkowitz, M.: Victor R. Basili's Contributions to Software Quality. IEEE Software 23(1), 16–18 (2006)
18. Simon, H.A.: The Sciences of the Artificial, 3rd edn. The MIT Press, Cambridge (1996)
19. Smolander, K.: Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In: International Symposium on Empirical Software Engineering, Nara, Japan (October 3-4, 2002)
20. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: an Introduction, Massachusetts (2000)
21. Zelkowitz, M.V., Wallace, D.R.: Experimental models for validating technology. IEEE Computer 31(5), 23–31 (1998)

Leveraging Architecture Patterns to Satisfy Quality Attributes

Neil B. Harrison and Paris Avgeriou

Department of Mathematics and Computing Science, University of Groningen,
Groningen, The Netherlands
harrisne@uvsc.edu, paris@cs.rug.nl

Abstract. Architectural design has been characterized as making a series of decisions that have system-wide impact. These decisions have side effects which can have significant impact on the system. However, the impact may be first understood much later; when the system architecture is difficult to change. Architecture patterns can help architects understand the impact of the architectural decisions at the time these decisions are made, because patterns contain information about consequences and context of the pattern usage. However, this information has been of limited use because it is not presented consistently or systematically. We discuss the current limitations of patterns on evaluating their impact on quality attributes, and propose integrating the information of patterns' impact on quality attributes in order to increase the usefulness of architecture patterns.

Keywords: Software Architecture, Architecture Patterns, Quality Attributes.

1 Introduction

One of the most challenging aspects of software design is creating a system that provides the quality attributes needed by the users. Quality attributes are characteristics of the system that are non-functional in nature. Typical quality attributes include reliability, usability, and security.

Because quality attributes are system-wide, their implementation must also be system-wide: satisfaction of a quality attribute requirement cannot be partitioned into a single module or subsystem. Thus, a system-level vision of the system is required in order to ensure that the system can satisfy its quality attributes. One of the primary purposes of the architecture of a system is to create a system design to satisfy the quality attributes. Wrong architectural choices can cost significant time and effort in later development, or may cause the system to fail to meet its quality attribute goals.

Designing an architecture so that it achieves its quality attribute requirements is one of the most demanding tasks an architect faces [3]. One reason is that the architect needs a great deal of knowledge about the quality attributes and about approaches to implementing systems that satisfy them. Yet there are many quality attributes; the ISO 9126 standard lists six primary and 21 secondary quality attributes [14]. In addition, quality attributes often interact – changes to the system often have

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 263–270, 2007.

repercussions on quality attributes elsewhere. Broad knowledge about how to manage tradeoffs among arbitrary quality attributes does not yet exist [2]. Requirements may not be sufficiently specific and are often a moving target. Finally, the consequences of decisions made are often overlooked [4]. As a result, architectural rework is common.

Architecture patterns are a viable approach for architectural partitioning, and have a well-understood impact on quality attributes [20]. However their application has been rather limited due to a number of factors [11]. We propose the systematic use of architecture patterns to help the architect satisfy quality attributes, and thus reduce the risk of later rework. We demonstrate that patterns can help the architect understand the impact architectural decisions that might be overlooked. We explore why patterns have been limited in large-scale industrial application. As an initial step to overcome this limitation we have analyzed several architecture patterns with respect to their impact to key quality attributes, as a means to leverage the knowledge in the patterns.

2 Architectural Decisions

The process of architectural design has been characterized as making a series of decisions that have system-wide impact. Most architectural decisions have multiple consequences, or as Jansen and Bosch put it, result in additional requirements to be satisfied by the architecture, which need to be addressed by additional decisions [15]. Some are intended, while others are side effects of the decision.

Some of the most significant consequences of decisions are those that impact the quality attributes of the system. Garlan calls them key requirements [10]. This impact may be the intent of the decision; for example, one may choose to use a role-based access control model in order to satisfy a security quality attribute. Other impacts may be side effects of different decisions. For example, the architect may adopt a layered architecture approach, which decomposes the system into a hierarchy of partitions, each providing services to and consuming from its adjacent partitions. A side effect of a layered architecture is that security measures can be easily implemented.

2.1 Unforseen Consequences

One of the key challenges in dealing with consequences is the vast amount of knowledge required to understand their impact on all the quality attributes. Bachmann et al note that the list of quality attributes in the ISO 9126 standard is incomplete, and that one must understand the impact on even the undocumented quality attributes [2]. Tyree et al note that traditional architecture methods do not focus on the rationale for an architectural decision and the options considered [21]. Kruchten notes that the reasoning behind a decision is tacit knowledge, essential for the solution, but not documented [19]. The result is that consequences of decisions may be overlooked.

Overlooking issues is a significant problem in architecture. In a study of architecture evaluations, Bass et al [4] report that most risks discovered during an evaluation arise from the lack of an activity, not from incorrect performance of an activity. Categories of risks are dominated by oversight, including overlooking consequences of decisions. Many of the overlooked consequences are associated with quality attributes. Their top risk themes included availability, performance, security, and modifiability.

Missing the impact on quality attributes at architecture time has an additional liability. Because quality attributes are system-wide capabilities, they generally cannot be fully tested until system testing [7]. Consequences that are overlooked are often not found until this time, and are expensive to fix.

3 Architecture Patterns

Patterns are solutions to recurring problems. A software pattern describes a problem and the context of the problem, and an associated generic solution to the problem. The best known software patterns describe solutions to object-oriented design problems [9], but patterns have been used in many aspects of software design, coding, and development. Patterns have been written for software architecture, and can be used in numerous software architecture methods [3] [5] [8] [15] [20].

Patterns have been shown to be a useful and potentially important vehicle for capturing some of the most significant architectural decisions [11]. One of the biggest difficulties of documenting architectural decisions is the capturing of rationale and expected consequences of a decision. This is where patterns are particularly strong, because the consequences of using the architecture pattern are part of the pattern.

The result of applying a pattern is usually documented as “consequences” or “resulting context” and is generally labeled as positive (“benefits”) or negative (“liabilities”). Each benefit and liability is described in some detail.

The payoff of using patterns can be great. When an architect uses a pattern, he or she can read the pattern documentation to learn about the side effects of the pattern. This reduces the chance of the architect failing to consider important consequences. This relieves the architect of the burden of being expert in all the quality attributes.

An important advantage of pattern-based architecting is that it is an integral part of most current architecture methods. It fits into the step of ADD that selects architectural patterns and tactics to satisfy the drivers (see [3] for further details.) The Siemens’ Views method [13] and the Rational Unified Process 4+1 Views [17] [18], use various strategies, such as patterns, to resolve issues identified in the views.

3.1 Limitations of Patterns in Identifying Consequences

The use of patterns in identifying and dealing with consequences is, however, currently significantly limited. The chief limitation is that patterns’ information on consequences is incomplete, not searchable or cross-referenced, and in general not as easy to use as it should be. Furthermore, it is difficult to learn about pattern interactions: how patterns may jointly impact quality attributes. These are the difficulties we focus on in this work.

Another difficulty is that pattern consequences are most often qualitative, not quantitative. Some quantification of architecture patterns’ impact on quality attributes has been done using a graded scale [20]. This is insufficient, since an architect needs to have rigorous analysis results of quality attributes to make informed decisions.

Even qualitative information is problematic: consequences are of different strengths but no such comparative information is given. We begin to address this in this work.

Another issue is that patterns contain proven, but general solutions. Architecture is concerned with specific, but tentative decisions. As such, the pattern use must be tailored to the specific system – the architect must evaluate the consequences of a pattern in the context of its proposed use. Several architecture patterns, particularly those in Buschmann et al [8], include common variants of the patterns that provide more specific solutions. However, the variants have not been extensively documented, and have little information on consequences. So the user is left to determine whether the consequences of a pattern still apply to a pattern variant under consideration.

An important source of unforeseen consequences is the interaction of multiple decisions. Multiple patterns may have overlapping consequences, or patterns and decisions not based on patterns may have overlapping consequences.

4 Analysis of the Impact of Patterns on QAs

In order for patterns to become a truly powerful architecture tool, it must be possible to find which patterns impact certain quality attributes, compare and contrast their impacts, and discover their interactions. To this end, we are analyzing the impact of patterns on quality attributes, and organizing this analysis in a way that is accessible and informative. This work is a companion to quantifying the impact of patterns on quality attributes: it adds a qualitative dimension by examining the nature of how a pattern impacts a particular quality attribute; not just how much.

We began by selecting a standard definition of quality attributes to be used in the study. We used the ISO quality model [14], which contains functionality, reliability, usability, efficiency, maintainability, and portability. We initially confined ourselves to the primary attributes, with the exception of functionality, where we selected the security sub-attribute. We added a property, implementability, as a measure of the difficulty of implementing the pattern.

We then selected the best-known architecture patterns, those from Buschmann et al [8]. We used the consequences in the book for our analysis of consequences. While the book gives several variants of the patterns, we limited this analysis to the “pure” form of each pattern – the variants will be investigated in our future work.

In the analysis of the consequences, we designated strengths as “strength” or “key strength,” and liabilities as either “liability” or “key liability,” based on the importance of the impact. If the impact on the quality attribute might be sufficient reason by itself to use or avoid the pattern, it was designated as “key.” This differentiation supports architectural reasoning: used in the context of a project’s architectural drivers, a key strength tends to enable fulfillment of an architectural driver, while key liability will severely hinder or perhaps prevent its fulfillment. We differentiated normal versus key impacts based on the severity described in the documentation. Where it was unclear, consequences were weighed against each other, and judgment was applied. Not every pattern had both key strengths and liabilities.

At least two to three sentences are needed to express each impact fully. Because of space limitations, we abbreviated the impacts to just a short sentence.

Table 1. Patterns' Impact on Usability, Security, Maintainability and Efficiency

	Usability	Security	Maintainability	Efficiency
Layers	*Neutral*	*Key Strength:* Supports layers of access.	*Key Strength:* Separate modification and testing of layers, and supports reusability	*Liability:* Propagation of calls through layers can be inefficient
Pipes and Filters	*Liability:* Generally not interactive	*Liability:* Each filter needs its own security	*Strength:* Can modify or add filters separately	*Strength:* If one can exploit parallel processing *Liability:* Time and space to copy data
Blackboard	*Neutral*	*Liability:* Independent agents may be vulnerable	*Key Strength:* extendable *Key Liability:* Difficult to test	*Liability:* Hard to support parallelism
Model View Controller	*Key Strength:* Synchronized views	*Neutral*	*Liability:* Coupling of views and controllers to model	*Liability:* Inefficiency of data access in view
Presentation Abstraction Control	*Strength:* Semantic separationo	*Neutral*	*Key Strength:* Separation of concerns	*Key Liability:* High overhead among agents
Microkernel	*Neutral*	*Neutral*	*Key Strength:* Very flexible, extensible	*Key Liability:* High overhead
Reflection	*Neutral*	*Neutral*	*Key Strength:* No explicit modification of source code	*Liability:* Meta-object protocols often inefficient
Broker	*Strength:* Location Transparency	*Strength:* Supports access control	*Strength:* Components easily changed	*Neutral:* Some communication overhead

4.1 Implications of Analysis

A few patterns have conflicting impacts on a quality attribute. The Blackboard pattern has both a positive and negative impact on maintainability, and efficiency is both a strength and a liability in the Pipes and Filters pattern. This shows the complex nature of quality attributes: the categories above should be broken down in more detail (see future work.) However, they also indicate that a pattern can have complex consequences. In these cases, the designer must consider multiple different impacts.

The context of the application affects the importance of the consequences. For example, the efficiency strength of Pipes and Filters to exploit parallel processing may not be achievable in some single thread systems. This also highlights how best to use the information: one uses the information as a starting point for more in-depth analysis and design. This is particularly true for the liabilities, as illustrated below.

Table 2. Patterns' Impact on Reliability, Portability, and Implementability

	Reliability	**Portability**	**Implementability**
Layers	*Strength*: Supports fault tolerance and graceful undo	*Strength*: Can confine platform specifics in layers	*Liability*: Can be difficult to get the layers right
Pipes and Filters	*Key Liability*: Error handling is a problem	*Key Strength*: Filters can be combined in custom ways	*Liability*: Implementation of parallel processing can be very difficult
Blackboard	*Neutral*: Single point of failure, but can duplicate it	*Neutral*	*Key Liability*: Difficult to design effectively, high development effort
Model View Controller	*Neutral*	*Liability:* Coupling of components	*Liability:* Complex structure
Presentation Abstraction Control	*Neutral*	*Strength:* Easy distribution and porting	*Key Liability:* Complexity; difficult to get atomic semantic concepts right
Microkernel	*Strength:* Supports duplication and fault tolerance	*Key Strength:* Very easy to port to new hardware, OS, etc	*Key Liability:* Very complex design and implementation
Reflection	*Key Liability:* Protocol robustness is key to safety	*Strength:* If you can port the meta-object protocol	*Liability:* Not well supported in some languages
Broker	*Neutral:* Single point of failure mitigated by duplication	*Key Strength:* Hardware and OS details well hidden	*Strength:* Can often base functionality on existing services.

We have used this information in evaluating the architecture patterns in a few industrial systems. While this work is early, our studies indicate that such evaluations can be very useful. The process consists of identification of the patterns in the architecture, and examining their impact on the important quality attributes of the system. In one case, we reviewed an architecture which used the Pipes and Filters pattern. A key liability of this pattern is reliability; it is difficult to implement error handling. This became a drill-down point in the review, and we investigated error handling in more depth. In another case, we observed the Layers pattern in a time-critical system. In order to deal with the fact that the Layers pattern has a performance liability, the designers allowed certain functions at the lowest layers to be called from the highest layer. Such "breakages" of a pattern should be designated as areas for careful testing, because they are intentional deviations from a proven design.

Early experience suggests that it supports lightweight architecture. It adds some rigor to architecture without extensive documentation and reviews. It strikes a "middle ground" between the extremes of no architecture and highly formalized architecture.

5 Related Work

Several quality attribute centered software architecture methods take an intuitive approach, including the QASAR method [5] and the attribute driven design (ADD) method [3]. Use of architecture patterns is also intuitive, and fits well in these models. In addition, the architecture pattern quality attribute information formalizes architecture patterns and their consequences, relieving the architect of some of the burden of ferreting out the consequences of architectural decisions.

Bachmann et al describe a knowledge framework designed to help architects make specific decisions about tradeoffs that impact individual quality attributes [2]. It focuses on individual quality attributes independently, while the pattern approach focuses more on interactions among patterns and quality attributes. It might be said that the knowledge framework favors depth, while the pattern-driven approach favors breadth. In this sense, it is likely that these two research efforts are complementary.

In the general model of architecture [12], the information is useful in the *Architectural Synthesis* activity, but is most valuable in the *Architectural Evaluation* activity. Architecture evaluators can use it to help them detect risks of omission [4].

6 Future Work

We have shown an initial analysis of a few architecture patterns identified to date, namely those found in Buschmann et al [8]. We are beginning to analyze others; most are described in Avgeriou and Zdun [1]. We have also begun analyzing the subcategories of quality attributes given in the ISO quality standard [14].

A process for using this data in pattern-based architecture reviews is being developed and used to collect data.

The interaction of the consequences of patterns has not been explored in detail. We intend to study which patterns are often used together. This helps identify potentially conflicting decisions, and help them make tradeoffs about which patterns to use.

The consequences in patterns are qualitative, but some quantification is useful. It would be useful to make the rudimentary quantification of the consequences: Key Strength, Strength, Neutral, Liability, and Key Liability more detailed. Such quantification is of necessity limited, and must be carefully crafted so as not to give the false impression that a numerical score of a pattern can replace analysis.

Pattern variants have rich potential. Variants of patterns should be investigated to understand in more detail how individual variants affect the impact of generic architecture patterns on the quality attributes. Different pattern variants have somewhat different strengths and liabilities. This information can be used to help the architect choose among different variants of patterns.

A table such as the ones above can show only very abbreviated information; more detailed information is needed. This information might be incorporated into a tool that functions as an architectural decision support system such as knowledge frameworks for quality attribute requirements as proposed by Bachmann et al [2].

References

1. Avgeriou, P., Zdun, U.: Architectural Patterns Revisited - a Pattern Language. In: 10th European Conference on Pattern Languages of Programs, Irsee, Germany (July 2005)
2. Bachmann, F., Bass, L., Klein, M., Shelton, C.: Designing software architectures to achieve quality attribute requirements. In: IEE Proceedings, vol. 152 (2005)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley, Reading, MA (2003)
4. Bass, L., Nord, R., Wood, W., Zubrow, D.: Risk Themes Discovered Through Architecture Evaluations. SEI Report CMU/SEI-2006-TR-012 (2006)
5. Bosch, J.: The Design and use of Software Architectures. Addison-Wesley, London (2000)
6. 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)
7. Burnstein, I.: Practical Software Testing. Springer, Heidelberg (2003)
8. Buschmann, F., Meunier, R., Rhonert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, West Sussex, England (1996)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
10. Garlan, D.: Software Architecture: a Roadmap. In: Proceedings of Future of Software Engineering, Limerick Ireland (2000)
11. Harrison, N., Avgeriou, P., Zdun, U.: Architecture Patterns as Mechanisms for Capturing Architectural Decisions. IEEE Software (September/October 2007)
12. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: Generalizing a Model of Software Architecture Design from Five Industrial Approaches. Journal of Systems and Software 30(1), 106–126 (2007)
13. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture, pp. 7–8. Addison-Wesley, Reading, MA (2000)
14. International Standards Organization: Information Technology - Software Product Quality - Part 1: Quality Model, ISO/IEC FDIS 9126-1
15. Jansen, A.G., Bosch, J.: Software Architecture as a set of Architectural Design Decisions. In: Proceedings of WICSA 5, pp. 109–119 (November 2005)
16. Klein, M., Kazman, R.: Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-T2-022 (October 1999)
17. Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software 12(6) (1995)
18. Kruchten: The Rational Unified Process: an Introduction, 3rd edn. Addison-Wesley, Reading (2004)
19. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, Springer, Heidelberg (2006)
20. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Addison-Wesley, Reading, MA (1996)
21. Tyree, J., Ackerman, A.: Architecture Decisions: demystifying Architecture. IEEE Software, 19–27 (March/April 2005)

Architecture for Developing Adaptive and Adaptable Collaborative Applications*

Mario Anzures-García[1,2], Miguel J. Hornos[2], and Patricia Paderewski-Rodríguez[2]

[1] Facultad de Ciencias de la Computación, Benemérita Universidad Autónoma de Puebla, 14 sur y avenida San Claudio. Ciudad Universitaria, San Manuel, 72570 Puebla, México
manzures@siu.buap.mx
[2] Departamento de Lenguajes y Sistemas Informáticos, E.T.S.I. Informática y de Telecomunicación, Universidad de Granada, C/ Periodista Saucedo Aranda, s/n, 18071 Granada, Spain
anzures@correo.ugr.es, {mhornos,patricia}@ugr.es

Abstract. Many organizations have to carry out their work by groups of people who are geographically distributed. The groups can experiment changes, which demand the development of applications supporting groupwork and allowing the adaptation to different groupwork organization styles and to both individual and collective needs. This paper proposes a SOA-based architecture that provides the suitable structure for the development of collaborative applications that are both adaptive and adaptable. We also present an adaptation process that allows the applications based on the architecture to be adapted to the changes in the groupwork organization and to the necessity of new functionalities.

1 Introduction

The collaborative applications must provide the appropriate infrastructures to support groupwork and to do it effectively. Moreover, it is advisable that these applications can be adapted to the group dynamic nature and to the different groupwork styles.

Software adaptation is based on the need of adjusting or transforming the system functionality in accordance with the new requirements that will appear in the future (changes in the environment, new users' needs, different devices, etc.), in such a way that the system can continue working correctly. There are two forms of adaptation [4], namely: 1) Adaptive, when the system adaptation is automatically performed and is based on certain mechanisms previously defined by the designer and/or the developer. 2) Adaptable, when the system adaptation is carried out by the user's direct intervention in accordance with a set of constrains that avoid inconsistencies in the system. This paper presents a SOA-based architecture that provides a suited infrastructure to develop collaborative applications that can be adapted to the changes and needs of the group in runtime. The next section shows our architectural proposal and the final section outlines our conclusions.

* This work is financed by the Spanish project CICYT with code TIN2004-08000-C03-02.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 271–274, 2007.

2 Adaptive and Adaptable Architecture

The SOA-based architecture to develop adaptive and adaptable collaborative applications that we propose is made up of three layers: Group Layer, Application Layer, and Control Layer, which are connected by the communication mechanisms (i.e. HTTP, SOAP, WSDL and UDDI). In next subsections, we will briefly explain the first two layers (an extended description of them can be found in [1]), while we will focus on the Control Layer, and mainly on the adaptation processes.

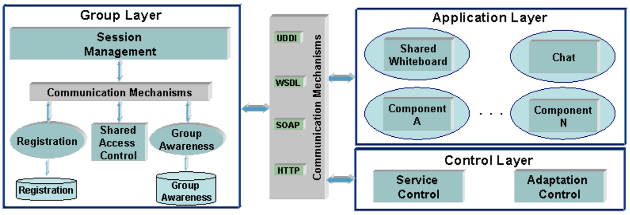


Fig. 1. Architecture to Develop Adaptive and Adaptable Collaborative Applications

2.1 Group Layer

It supplies the appropriate infrastructure to support groupwork. This layer contains the following components:

- *Registration Service*, which allows registering sessions and users. To do it, it uses an authentication mechanism.
- *Session Management Module*, which provides a mechanism that allows users to carry out the groupwork using shared resources. In addition, this module defines how the groupwork will be organized by means of the session management policy, which we have modelled using an ontology [2].
- *Shared Access Control Module*, which determines how participants to a session cooperate, by providing temporary access to resources for collaborating users in order to mutually guarantee exclusive resource usage.
- *Group Awareness Service*, which helps participants to establish a common context and coordinate activities, avoiding surprises and reducing conflicts in the group.

2.2 Application Layer

It contains the specific collaborative applications that users want to employ to carry out the groupwork (e.g. a shared whiteboard). When needed, this layer allows the extending of the architecture functionality to new web services without having to modify existing services. Since the architecture has been designed using a service-oriented architectural style, third-party applications, components or tools (which are wrapped as web services) may be added to provide new functionalities.

2.3 Control Layer

It is made up of two modules: the *Service Control Module*, which controls the description, discovering, binding and/or publication of web services in order to facilitate the building or extension of applications using the existing services; and the *Adaptation Control Module* (see Figure 2), which controls how the components of our architecture will be adapted when a change (or event) requiring the modification of the collaborative application takes place, so that the application functionality can be preserved. The adaptation process, which is based in the model proposed in [3], includes the following phases:

- *Event Detection*, which monitors events in the execution environment and determines whether the adaptation process has to be carried out, using the Detection Mechanism. Once an event has been detected, this phase checks whether it is stored in the Adaptation Event repository. This determines whether it is necessary to carry out an adaptation process and which components (stored in the Adaptation Component repository) will be affected by the adaptation.
- *Agreement*, which is performed only in an adaptable process, where all users have to reach a consensus on whether an adaptation process should be performed or not. This consensus process is carried out by the Agreement Mechanism. Several kinds of agreement (such as the one based on either a majority vote, or a maximum or a minimum value, etc.; each of them is applied to a specific situation) can be reached using the Voting tool. All components of the Adaptation Component repository relating to the corresponding adaptation process must meet the necessary conditions (which are stored in the Requirement repository) in order to avoid possible inconsistencies in the collaborative application.
- *Action*, which carries out the corresponding changes by means of the Action Mechanism, in such a way that the application is continuously running its correct functionality. To do that, it executes a set of operations (stored in the Operation repository) that determine which actions must be carried out in each component/service to be adapted.

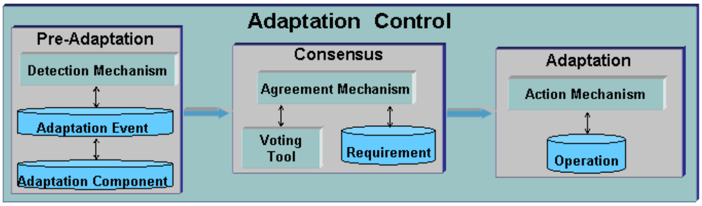


Fig 2. General Schema of the Adaptation Control Module

2.4 Applying the Adaptation Process

We present two examples: an adaptive (when a user leaves a session) and an adaptable (when the group changes the session management policy to be applied) process. In the former, the actions carried out are: a) *Event Detection*. The Detection Mechanism detects the *leave_session()* event (which is stored in the Adaptation Event repository), so it determines (by querying the Adaptation Component repository) that the Registration and Shared Access Control services and the Session Management module have to be adapted. b) *Agreement*. This phase is not carried out. c) *Action*. Once the *leave_session()* event has been triggered, the following operations have to be performed: 1) to delete the user name from the Registration repository, 2) to close the connection with the user's site; 3) to check whether the role the user was playing is necessary; in such a case, this role is assigned to another user in accordance with the session management policy; 4) to assign the tasks carried out and the shared resources used by the user to another user (if necessary); 5) to remove the user name of the waiting list of certain shared resources (if necessary); 6) to notify all the session users of the user's leaving; and 7) to update the user interface of all session users. In the latter, the adaptation process carried out consists on: a) *Event Detection*. The detection mechanism detects the *change_policy()* event, which is triggered when the group decides to change the groupwork organization style. Since this event is stored in the Adaptation Event repository, this phase determines (by consulting the Adaptation Component repository) that the Session Management module have to be adapted. b) *Agreement*. In this case, users determine whether the session management policy must be changed using a voting tool or by decision of the user with the highest status. In order to carry out this change, the users must momentarily stop their collaborative tasks so that they liberate the resources they are using. c) *Action*. If the corresponding conditions are fulfilled, the following operations are carried out: 1) to change the session management policy, 2) to assign a role to each user with its status and access rights; 3) to assign tasks to each user (and this depend on the role he/she is playing); and 4) to notify all session users of their role change.

3 Conclusions

In this paper, we have presented a SOA-based architectural proposal to build collaborative applications that are adaptive and adaptable. The adaptation process is focused on supporting different organization styles and covering the possible new needs of the group. In order to do that, we have modelled the session management policy, which defines and manages the necessary dynamic changes to carry out the groupwork adequately, and we have proposed an adaptation process, which is carried out in three phases.

References

1. Anzures-García, M., Hornos, M.J., Paderewski, P.: Development of extensible and flexible collaborative applications using a web service-based architecture. LNCS, vol. 4401, pp. 66–80. Springer, Heidelberg (2007)
2. Anzures-García, M., Sánchez-Gálvez, L.A., Hornos, M.J., Paderewski, P.: Ontology-Based Modelling of Session Management Policies for Groupware Applications. LNCS. Springer, Heidelberg (in press, 2007)
3. Hiltunen, M.A., Schlichting, R.D.: Adaptive Distributed and Fault-Tolerant Systems. Int. Journal of Computer Systems and Engineering 11(5), 125–133 (1995)
4. Medina, N., García, L., Torres, J.J., Parets, J.: Evolution in Adaptive Hypermedia Systems. In: Proc. of Conference on Principles of Software Evolution, pp. 34–38 (2002)

Analyzing Styles of the Modular Software Architecture View

Rogelio Limon Cordero and Isidro Ramos Salavert

Department of Information Systems and Computation, Technical University of Valencia,
Camino de Vera s/n E-46022 Valencia
{rlimon,iramos}@dsic.upv.es

Abstract. Software architecture views represent the basic structures of a complex software system. By means of these views, it is possible to shape the different concerns that appear in the requirements and design phases. A modular view specifies the elements that must be built in the detailed design, and the relationships that must be established among them. This paper makes an analysis of the styles present in the modular view. This work establishes how these styles can be shaped and analyzed by means of their relations.

1 Introduction

Software architecture is a key element in software development because it is the bridge between the requirements and design phases. The design of this architecture allows the user to determine the structures that will shape the software system and the way in which its elements communicate before proceeding with the detailed design. Each structure is really an architectural view, which contains architectural element relationships with each other.

The aim of this paper is to study this area by considering the architectural style as a relationship instead of a type of view. In addition, a proposal is presented to model the relationships in order to make the styles operative in the analysis phase of the modular architectural view.

2 The Modular Architectural View and Its Styles

From the definition presented in [1], the architectural view is considered as a specific perspective of each structure of the architecture.

There are several approaches about what the architectural view should be; among the more relevant, we can enumerate the following: SEI[1], Kruchten [2], and SIEMS[3]. Even though there are some differences among them about how they interpret a view, there are coincidences in the type of elements and relations that are considered in each one. The modular view is included in these approaches. The focus of this work is to analyze and deal with the relationships of elements of the modular view-type[1].

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 275 – 278, 2007.

The relationships among the elements of a view-type adopt either some of the pre-established shapes known as architectural styles (or simply styles) or similar shapes. *Styles* define a set of features and constraints that typify the relationship among the architectural elements.

The basic styles in a modular view type are: *decomposition, uses, generalization* and *layered*. The links established among the architectural elements are relationships of dependence.

3 Establishing the Architectural Relationships

Considering that an architectural style is a specific type of a relationship that is established between at least two architectural elements, a mechanism must be established to identify, represent, and analyze such relationships in order to improve the architectural design process and to achieve a quantitative analysis.

Our approach is based on the design rules of [4], where a dependency matrix for analyzing relations among elements is proposed. In this paper, a basic binary relationship is used to represent the relationships in each one of the styles of the modular view-type.

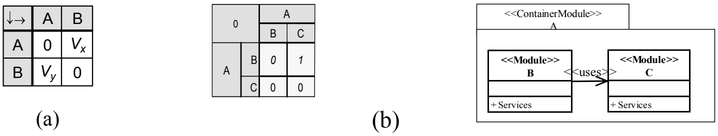
Let's consider that a relationship between two elements (modules) named A and B can be represented by a relationship matrix as shown in Figure 1(a).

In the relationship matrix, V_x, and V_y (in a cell) represent values that indicate the relation between their corresponding row and column elements. Each value has a different interpretation to indicate the type of relationship. Using a representation of binary relational algebra, A V_x B means that A and B have a type of relationship that is represented by the value of V_x, where V_x and V_y are associated with a type of relationship (A_Relationship and B_Relationship) by means of a correspondence, i.e., types of relationships = {(0, null), (V_x, A_Relationship), (V_y, B_Relationship)}.

The range of values for V_x and V_y varies from 0 to R-1, where R is the number associated to the relationship types, and the ´-1` is due to the fact that the null relationship (0) must be excluded.

In this case, the dependence and decomposition relationships are used because the modular view only represents a static relationship; therefore, the value of R for the matrix is 2.

The decomposition and uses architectural styles are represented by means of matrices and their corresponding UML 2.0 representation, as shown in Figure 1(b).



↓→	A	B
A	0	V_x
B	V_y	0

0		A	
		B	C
A	B	0	1
	C	0	0

(a) (b)

Fig. 1. (a) Relationship matrix between two elements, (b) Matrix for basic architectural styles of a modular view, and its corresponding representation in UML 2.0

4 Analysis of Architectural Styles

There are several methods for analyzing software architectures. The main goal of these methods is assess the quality attributes of the system. In a survey that was done on eight methods [5], it was shown how most of these techniques use qualitative criteria to assess the quality attributes of the system; for example ATAM [1][5], which is currently widely used.

The goal in this work is improve the current methods through the inclusion of indicators that evaluate the relationship types used in the software architecture views. These indicators use the relationship matrices as objects of analysis.

The indicators quantify the architectural styles used in each architectural view, taking into account the relationship types that their styles have.

In a modular view type, the styles link the modules through the <<uses>>, <<layered >>, and <<decomposition>> relationships. The interest in these relationships is focused on how the modules are distributed, i.e., if a *module* is overused by others, or if a *Container-Module* includes too many elements. The following indicators are identified to assess this: $UM_{i,m}$ means that the module *m* is used by the module *i*; LD_m indicates the level of dependence in a module *m*, i.e., LD is the number of modules that uses to the module *m*; DX indicates the maximum level of dependence in the view, i.e., $MAX(LD_m)$; IM_m indicates the number of modules included in the module *m;* TM indicates the total of container modules in the view; AM is the average number of modules included in the modular view; MD_m is the modular density of module *m*, where *m* is a module of the <<container>> class.

The valuations in some indicators are:

$$LD_m = \Sigma_{i=1} UM_{i,m} \quad \text{where } i \neq m$$
$$MD_m = NM_m / AM \quad ; \quad \text{where } AM = \Sigma_{i=1} NM_i / TM$$

These can be evaluated using the relationship matrix of the view to be analyzed. For example, $UM_{i,m}$ is obtained in a straightforward way accessing row *i* and column *m* from the matrix. LD_m for the element *m* is calculated by summing all the values of each row of the column where element *m* is located, and so on for the other indicators.

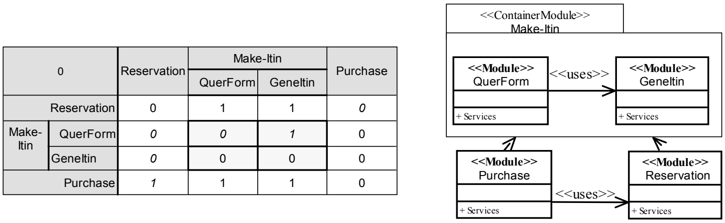
These indicators are useful for making comparisons between two or more views generated in the design process, or for restructuring a module when its MD > 1 or when it has a high value of LD.

5 Applying the Analysis

An analysis is applied to the software architecture for a purchase and reservation ticket system for any kind of passenger transport (bus, air plane, train, etc). This example was presented in [6].

In order to focus on the use of the architecture styles proposed here, let’s suppose that the architectural modules derived from the requirements are: Query formation (QuerForm); Generating itinerary alternatives and routes (GeneItin); Making the reservation(Reservation), and Purchasing a ticket(Purchase). This architecture is shown in matrix and UML 2.0 notation in Figure 2.

For the modules shown in Figure 2(a), the dependence level (DL) for Reservation, Make-Itin and Purchase is: 1, 2 and 0, respectively. The average of modules (AM) included in this modular view is 2. The modular density (MD) for Make-Itin (the only container modular) is 1. This example case only indicates how to evaluate some indicators through the relationship matrices. However, an exhaustive quantitative analysis of a software architecture can be carried out with these matrices in order to evaluate its complexity.



0		Reservation	Make-Itin		Purchase
			QuerForm	GeneItin	
Reservation		0	1	1	*0*
Make-Itin	QuerForm	*0*	*0*	*1*	0
	GeneItin	*0*	0	0	0
Purchase		*1*	1	1	0

Fig. 2. The architectural modular view for a ticket purchase-reservation system, in relationship matrix and UML 2.0 representations

6 Conclusions

By modeling the architectural style as a type of relationship, we have proposed a novel way to analyze an architectural view. The identification and analysis of the basic relationships proposed here can be used in other view types.

The use of the relationship matrices allows us to: a) adapt some view types where there is dynamism among the element relationships; b) represent structural relationships as a composition of architectural views; c) make a quantitative analysis using relationship matrices.

Finally, it is shown that the application of the architectural styles improves the method of design of the software architectural views by making the analysis of relations in the architectural design process operational.

References

1. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2^a edn. Addison-Wesley, Reading (2003)
2. Philippe, K.: The 4+1 View Model of Architecture. Paper published in IEEE Software 12(6), 42–50 (1995)
3. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison Wesley, Boston, MA (2000)
4. Baldwin, C., Clark, K.: Design Rules: The Power of Modularity, vol. 1. The MIT Press, Cambridge, MA (2000)
5. Dobrica, L., Niemelä, E.: A survey on software architecture analysis methods. IEEE Transactions On Software Engineering 28(7), 638–653 (2002)
6. Limon, C.R., Ramos, S.I., Torres, J.J.: Designing Aspectual Architecture Views in Aspect-Oriented Software Development. In: Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganà, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3983, pp. 726–735. Springer, Heidelberg (2006)

Dynamic Reconfiguration of Software Architectures Through Aspects

Cristóbal Costa[1], Nour Ali[1], Jennifer Pérez[2], José Ángel Carsí[1] , and Isidro Ramos[1]

[1] Department of Information Systems and Computation,
Polytechnic University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain
ccosta@dsic.upv.es, nourali@dsic.upv.es,
pcarsi@dsic.upv.es, iramos@dsic.upv.es
[2]Technical University of Madrid (UPM)
E.U. Informática, Ctra. Valencia km 7, 28051 Madrid, Spain
jeperez@eui.upm.es

Abstract. Currently, most software systems have a dynamic nature and evolve at run-time. The dynamic reconfiguration of software architectures has to be supported in order to enable their architectural element instances and their links to be created and destroyed at run-time. Complex components also need reconfiguration capabilities to evolve their internal compositions. This paper introduces an approach to support the dynamic reconfiguration of software architectures taking advantage of aspect-oriented techniques. It enables complex components to autonomously reconfigure themselves: they are capable of both having knowledge of their current configuration and reconfiguring themselves at run-time. This approach has been developed for the PRISMA aspect-oriented architectural model. A new kind of aspect has been created in PRISMA in order to provide dynamic reconfiguration services to each complex component; it is called the *Configuration Aspect*.

Keywords: Dynamic reconfiguration, software architectures, AOSD.

1 Introduction

Currently, most software systems have a dynamic nature that requires them to evolve and adapt to changes at runtime. The development of this kind of systems is a complex task since they are large and may consist of heterogeneous entities. As a result, a great effort has to be done in order to provide system reconfiguration at runtime. Software architectures [8] provide techniques for describing the structure of complex software systems in terms of architectural elements (components and connectors) and interactions among them. Software architectures have a hierarchical structure where components can be composed into complex components and these, in turn, can be composed into more complex components.

Dynamic reconfiguration [3] has to be supported in order to enable architectural element instances and links to be created and/or destroyed at run-time. In the last years, a lot of research efforts have been done to address dynamic reconfiguration of software architectures [1,2]. However, most of these approaches address dynamic

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 279 – 283, 2007.

reconfiguration from a centralized point of view: a global entity is responsible of providing dynamic reconfiguration capabilities to all the architecture. However, architectures are often made up of heterogeneous components and each one has different reconfiguration needs. For this reason, complex components usually need to reconfigure their internal composition by themselves in an autonomous way.

Aspect-Oriented Software Development (AOSD) [4] proposes the separation of the crosscutting concerns of software systems into separate entities called aspects, avoiding the tangled concerns of software and allowing the reuse of the same aspect in different entities of the software system as well as its maintenance. Dynamic reconfiguration is a concern that crosscuts the architecture of those software systems that have a dynamic nature. In order to prevent the dynamic reconfiguration concern from being tangled with the rest of the architecture, a new kind of aspect called *Configuration* has been defined. Thereby, a *Configuration* aspect encapsulates the properties and the behaviour of this concern. Only few proposals have addressed dynamic reconfiguration through aspects [9], and their work is focused at the implementation level instead of dealing reconfiguration at the architectural level.

This paper presents an approach for supporting dynamic reconfiguration in software architectures by means of aspects. The approach enables PRISMA complex components to autonomously reconfigure themselves at runtime, through a *Configuration* aspect, whereas other systems or components of the software architecture are unaware of these dynamic changes.

2 PRISMA

PRISMA [5] is a model that integrates AOSD and software architectures in order to describe the architectural models of complex software systems. In PRISMA, architectural elements are specified by importing a set of aspects. Aspects are first-order citizens of software architectures and encapsulate the properties and the behaviour of *concerns* (safety, coordination, etc) that crosscut the software architecture. A PRISMA architectural element can be seen from two different views: the internal and the external. In the external view, architectural elements encapsulate their functionality as black boxes and publish a set of services through their ports. The internal view shows an architectural element as a prism (white box view). Each side of the prism is an aspect that the architectural element imports. The aspects of architectural elements are synchronized among them through *weavings* relationships.

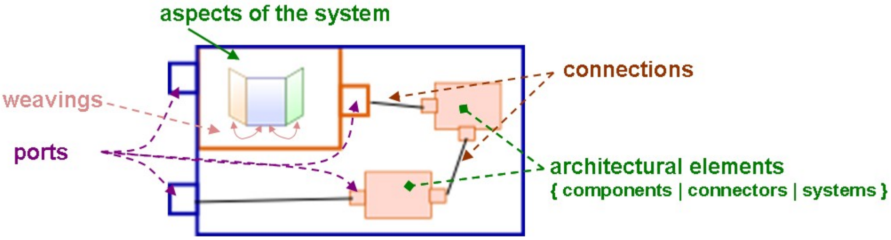


Fig. 1. PRISMA Systems

PRISMA has three kinds of architectural elements: components, connectors, and systems. Components and connectors are simple; systems are complex components. A component captures the functionality of software systems whereas a connector acts as a coordinator among other architectural elements. A PRISMA system is a complex component that includes a set of architectural elements (connectors, components and other systems), the connections among them, and its own aspects and weavings (see Figure1).

3 Dynamic Reconfiguration Through an Aspect

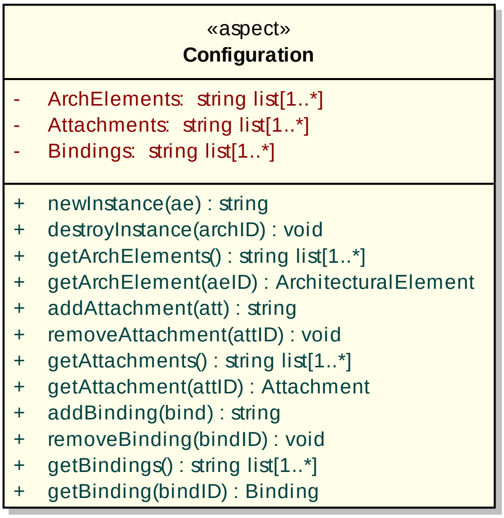

«aspect»
Configuration
- ArchElements: string list[1..*]
- Attachments: string list[1..*]
- Bindings: string list[1..*]
+ newInstance(ae) : string
+ destroyInstance(archID) : void
+ getArchElements() : string list[1..*]
+ getArchElement(aeID) : ArchitecturalElement
+ addAttachment(att) : string
+ removeAttachment(attID) : void
+ getAttachments() : string list[1..*]
+ getAttachment(attID) : Attachment
+ addBinding(bind) : string
+ removeBinding(bindID) : void
+ getBindings() : string list[1..*]
+ getBinding(bindID) : Binding

Fig. 2. Configuration Aspect

A PRISMA system is an architectural pattern that defines the type of architectural elements a system is composed of and the kind of valid connections among them. This architectural pattern is instantiated to a specific configuration in order to start the architecture execution. However, dynamic systems need to change their initial configuration several times as a result of its normal behaviour (always satisfying their architectural patterns). For this reason, the system configuration at run-time must be considered as a part of the system state. Thus, a system must have services to query its current configuration, and to change it at run-time. In this way, a system is aware of its current configuration and can decide whether to change it or not. A new concern has been created using the PRISMA Aspect-Oriented Architecture Description Language (AOADL) [6] to encapsulate these services into aspects. This is the *Configuration* concern. A *Configuration* aspect encapsulates every property and behaviour related to dynamic reconfiguration. Any system that needs reconfiguration capabilities to evolve its internal composition imports the *Configuration* aspect.

This aspect has a set of attributes that contain the current configuration of the system and a set of services that maintain and evolve this configuration. The attributes the *Configuration* aspect provides are dynamic lists which store *references* to: (i) the architectural element instances of the system at run-time (component, connector, and system instances), and (ii) connection instances of the system at run-time. The services that the *Configuration* aspect provides (see Figure 2) allow systems to know and modify the data that these attributes store (the system configuration): (i) create or destroy instances of system architectural elements, (ii) establish connections between system architectural elements, and (iii) provide query services in order to make the system aware of its current configuration. In this way, reconfiguration characteristics are specified without extending the PRISMA AOADL with new primitives, and the reconfiguration concern is not tangled with other concerns.

The infrastructure where the software architecture is running is responsible for providing to each system instance the mechanisms of dynamic reconfiguration. PRISMA software architectures are executed by the PRISMANET middleware [7], which provides the mechanisms required for knowing and modifying the running configuration of each system instance at run-time. PRISMANET guarantees that the configuration dependencies among the different elements are preserved.

4 Conclusions and Further Work

This paper has presented an approach to dynamically reconfigure software architectures taking advantage of aspect-oriented techniques. This approach consists of providing each complex component with an aspect called *Configuration*, that provides dynamic reconfiguration services at run-time. As a result, this approach has the advantage of keeping dynamic reconfiguration properties and behaviour from being tangled with the rest of the architecture. This has been done without increasing the complexity of the PRISMA AOADL, only by using its original primitives.

Autonomous system instance reconfigurations must satisfy the constraints of the system architectural pattern. We are currently working on specifying how the reflection mechanisms ensure that the requested changes do not violate the restrictions described in the architectural pattern of the system.

Acknowledgements. This work is funded by the Dept. of Science and Technology (Spain) under the National Program I+D+I, META project TIN2006-15175-C05-01. This work is also supported by a fellowship from Conselleria d'Educació i Ciència (G. Valenciana) to C. Costa.

References

1. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: WOSS’04. Proc. of 1st ACM SIGSOFT Workshop on Self-Managed Systems, Newport Beach, California, pp. 28–33. ACM Press, New York (2004)
2. Cuesta, C.E.: Dynamic Software Architecture Based on Reflection (in Spanish). PhD Thesis, Department of Computer Science, University of Valladolid, Spain (2002)
3. Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: Dynamic Coordination Architecture through the use of Reflection. In: SAC 2001. Proc. of 2001 ACM Symposium on Applied Computing, pp. 134–140 (2001)
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
5. Pérez, J.: PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, Spain (2006)

6. Pérez, J., Ali, N., Carsí, J.Á., Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 123–138. Springer, Heidelberg (2006)
7. Pérez, J., Ali, N., Costa, C., Carsí, J.Á., Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. In: Proc. of 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, pp. 97–108 (June 2005)
8. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
9. Rasche, A., Polze, A.: Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft.NET. In: ISORC'03. Proc. 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Japan, pp. 164–171 (2003)

Model-Driven Approach for Designing Industrial Control Systems

Elisabet Estevez and Marga Marcos

Automatic Control and Systems Engineering Department, University of the Basque Country,
Alda Urquijo S/N, 48013, Bilbao, Spain
{elisabet.estevez,marga.marcos}@ehu.es

Abstract. Industrial Control Systems are used in most of the industrial sectors to achieve production improvement, process optimization and time and cost reduction. Integration, reuse, flexibility and optimization are demanded to adapt to a rapidly changing and competitive market. There is also a growing requirement that all software tools that support the different phases of the development process (design, configuration, management) can be integrated as well. Thus, a consolidation of modeling methodologies for achieving this goal is needed. This paper proposes a Model-driven approach based on different views of the application for designing industrial control systems. XML schema and schematron technologies are selected for defining the domain languages and for checking their coherency and consistency.

Keywords: Industrial Control Systems, component – based modeling, XML schema, XPath, consistency analysis.

1 Introduction

Industrial control systems are specific computer systems not only because of the special characteristics of the environment they control (industrial processes) but also because they use specific equipment, such as Programmable Logic Controllers (PLCs), industrial communication systems (fieldbus), and dedicated I/O devices. On the other hand, as a computer system, its design deals with the definition of the software architecture. Traditionally, each control equipment vendor offered a development system based on proprietary software architectures and programming languages. In the last years, a great effort is being done by international organisations that have led to the definition of standards for this application field. PLCOpen [1] was founded in 1992 and it is a vendor- and product-independent worldwide association whose mission is to be the leading association resolving topics related to control programming to support the use of international standards in this field. The International Electrotechnical Commission (IEC) promotes open systems to be used in the industrial control field. To achieve that, the IEC 61131-3 standard [2], [3] proposes a software model and programming languages for Industrial Process Measurement and Control Systems (IPMCS). Most of the PLC manufacturers are doing a great effort in order to become IEC 61131-3 standard compliant.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 284 – 287, 2007.

Furthermore, as fast as industry reaches a greater maturity level and applications become more and more complex, a consolidation of methodologies, which allow system description and definition before its construction, becomes more and more necessary.

This paper presents XML based Model Driven Approach for defining an industrial control system.

2 Industrial Control System Description Languages

The design of distributed industrial control applications involves different domain fields. The *Control engineering* domain defines the functional design that contains the control strategies. The *Electronic- electrical engineering* domain deals with the selection of the equipment needed to start-up the control system. The *Software engineering* domain deals with the software architecture which implements the functional design. Furthermore, as the three 3 views represent the same application, they are not independent and they are related to each others. In [4] the main elements involved for describing these views are identified. The work defines them in terms of domain specific components and connectors.

The definition of a language for each domain view is necessary in order to express application models based in these three views. The selected technology for describing each view characteristics is XML. There are different XML standards that can be used not only for describing Industrial Control Systems, but also for performing model coherency and consistency checks e.g. XML schema [5] and schematron rules [6].

2.1 ctrlEng-ML: Control Engineering Markup Language

A general overview of the Control Engineering Markup Language is illustrated in Fig. 1. The *ctrlEng-ML* XML schema represents the architectural style of the control engineering view, defining the relationships among the elements that constitute the language.

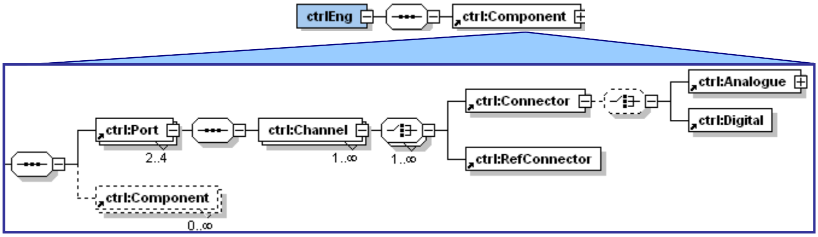


Fig. 1. General view of ctrlEng-ML.xsd

This schema is defined by as many XML schema elements as elements for describing the functionality of industrial control applications. A functional component is characterized by its name, level at the hierarchy, optionally by a set of functional requirements and at least by two Ports. Besides, it could contain a set of lower level components. 13 composition rules have been developed as schematron rules.

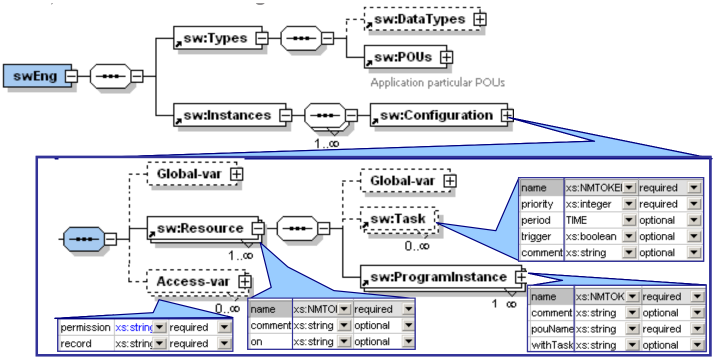
For instance, in order to characterize a component with the hierarchical level it belongs to, a new XML data type (*level*) is defined. But this new data type does not assure that the levels within the hierarchy are sequential. To check this, the following *schematron* rule has been developed.

```
<rule context="ctrl:ctrlEng/ctrl:Component">
    <assert test="current()/@level=0" priority="high">
    application functionality must start by a component with 0 level in the hierarchy
    </assert>
<rule context="ctrl:Component[@level&gt;0]">
    <assert test="current()/@level=parent::*[name()='ctrl:Component']/@level+1" priority="high">
    value of level in the Component is not correct
    </assert>
</rule>
```

Fig. 2. Checking rule for assuring sequential levels within the hierarchy

2.2 swEng-ML: Software Engineering Markup Language

A general overview of the Software Engineering Markup Language (*swEng-ML* XML schema) is illustrated in Fig. 3.



name	xs:NMTOKEI	required
priority	xs:integer	required
period	TIME	optional
trigger	xs:boolean	optional
comment	xs:string	optional

permission	xs:string	required
record	xs:string	required

name	xs:NMTOI	required
comment	xs:string	optional
on	xs:string	optional

name	xs:NMTOK	required
comment	xs:string	optional
pouName	xs:string	required
withTask	xs:string	optional

Fig. 3. General overview of swEng.xsd

The description of the software architecture has two main parts [7]: one contains the data types and the POUs defined by the programmer (*sw:Types*), and the other contains the automation project itself (*sw:Instances*). Each type of component identified for describing the software architecture is defined as a XML schema element. Additionally, a set of composition rules implemented as schematron rules have been developed. The software architecture connectors, the variables, are characterized by their visibility (as illustrated in Fig. 3), *type* and initial value. In this sense, and in order to check the correctness of initial values, the data types that the standard considers elementary have been defined in XML. They are characterized by their range of values as well as by their representation patterns.

2.3 eeEng-ML: Electronic – Electric Engineering Markup Language

A general overview of the Electronic-Electric Engineering Markup Language (*eeEng-ML* XML schema) is illustrated in Fig. 4.

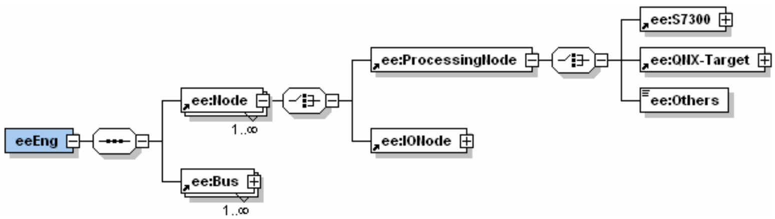


Fig. 4. General view of eeEng-ML.xsd

3 Conclusions

Three different views have been identified as the domains involved in the design of such type of systems. An implementation using XML technologies for the three description languages have been proposed. These XML-based descriptions are going to be used for supporting the development cycle of the application. More detail about the description markup languages can be found in http:// www.disa.bi.ehu.es/gcis/projects/merconidi/icsML.

The XML stylesheet technology jointly with DOM can be used to integrate the set of COTS tools involved in the application design. In fact, these tools will generate/consume the different domain models. In summary, the use of open software technologies has been proven to be directly applicable in other application fields.

Acknowledgements. This work has been supported by MCYT&FEDER under project DPI 2006-4003.

References

1. PLCopen. Available at: http://plcopen.org/
2. Lewis, R.W.: Programming Industrial Control Systems using IEC 1131-3. IEE Control Engineering Series (1998)
3. John, K.-H., Tiegelkamp, M.: IEC1131-3: Programming Industrial Automation Systems. Springer, Heidelberg (2001)
4. Estévez, E., Marcos, M., Sarachaga, I., Orive, D.: A Methodology for Multidisciplinary Modeling of Industrial Control Systems using UML. In: Proc of the 5th International Conference on Industrial Informatics, Austria, Viena (July 2007)
5. Van der Vlist, E.: XML schema, ed. O'REILLY (2002)
6. Rick Jelliffe schematron rules. Available at http://www.ascc.net/xml/schematron/
7. PLCopen TC6. Available at: http://plcopen.org/TC6/XML_Intro.htm

Informed Evolution

Katrina Falkner[1], Dharini Balasubramaniam[2], Henry Detmold[1], and David S. Munro[1]

[1] School of Computer Science, University of Adelaide, Adelaide, S.A. 5005, Australia
{katrina,henry,dave}@cs.adelaide.edu.au
[2] Department of Computer Science, University of St Andrews, St Andrews, Fife
KY16 9SX, UK
dharini@dcs.st-and.ac.uk

Abstract. Ageless Software evolves, to meet new requirements, without reducing its efficiency or understandability. Here we introduce a methodology called Informed Evolution for supporting the construction and evolution of ageless software. This methodology integrates the software architecture (structure and constraints) and the system implementation (behaviour) within system execution. Evolution is effected by evolution patterns which are in turn guided by constraints specified in the software architecture. The availability of the software architecture and implementation at run-time ensures that changes are informed by design and implementation decisions, thus preserving efficiency and understandability. In this paper, we outline Informed Evolution, and describe how evolution patterns may be expressed for systems developed using this methodology.

1 Introduction

When software evolves, the set of design considerations of the original system is generally unknown to the software team performing the required modifications. These design considerations include not only the current state of the system but, also, how it should be evolved. Changes that are made without this knowledge can age the software system and introduce complexity in future understanding and modification [1]. Ageless Software is inherently designed to evolve to new requirements or environmental changes. An ideal in software development, ageless software requires that changes are made to a software system without reducing its efficiency or understandability. For efficiency, evolution should not produce performance degradation due to unidentified interactions and dependencies within the software system. For understandability, the design of the software system must be at least as clear as when it was first produced.

Our approach is based on integrating the software architecture (for structure and constraints) and the software implementation (for behaviour) within the system execution. The advantages of such integration are that design and implementation decisions are available at run-time to guide the software evolution process, and that the architecture can evolve in step with the system.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 288–291, 2007.

In this paper, we introduce a methodology called Informed Evolution for constructing and evolving ageless software systems. It makes use of *evolution patterns* to effect coordinated changes to the system. Evolution patterns define a sequence of software changes that can be applied (and reapplied) to adapt the software system in a fashion that preserves evolvability, and are guided by the constraints in the software architecture. Informed Evolution provides a systematic means for describing changes to be made to a software system, and a systematic means for linking these changes to the software architecture.

2 Informed Evolution: A Methodology

Informed Evolution supports the integrated specification of software architecture and software implementation. Evolution patterns, defined as part of the software architecture, can access both at run-time in order to co-ordinate changes to both behaviour and structure of a software system.

Structural and behavioural constraints specified in the software architecture are employed by evolution patterns in understanding the executable software system. Specification of software architecture and software implementation is an iterative process, operating over an integrated representation. When this representation is compiled into an executable form, behavioural constraints are compiled into the running program. Structural constraints form annotations made to the executable system.

Violation of a constraint identifies the need for change, resulting in the invocation of an evolution pattern to coordinate the required change. Prior to evolution, the component to be changed must be decomposed, in order to suspend its execution and provide access to internal components, and connections between those components [2]. In its decomposed form, reified representations of the software architecture and software implementation are accessible to evolution patterns. Evolution patterns reflectively alter the structure and behaviour of the component, evolving both software architecture and behaviour at run-time. Subsequent to evolution, the system is recomposed to form an executable system. During evolution, constraints are used to validate any changes made by the evolution pattern. If violated, these constraints may also invoke further evolution patterns to coordinate any required change either in the implementation, or in the software architecture itself.

For example, the specification of a condition setting a bound upon hardware resource utilisation (a behavioural constraint) may be associated with an evolution pattern to redistribute the component system so as to enable access to an increased number of resources. In this example, a condition specifies co-location constraints upon some of the components (a structural constraint). Both constraints and an evolution pattern associated with the behavioural constraint are defined as part of the software architecture of the component system. The behavioural constraint is compiled into the behaviour of the component system so that, when breached, an evolution pattern to rebalance the allocation of components across the system is invoked. Software architecture constraints also evolve upon the application of the evolution pattern, in that the constraint bounding resource utilisation is changed to reflect the change in available resources.

The overall system model is similar to that expressed in ArchWARE [2]. Software systems are modeled as multiple interacting software components, where each component has a *controller* that effects change, via evolution patterns.

3 Describing Evolution Patterns

The Informed Evolution methodology is based on defining appropriate support for the specification of evolution patterns, using an evolution control API. The API defines operations to traverse a reified representation of the software system (both software architecture and implementation), apply changes to this representation, and compose operations to create evolution patterns. The process of applying an evolution pattern consists of four phases: identification, decomposition, evolution and composition. Each of the four phases models its activity in terms of operation over a component graph model of the software system. The evolution control API provides operations to extract this component graph, obtaining a reified representation of the software system, and to traverse the component graph, inspecting the representation (see Table 1). These operations are invoked by the component coordinating the application of the pattern.

Table 1. Operations provided by the Evolution Control API

locate	locates a component(s) that matches the provided component graph.
decompose	suspend component execution, and make internals accessible.
extract	extract the reified software architecture as a component graph.
evolve	apply evolution pattern if possible.
compose	composes provided components, applying structural constraints.

The *evolve* operation applies an evolution pattern to a component graph model; it is parameterised by an evolution pattern that defines a sequence of changes to be made to the component graph. Change is applied at the granularity of components, their interactions (connections) and properties associated with each. Evolution patterns apply changes to a component graph model, based on matching properties of components (or connections), or expressions that indicate paths through the component graph. Accordingly, evolution patterns are parameterised by a component graph, a set of properties, and a path specification. The evolution operation is also parameterised by the information needed by the evolution pattern.

The requirements for supporting our methodology in a target language are:

- Representation of the software system as a component graph
- Integration of a reified representation of the software architecture and system source as part of the component graph.
- Implementation of the given API for traversing, modifying and defining evolution patterns over this representation.

We have developed a prototype of this work using the ACME ADL [3] to specify software architecture. At this stage, we are defining the reflective and generative programming rules required to support evolution using our API.

4 Related Work

Software architecture is typically represented as a set of software artifacts which are separate from the software implementation, meaning that constraints upon composition or execution of the software system have no direct affect upon execution of the software system. Integrated environments, such as Plastik [4] and ArchWARE [2], support coordinated evolution of the software architecture with the software implementation. Limited work has been done within ArchWare in defining a proof of concept evolution pattern [5]. Building upon this work, Informed Evolution provides a framework for the systematic description of change.

5 Conclusions

Informed Evolution supports the integrated specification of software architecture and software implementation to address the challenge of constructing ageless software. We define evolution patterns as systematic descriptions of change, specifying how to change a software system with access to both the software architecture and software implementation at run-time. In this paper, we have described our approach to defining evolution patterns, and the general system requirements our approach places upon the executing software system.

Acknowledgements

This work was partially supported by an EPSRC Visiting Research Fellowship EP/E009212/1 for Drs. Falkner and Munro whilst at St Andrews. We would like to thank Ron Morrison for his comments and assistance with this project.

References

1. Parnas, D.L.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering, pp. 279–287 (1994)
2. Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., Occhipinti, C.: ARCHWARE: architecting evolvable software. In: Proceedings of the First European Workshop on Software Architecture, pp. 257–271 (2004)
3. Garlan,D.,Monroe,R.T.,Wile,D.:ACME:ArchitecturalDescriptionofComponent-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
4. Joolia, A., Batista, T., Coulson, G., Gomes, A.T.: Mapping adl specifications to an efficient and reconfigurable runtime component platform. In: Working IEEE/IFIP Conference on Software Architecture (WICSA) (2005)
5. Mickan, K., Morrison, R., Kirby, G.: Using generative programming to visualise hypercode in complex and dynamic systems. In: Australian Computer Science Conference, pp. 377–386 (2004)

Using Connectors to Model Crosscutting Influences in Software Architectures*

Lidia Fuentes, Nadia Gámez, Mónica Pinto, and Juan A. Valenzuela

Dpto. Lenguajes y Ciencias de la Computación, University of Málaga, Spain
{lff,nadia,pinto,valenzuela}@lcc.uma.es
http://caosd.lcc.uma.es/

Abstract. AO-ADL is an aspect-oriented architecture description language where 'crosscutting' becomes a new kind of relationship between components. The semantic of connectors is extended in order to represent such crosscutting relationships. In this paper we focus on an important contribution of AO-ADL, its mechanism for defining aspect-oriented connector templates, which capture generic and reusable crosscutting influences, providing an aspect-oriented architectural pattern.

1 Introduction

During the last years, several aspect-oriented (AO) architecture description languages (ADLs) have been defined [1,2,3,4,5,6]. Two of the most relevant issues to be considered by these ADLs are: (1) the *decomposition model* used to separate and represent crosscutting and non-crosscutting concerns, and (2) the *composition model* used to specify the composition between them.

Our approach, the AO-ADL language [6], considers a *symmetric decomposition model* in which both non-crosscutting and crosscutting concerns are modeled by components. This symmetry clearly affects how the *composition model* is defined, since apart from composing base components, AO-ADL must define a new relationship for weaving aspectual components. Likewise coordination and communication are usually modeled by connectors in ADLs [7], connectors seems to be the natural place to specify such crosscutting relationships [6].

In this paper we focus on that the relationships between crosscutting concerns and the rest of architectural components are sometimes recurrent in different applications. Thus, it may be useful to have an ADL that provides mechanisms to capture generic and reusable crosscutting influences. AO-ADL allows the specification of recurrent crosscutting relationship by means of *connector templates*. These connector templates are then instantiated in particular software architectures. After this introduction, section 2 describes the main features of AO-ADL, of the connector templates and of their development using Eclipse and JET. Our conclusions and future work are presented in section 3.

* Work supported by European Projects IST-2-004349 (AOSD-Europe) and IST-033710 (AMPLE), and Spanish Project TIN2005-09405-C02-01.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 292–295, 2007.

2 AO-ADL Connector Templates

For certain concerns, the same pattern of interaction is recurrent independently of the domain application (this may occur for either crosscutting or non-crosscutting concerns, though in this paper we focus on crosscutting concerns).

Thus, connector templates have to be seen as AO architectural patterns that allow modeling generic and reusable crosscutting influences. In their definition we combine the vision in [7], of defining certain concerns within the specification of connectors, with the special mechanisms offered by AOSD to represent and compose crosscutting and non-crosscutting concerns. Concretely, we define connector templates as a contribution of the AO-ADL language [6].

In the rest of this section, we begin briefly describing the main features of AO-ADL. Then, we illustrate our approach by defining a connector template for modeling concurrency. Finally, we describe how to automate the definition and instantiation of connector templates.

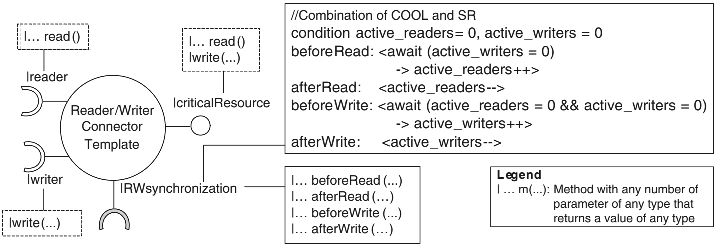
2.1 The AO-ADL Language

The main elements of AO-ADL are components and connectors. The specification of AO-ADL components is similar to other ADLs [8]. Connectors, in comparison with other ADLs, have been extended with: (1) a new kind of *role* named aspectual role – components attached to this kind of role behave as 'aspectual' components, and (2) a new kind of *bindings specification* named aspectual bindings – specification of the connections between 'aspectual' and 'base' components. A detailed specification of AO-ADL can be found in [6].

2.2 The Concurrency Concern

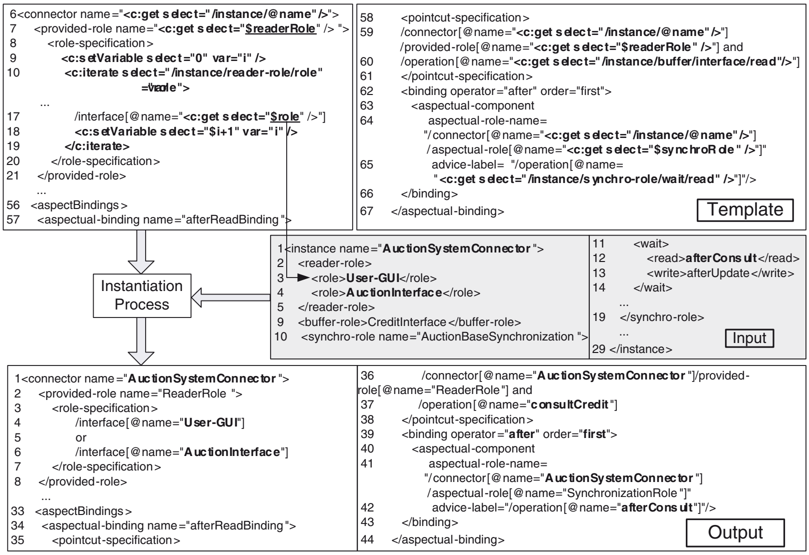
A recurrent crosscutting concern identified in many different case studies is concurrency. Concretely, by analyzing the requirements specifications of these case studies, we would obtain the following information: (1) in an auction system, concurrency influences 'users', 'auctions' and 'credits'; in an ATM case study, concurrency influences 'users', 'banks' and 'accounts'; (2) the critical resources being accessed concurrently are the 'credits' and the 'accounts', respectively, and (3) in all the case studies there is going to be several components reading information associated to the critical resource, and several components updating that information. This can clearly be identified as a reader/writer synchronization problem, since in spite of the different vocabulary used in the case studies they all have to cope with the same problem.

In this context, the software architect can use the connector templates provided by AO-ADL to model the reader/write synchronization problem (see fig. 1). Notice that the connector roles are parameters of the template (|reader, |writer, |criticalResource and |RWSynchronization). Additionally, these parameterized roles specify a list of parameterized operations that will also have to be instantiated. The details of how to instantiate these parameters are provided below. Notice that these parameters allow the specification of reusable pointcuts, which



```
//Combination of COOL and SR
condition active_readers= 0, active_writers = 0
beforeRead: <await (active_writers = 0)
                    -> active_readers++>
afterRead:   <active_readers-->
beforeWrite: <await (active_readers = 0 && active_writers = 0)
                    -> active_writers++>
afterWrite:    <active_writers-->
```

Fig. 1. AO-ADL Architectural Synchronization Pattern



```
<connector name="<c:get select="/instance/@name"/>">
  <provided-role name="<c:get select="$readerRole" /> ">
    <role-specification>
      <c:setVariable select="0" var="i" />
       <c:iterate select="/instance/reader-role/role"
                        var="role">
   ...
         /interface[@name="<c:get select="$role" />"]
         <c:setVariable select="$i+1" var="i" />
       </c:iterate>
     </role-specification>
  </provided-role>
  ...
<aspectBindings>
  <aspectual-binding name="afterReadBinding">
    <pointcut-specification>
    /connector[@name="<c:get select="/instance/@name"/>"]
    /provided-role[@name="<c:get select="$readerRole" />"] and
    /operation[@name="<c:get select="/instance/buffer/interface/read"/>"]
    </pointcut-specification>
    <binding operator="after" order="first">
      <aspectual-component
          aspectual-role-name=
         "/connector[@name="<c:get select="/instance/@name"/>"]
          /aspectual-role[@name="<c:get select="$synchroRole" />"]"
           advice-label=  "/operation[@name=
           "<c:get select="/instance/synchro-role/wait/read" />"]"/>
    </binding>
  </aspectual-binding>
```

```
<instance name="AuctionSystemConnector">
  <reader-role>
    <role>User-GUI</role>
    <role>AuctionInterface</role>
  </reader-role>
  <buffer-role>CreditInterface</buffer-role>
  <synchro-role name="AuctionBaseSynchronization">
      <wait>
        <read>afterConsult</read>
        <write>afterUpdate</write>
      </wait>
    ...
  </synchro-role>
    ...
</instance>
```

```
<connector name="AuctionSystemConnector">
  <provided-role name="ReaderRole ">
    <role-specification>
         /interface[@name="User-GUI"]
         or
         /interface[@name="AuctionInterface"]
    </role-specification>
  </provided-role>
  ...
<aspectBindings>
  <aspectual-binding name="afterReadBinding">
    <pointcut-specification>
          /connector[@name="AuctionSystemConnector"]/provided-
role[@name="ReaderRole"] and
          /operation[@name="consultCredit"]
    </pointcut-specification>
    <binding operator="after" order="first">
      <aspectual-component
          aspectual-role-name=
         "/connector[@name="AuctionSystemConnector"]
          /aspectual-role[@name="SynchronizationRole"]"
           advice-label="/operation[@name="afterConsult"]"/>
    </binding>
  </aspectual-binding>
```

Fig. 2. Readers-Writers Instantiation

determine the join points to which the Synchronization component is attached, as well as the advices to be injected. Finally, we have specified the semantic of the reader/writer synchronization mechanism. During the instantiation process, this information is automatically propagated to the connector instance.

2.3 Tool Support

Among the several approaches available to implement templates, we have decided to use the Java Emitter Templates (JET)[1]. In fig. 2 we partially show the example of the reader/writter synchronization template developed in JET. In the upper part of the figure, part of the template for the readers/writers synchronization problem is shown. We have written in bold letters the JET code. This code represents the data that can be modified for adapting the generic template to our particular application domain.

[1] Web Site: www.eclipse.org/emft/projects/jet/

For showing the instantiation process we focus on an auction system. We have identified the components that would play the role of the readers/writers, the component that would play the role of buffer, and so on. For example, we have identified two components playing the reader role: the User-GUI and the Auction components. This information is provided in the input file (input, lines 3 and 4). The template iterates over all the readers (template, lines 10-19), adding the corresponding information to the output (output, lines 4-6). Notice that the only information provided by the software architect is the one in the grey box.

3 Conclusions and Future Work

This paper have presented the concept of connector template and its usability in the modeling of generic and reusable crosscutting influences. We have shown the viability of this approach by describing connector templates for the concurrency crosscutting concern. Additionally, we have shown how the definition and instantiation of connector templates can be automated using Eclipse and the JET module, available within the Eclipse Modeling Framework. As part of our ongoing work we need to define a catalogue of templates, analyzing as much crosscutting concerns as possible in order to verify that our approach is generic enough to cope with different and heterogeneous representation of concerns.

References

1. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A model for developing component-based and aspect-oriented systems. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 259–274. Springer, Heidelberg (2006)
2. Pinto, M., Fuentes, L., Troya, J.M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 118–137. Springer, Heidelberg (2003)
3. Garcia, A., et al.: On the modular representation of architectural aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, Springer, Heidelberg (2006)
4. Pérez, J., et al.: PRISMA: towards quality, aspect-oriented and dynamic software architectures. In: 3rd IEEE Intl Conf. on Quality Software (2003)
5. Navasa, A., Pérez, M.A., Murillo, J.: Aspect modelling at architecture design. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 41–58. Springer, Heidelberg (2005)
6. Pinto, M., Fuentes, L.: AO-ADL: An ADL for describing aspect-oriented architectures. In: Early Aspect Workshop at AOSD 2007 (2007)
7. Mehta, N., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: 22nd ICSE'00, Ireland, pp. 178–187. ACM Press, New York (2000)
8. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transaction on Software Engineering 26(1), 70–93 (2000)

From Mobile Business Processes to Mobile Information Systems[*]

Volker Gruhn and Clemens Schäfer

Chair for Applied Telematics / e-Business[**]
University of Leipzig, Germany
{gruhn,schaefer}@ebus.informatik.uni-leipzig.de

Abstract. We suppose a methodology and a middleware to build information systems that support mobile business processes. Our approach allows applications to make use of a self-mobilizing code paradigm, i.e. the ability of components to be distributed to devices depending on internal and external changes.

The methodology is aimed at deriving architectural decisions from the business processes to be supported. We assess processes and their implications for mobility concerns and make use of architecture simulation to check feasibility and quality of service of such applications at design time and runtime.

1 Motivation

Analyst forecasts show that there is a growing market for mobile information systems in the next years: The market for mobile applications is expected to grow due to higher number of 3G subscribers and new pricing models. Van Veen [5] predicts an approximately 100% growth from 2006 until 2010 for 3G subscribers. 87% of investments in mobile applications will focus on business to employee (B2E) systems. In Central and Eastern Europe mobile intranet and extranet business will increase from 0 in 2006 to 10 billion Euro in 2012 according to [1]. All these forecasts can be taken as an indicator for an increasing demand for mobile applications in the B2E area.

In the past, business cases for mobile systems often failed due to several reasons. Rapid innovation cycles for mobile devices and growing user expectations make it necessary to use a mobile application with mobile devices which have not been existing at design time. This requires proactive management of uncertainty during mobile system development–and makes the development an intrinsically complex and conflictive task.

Furthermore, mobile business processes are more likely to change over time than non-mobile business processes. This requires flexible support by mobile systems and the anticipation of process changes. Todays design approaches do not sufficiently focus on such mobile business processes.

[*] This project is funded by Deutsche Telekom Laboratories.

[**] The chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 296–299, 2007.

Development of mobile communication networks in future, especially area coverage with broadband access, is uncertain, requiring adaptive solutions which are efficient to be developed and maintained.

All these challenges can be subsumed in the demand for flexible deployment of code and data during runtime, avoiding "wrong" distributions which negatively affect a system's usability. Consequently, the major goal of our work is to allow dynamic re-arrangement of code and data during runtime, opposed to the "traditional" design-time-only deployment determination approach.

2 The MobCo Approach

Our approach called MobCo (for mobile code) consists of several building blocks, which are all subject to ongoing and further research.

2.1 Mobile Middleware

There are many middlewares for mobile code, both academic and commercial ones. For our purpose, a middleware should be widely deployable and has to fulfill a set of core requirements: As MobCo aims to mobilize code even across platforms, support for a great number of platforms is necessary. Also, disconnected operation as well as replication and conflict solving strategies for data are needed. Furthermore the support of installation, deployment, and updates of mobile components are required. These key criteria are supplemented by possible self management strategies, as well as the awareness of location and other contextual information. Evaluation of market surveys like [3] showed that such a middleware is not available (neither commercial nor academic) at the moment.

2.2 Reconfiguration Triggers

It is up to the MobCo middleware to provide an infrastructure such that the software system can support the business process as good as possible. For this purpose, the middleware has to react to the occurrence of several events (called reconfiguration triggers) that can result in a change of context for the software system as a whole. Occurrence of these triggers can–according to rules and policies– force the MobCo middleware to reconfigure the architecture of the distributed software system at runtime. This can be done by migrating code and data or by reconfiguring some of the software components.

There might be reconfiguration triggers, which change the system in response to changes in location and proximity of mobile devices, available network connectivity, other devices available at a location and dependencies between business processes and location. Triggers for system reconfiguration may also depend on business processes or software. When flexible business processes are supported, system reconfiguration can be changes by the new business process. Additionally, system reconfiguration triggers can be used to propagate (regular) software updated through the system. Availability of resources can also influence the reconfiguration of systems. Besides network resources, computing and storage resources and assignments between users and devices can be evaluated to determine software deployment. Another means to determine software distribution can be user profiles, user rights and user preferences, which all can be evaluated to determine appropriate software development, and their changes can trigger reconfiguration of the system. Time dependencies between business processes, actions and requirements for data freshness can also guard the reconfiguration of the system.

2.3 Runtime Evaluation and Self-management

A system making use of reconfiguration triggers has to collect all related status information. This information will be evaluated by the triggers. However, to avoid local optimizations of the system, we foresee a central instance supporting the actual reconfiguration of the system. Validating that mobilization of code detected by the triggers contributes to overall system performance and user satisfaction can be provided by simulating the system. Thus, determining which changes to the system indicated by mobilization triggers are suitable is possible (see Figure 1).

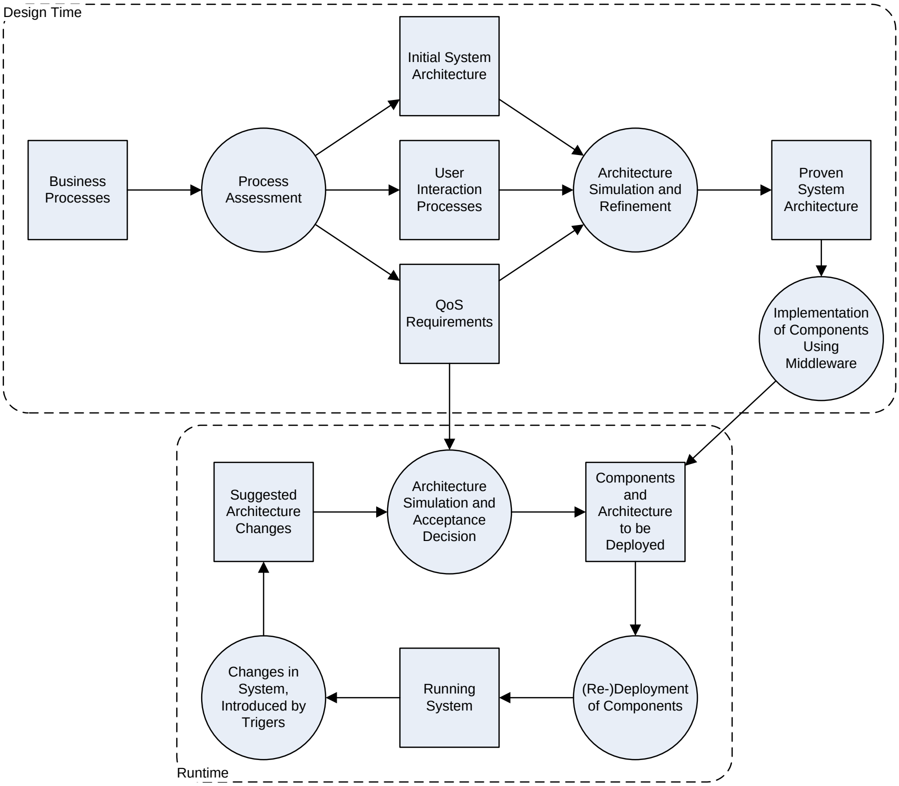

Design Time
Initial System Architecture
Business Processes
Process Assessment
User Interaction Processes
Architecture Simulation and Refinement
Proven System Architecture
QoS Requirements
Implementation of Components Using Middleware
Suggested Architecture Changes
Architecture Simulation and Acceptance Decision
Components and Architecture to be Deployed
Changes in System, Introduced by Trigers
Running System
(Re-)Deployment of Components
Runtime

Fig. 1. MobCo design time and runtime

2.4 Methodical Approach

For the development of mobile applications we plan to close the gap between business process and architecture modeling. We suppose to create a process assessment scheme to derive an initial architecture from business processes. First steps into this direction have been presented in [2]. The idea is that from a business process model a first model of the information system can be derived together with the interaction of the users with the system and the necessary QoS requirements. This information can be used as input parameters of a simulation of the mobile systems architecture as presented in [4].

After a first architecture has been determined, the system can be implemented using the middleware. During runtime triggers will stimulate changes in system architecture which undergo simulation to check their validity. Outcome of this runtime simulation is whether changes in the system's architecture are to be applied or discarded.

3 Expected Results

There will be two main contributions to the state of the art from the project MobCo. The platform-independent mobile middleware allows self-mobilizing code by means of reconfiguration triggers, whereby the influence of the triggers is evaluated by system simulation during runtime. Additionally, methods for the design process will allow developers to detect mobile parts of business processes and transform them methodically in a mobile architecture.

There are many open research questions like how to build such a middleware with feasible effort, the appropriate selection reconfiguration triggers and the practical implementation of a runtime simulation evaluation of mobile system.

By these results we believe in allowing developers to build mobile systems which are in future more successful than systems have been in the past.

References

1. Dicks, D.: The Evolution of High-Speed Mobile Data Services and Applications in CEE. Analysys (2006)
2. Gruhn, V., Köhler, A.: Aligning System Architectures on Requirements of Mobile Business Processes. In: Proceedings of the IASTED International Conference on Software Engineering (2007)
3. Mascalo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware. In: Networking 2002 Tutorials, Springer, Heidelberg (2002)
4. Schäfer, C.: Modeling and Analyzing Mobile Software Architectures. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, Springer, Heidelberg (2006)
5. van Veen, N.: European Mobile Forecast: 2005 To 2010. Forrester Research (2005)

An Architectural Model for Small-Scale Component-Oriented Frameworks

Sérgio Lopes, Adriano Tavares, João Monteiro, and Carlos Silva

Department of Industrial Electronics of University of Minho
Campus de Azurém, 4800-058 Guimarães, Portugal
{sergio.lopes,adriano.tavares,joao.monteiro,
carlos.silva}@dei.uminho.pt

Abstract. Frameworks are an important form of reuse. However, they are often complex and hard to understand, what limits their success as a reuse option. To answer this problem, it has been widely recognized the need to effectively communicate frameworks and provide appropriate tool support, but difficulties still endure.

We argue that the properties of frameworks are another aspect that is decisive for reuse problems and has not been sufficiently explored. We discuss these issues and we propose a framework architectural model that can be used to design frameworks that are easier to reuse.

Keywords: Component-oriented design, frameworks, object-oriented design and programming, role modelling, software architecture, software reusability.

1 Introduction

Frameworks are one of the dominant industry practice technologies. A framework is an incomplete design that factors commonalities and variabilities of a specific domain of application. It provides implementation for the ready-to-use common elements (also called *frozen spots*) and localizes variability at predefined abstract *variation-points* (or *hotspots*). Framework design is more complex and abstract than usual programs. Consequently, frameworks are often hard to understand and difficult to reuse.

In order to handle this problem, it has been widely recognized that it is necessary to effectively communicate them and provide appropriate tool support for instantiation activities. In spite of advances in these directions (e.g. [1], [2]), difficulties still endure [3]. Previous works do not consider an aspect that we find vital: the impact of framework properties on the difficulties of describing, understanding and instantiating frameworks. Moreover, the framework properties are decisive, because both the documentation format and the level of tool support depend on them.

Programming and design paradigms determine framework characteristics that are decisive to the problems and complexities that can arise when reusing frameworks. For example, *whitebox* frameworks need to be described in more detail than *blackbox*, and therefore, they are likely more difficult to understand. Since the former expose more details and are adapted by inheritance, they are more susceptible of design

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 300 – 303, 2007.

breaking than the latter. Another example is that smaller pieces of software are more reusable than large size ones [5], and they are also less prone to functionality overlap.

Traditional whitebox versus blackbox designs fall short in providing both reuse ease and flexibility. More elaborated models are necessary to combine positive aspects of different techniques. A framework model is described in [8], but it is more of a design and development model for in-house reuse, than a framework architectural model for third party reuse. It includes a whitebox adaptation perspective using inheritance, and leaves aside how application specific components relate to the kinds of variation-points and whether these are predefined or not. We propose a different framework model that combines a set of properties that help to facilitate framework reuse.

2 The Framework Model

2.1 Framework and Component(s)

The proposed framework concept is inspired by the idea of *framelets* [1]. We also consider frameworks to be small and not assuming exclusive control of applications' execution flow. However, contrary to framelets, we do not define an explicit size limit for frameworks (it is very dependent on the addressed problem or application domain) and they can be used independently (not necessarily integrated in a family).

Frameworks that do not assume full control must provide means for its operation to be controlled externally. This is crucial for the problem of software composition. We consider frameworks that, just like components, provide services and mechanisms to control them (e.g., initiating a task or an operation cycle). As illustrated in figure 1, *framework* inherits from both *package* and *component*, which means that it is independently deployed as a package to be reused by third parties as a component.

We consider the framework to be constituted by *internal components* (see fig.1. In the context of object-oriented languages, internal components are object-oriented components. They are not full-fledged components [5], because they are not independently deployable, but they are reused as-is in the context for which they were developed (i.e. the framework). An internal component is a class or a few classes, represented by a concrete class, as depicted in fig.1. The component is instantiated by instantiating its representative class, which creates and hides other objects that constitute the internal component.

2.2 Variation-Points and Service-Points

Frameworks that communicate with application specific components solely through inversion of control (*calling*) do not provide services to control its execution. On the other hand, frameworks that offer only services to clients (*called*, or class libraries), cannot be tailored to application specific needs in a disciplined way. Therefore, to provide effectively both composability and adaptability, two key features for reuse, a mixed architecture is required.

To support this perspective, we introduce the concept of *service-points* – parts of the framework interface that are designed to be called by clients. The service-points are anticipated by framework developers and identify services offered to enable external control of its operation. As illustrated in fig.1, the proposed framework model combines *variation-points* for adaptability, and service-points supporting composition. A variation-point contains a *variation interface*, i.e. a required interface that expresses the obligations of the adaptations. A service-point comprises a provided *service interface* and a corresponding required *client interface*. The client interface is optional, because service-points do not always need to impose constraints on clients.

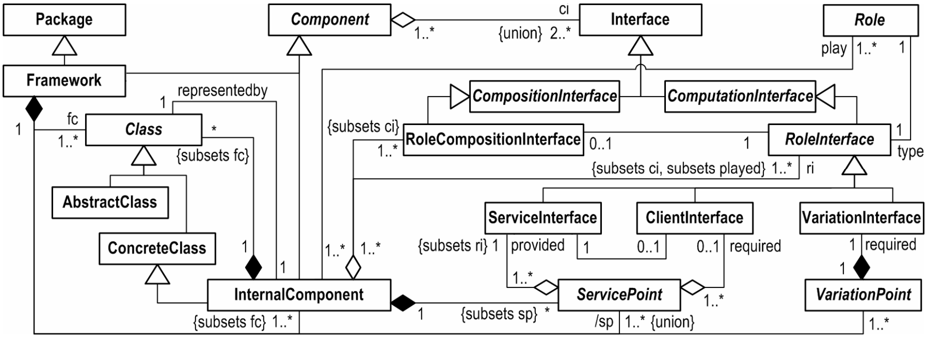


Fig. 1. Framework architectural model

2.3 Object Composition and Blackbox Reuse

Both class inheritance and object composition have advantages and disadvantages. Our framework is based on object composition with message forwarding via interfaces because it offers the necessary variability for anticipated reuse and facilitates reuse. For unanticipated reuse, application developers can provide their own implementations of internal components. The relationships between internal components are object relationships (association and aggregation). Inheritance remains as a useful mechanism to reuse implementation inside internal components. Moreover, our concept of blackbox ignores whether framework designers or application developers are the ones that provide the implementations of required variation interfaces.

2.4 Exploiting Role Modelling

The *role-modelling* technique [6] is based on object relationships, making a perfect match with our internal components. Comparatively to class modelling, it brings two very important advantages for framework reuse: extra flexibility in implementing object collaborations (see [8]), and a more natural way of defining restrictions upon clients of a class (see [7]). A combination of role and class modelling has been formalized at design level for whitebox framework reuse with adaptation through inheritance[7]. Our model defines the application of role modelling at implementation level to blackbox frameworks adapted through object composition.

Internal components provide separate *computation* and *composition interfaces*, as illustrated in fig.1. All computation interfaces are *role-interfaces*, i.e. type a role, including variation interfaces, service interfaces and client interfaces. An internal component *Ca* has one *role interface* for each role *Ra* it plays to another internal component *Cb*. If the object relationship from *Cb* to *Ca* is navigable, *Cb* has a *role composition interface* corresponding to role *Ra*.

3 Conclusion

Our model proposes a novel integration of different techniques (components, framelets and role modelling), adapting and/or extending them. It also includes a different perspective on the blackbox framework concept.

The proposed framework model enables a blackbox approach for anticipated reuse that offers adaptability and composability, exclusively based on calling provided interfaces and implementing required interfaces. It defines component-oriented frameworks, constituted by internal components that are the building blocks of applications. This way it is possible to divide the framework instantiation in two smaller and more systematic activities: provide implementations for required interfaces – *framework adaptation* – and compose internal components – *application instantiation*. We introduce the concept of service-points, to support use relationships with restrictions on clients. The use of role interfaces allows a more flexible and fine-grained control-flow composition, critical for interconnecting components and integrating several frameworks. All these features help to improve reusability and reduce difficulties.

References

1. Oliveira, T., Alencar, P., Cowan, D.: Towards a declarative approach to framework instantiation. In: Proc. of the Workshop DMP of the 17th IEEE ASE (2002)
2. Lopes, S., Tavares, A., Monteiro, J., Silva, C.: Describing Framework Static Structure: promoting interfaces with UML annotations. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 54–61. Springer, Heidelberg (2006)
3. Kirk, D., Roper, M., Wood, M.: Identifying and Addressing Problems in Framework Reuse. In: Proc. of the 13th International Workshop on Program Comprehension (2005)
4. Pree, W., Koskimies, K.: Framelets - small and loosely coupled frameworks. ACM Computing Surveys 32(1) (2000)
5. Szyperski, C.: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley/ACM Press (2002)
6. Reenskaug, T., Wold, P., Lehne, O.: Working with objects: The OOram Software Engineering Method. Manning/Prentice-Hall (1996)
7. Riehle, D., Gross, T.: Role model based framework design and integration. In: Proc. of the 13th ACM OOPSLA'98, pp. 117–133 (1998)
8. Gurp, J., Bosch, J.: Design, implementation and evolution of object oriented frameworks: concepts and guidelines. Software Practice & Experience 31(3), 277–300 (2001)

UML Profile for the Platform Independent Modelling of Service-Oriented Architectures

Marcos López-Sanz, César J. Acuña, Carlos E. Cuesta, and Esperanza Marcos

Computer Languages and Systems II Department
Rey Juan Carlos University - Mostoles (Spain)
marcos.lopez@urjc.es, cesar.acuna@urjc.es,
carlos.cuesta@urjc.es, esperanza.marcos@urjc.es

Abstract. The vast diversity of implementation and support platforms for service-oriented architectures (such as Web, Grid or even CORBA) increases the complexity of the development process of service-based systems. To reduce it, both the architectural properties of the SOC paradigm and a development approach based on the MDA proposal can be studied. This work describes a UML profile for the PIM-level service-oriented architectural modelling, as well as the correspondent metamodel. PIM (Platform Independent Model) level is chosen because it does not reflect constraints about any specific platform or implementation technology. The proposal sketched in this article is part of our research of a service-oriented development method (SOD-M) called MIDAS.

Keywords: Service-Oriented Architecture, Model-Driven Architecture, PIM-level modelling, UML Profiles.

1 Introduction

In the last years the development of systems based on services has grown in importance. However, several problems have come up along with this evolution. Issues like the migration of execution platform, the increase in the complexity of the development process and the lack of a precise definition of the concepts involved in SOSA (Service Oriented Software Architecture) solutions are among them.

To tackle these problems we aim at the definition of a service-oriented development method (SOD-M) based on the MDA principles and architecture-centric. To achieve this goal we follow two different action lines:

1. To accomplish the study of the architectural principles governing the service-oriented (SO onwards) designs as the architecture of a system reflects the structure and behaviour of a system and how it evolves as time elapses.
2. To follow a methodological approach to reduce the complexity of the SO development process. The ideas behind the MDA (Model-Driven Architecture) [1] proposal can facilitate and improve the development of SO solutions.

Following the conclusions of previous research works [2]; in this article we focus on the definition of the main elements of service architectures at the PIM level of the MDA proposal. To do so, we present a UML profile for the PIM-level service-oriented architectural modelling, together with the correspondent metamodel.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 304 – 307, 2007.

2 Service Architecture Within a MDA Framework

The research work presented in this article is part of MIDAS [2], a methodological framework for the development of information systems based on MDA. This framework is defined by three orthogonal dimensions reflecting, respectively, the MDA abstraction levels, modelling the separation of concerns in information system development and defining cross-cutting aspects to the previous dimensions, such as the architectural aspect.

The architecture is considered to be the driving aspect of the development process. It allows to specify which aspects and models in each level are needed according to the software architecture design.

The architectural models are divided into two levels which correspond to the PIM and PSM levels. The reason to make this division is that with an architectural view of the system at PIM-level we ensure that there are no technology or implementation constraints in the model, besides facilitating the possibility of establishing different PSM-level models. Moreover, and applying this conception to a system following the Service-Oriented Computing (SOC)[6] paradigm, the services that appear at the PIM-level architecture model will be understood as conceptual services, classified depending on the role or functionality given and the entities that group those services.

3 Proposal of PIM-Level UML Profile for SOSA

The set of concepts that appear in the proposed UML profile are enumerated next. Table 1 lists the most relevant stereotypes defined as well as the base UML metaclass from which they derive and the tag used to represent each stereotype.

3.1 Service Providers

A Service-Oriented Architecture is built upon independent entities which provide and manage services. Because SOSA is widely used as a way to integrate enterprise applications, the existence of these providers is justified by the necessity to project the business organizations involved in those processes into the architecture model. These providers act as *service containers* in charge of presenting the services contained to the world and also in charge of managing user interaction.

3.2 Active Components

In a SO solution, the main elements are *services*; however, as user responses are needed in many cases, a first-class element which retrieves the user interaction must be represented in the architecture model, namely *SystemFront-End*.

– *System front-end:*

By definition, services lack of the ability to interact or, in particular, to modify its behaviour according to user interaction. This reason justifies the necessity to include a component in the architecture model able to represent those capabilities. *SystemFront-End* components will be, therefore, identified as system boundaries.

– *Service:*

The main role of services inside SOSA is to support the functionalities offered by the system. Our vision of service (at PIM level) is aligned with that of the OASIS reference model for services *–A service is a mechanism to enable access to a set of one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description* [4]–.

Since both service providers and composite services are means of grouping services, composite services are distinguished for having a property indicating the coordination policy used. It can be set to: *Orchestration* or *Choreography.* In the former case, a service tagged with <<*OrchServ*>> will be needed.

3.3 Service Interactions and Messages

Services relate, communicate and interact with each other through *contracts*. The contracts in a service oriented architecture model act as connectors between the different architecture components. These contracts will have different features depending on the components connected. So, we refer to *ServiceContract* when talking about service-to-service connections and *InteractionContract* when talking about the communication between a *SystemFront-End* and a *service.* The communication item exchanged between services is represented by Messages identified as separated entities related to service contracts.

Table 1. Some relevant stereotypes used in the proposed UML profile for SOSA

NOTATION	ASSOCIATED CONCEPT	BASE UML META-CLASS	MEANING
<<outerProvider>>	Outer provider	Package	External provider of services
<<businessContract>>	Business contract	Dependency	Contract established between providers
<<FrontEnd>>	System Front-End	Classifier	Component that interact with user
<<CompServ>>	Composite service	Classifier	Service compound of services
<<BasicServ>>	Basic service	Classifier	Represents a basic funcionality
<<OrchServ>>	Orchestrator service	Classifier	Acts as orchestrator in a coreography
<<ServContract>>	Service contract	Association Class	Acts as connector of services
<<contractClause>>	Restriction clause	Classifier	pre-/post- conditions to be fulfilled
<<servOp>>	Service operation	Property	Service atomic functionality

4 Conclusions and Future Works

In this article we have presented a UML profile for the design of PIM-level SOA-based architecture models in the context of a SOD-M and methodology for the development of information systems.

The application of the MDA principles to the SOA paradigm favours the development of solutions based on services. In addition, the design of the architecture is a critical aspect in SO-based developments as one of its main goals is to achieve flexible implementations of interacting software entities.

Although there are some other works related with the topic of this article which also deal with the definition of the SOA principles ([5], [6]) or with UML profiles for service-based developments ([7], [8], [9]), many of them consider Web Service principles as basis (leading to a misconception of the SOA term). The ones which are enough generic to avoid fixing to a specific technology usually can not be applied to MDA or do not use a high-level standard notation to help the development process.

At this moment we are working on the specification of PSM-level SOA architectural models for different service execution platforms (Web Service, Grid, agents, etc) as well as the validation of the proposed architecture model.

Acknowledgments

This research is partially granted by project GOLD (TIN2005-00010) financed by the Ministry of Science and Technology of Spain.

References

[1] Miller, J., Mukerji, J. (eds.): OMG. OMG Model Driven Architecture. Document Nro. ormsc/2001-07-01 (2001), Available at: http://www.omg.com/mda

[2] Marcos, E., Acuña, C.J., Cuesta, C.E.: Integrating Software Architecture into a MDA Framework. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 127–143. Springer, Heidelberg (2006)

[3] Cáceres, P., Marcos, E., Vela, B.: A MDA-Based Approach for Web Information System Development. In: Workshop in Software Model Engineering (2003), http://www.metamodel.com/wisme-2003/

[4] OASIS: Reference Model for Service Oriented Architecture. Committee draft 1.0 (2006), Available at: http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf

[5] Lublinsky, B.: Defining SOA as an architectural style: Align your business model with technology. IBM DeveloperWorks site (January 09, 2007), http://www-128.ibm.com/developerworks/webservices/library/ar-soastyle/ index.html

[6] Papazoglou, M.P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: WISE 2003, pp. 3–12.

[7] Wada, H., Suzuki, J., Oba, K.: Modeling Non-Functional Aspects in Service Oriented Architecture. In: Proc. of the 2006 IEEE International Conference on Service Computing, Chicago, IL (September 2006)

[8] Zdun, U., Dustdar, S.: Model-Driven Integration of Process-Driven SOA Models. International Journal of Business Process Integration and Management, Inderscience, Pendiente de publicación (2007)

[9] Zhang, T., Ying, S., Cao, S., Jia, X.: A Modeling Framework for Service-Oriented Architecture. In: QSIC 2006. Proceedings of the Sixth International Conference on Quality Software, pp. 219–226 (2006)

Managing Separation of Concerns in Grid Applications Through Architectural Model Transformations

David Manset[1,2,3], Hervé Verjus[1], and Richard McClatchey[2]

[1] University of Savoie – Polytech' Savoie – LISTIC/LS
{david.manset,herve.verjus}@univ-savoie.fr
[2] CCCS, University West of England, Bristol, UK
richard.mcclatchey@uwe.ac.uk
[3] Maat Gknowledge, Toledo, Spain
dmanset@maat-g.com

Keywords: Grid, MDE, Software Architecture, Model Transformation.

1 Introduction

Grids enable the aggregation, virtualization and sharing of massive heterogeneous and geographically dispersed resources, using files, applications and storage devices, to solve computation and data intensive problems, across institutions and countries via temporary collaborations called virtual organizations (VO) as described in [1]. Most implementations result in complex superposition of software layers, often delivering low quality of service and quality of applications. As a consequence, Grid-based applications design and development is increasingly complex, and the use of most classical engineering practices is unsuccessful. Not only is the development of such applications a time-consuming, error prone and expensive task, but also the resulting applications are often hard-coded for specific Grid configurations, platforms and infrastructures. Having neither guidelines nor rules in the design of a Grid-based application is a paradox since there are many existing architectural approaches for distributed computing, which could ease and promote rigorous engineering methods based on the re-use of software components. It is our belief that ad-hoc and semi-formal engineering approaches, in current use, are insufficient to tackle tomorrow's Grid developments requirements. Because Grid-based applications address multi-disciplinary and complex domains (health, military, scientific computation), their engineering requires rigor and control. This paper therefore advocates a formal model-driven engineering process and corresponding design framework and tools for building the next generation of Grids. To achieve these objectives, two approaches are combined: (1) a formal semantic is used to model and check Grid applications; (2) a model-driven approach is adopted to promote model re-use, through separation of concerns, to model transformations, to hide the platform complexity and to refine abstract software descriptions into concrete usable ones.

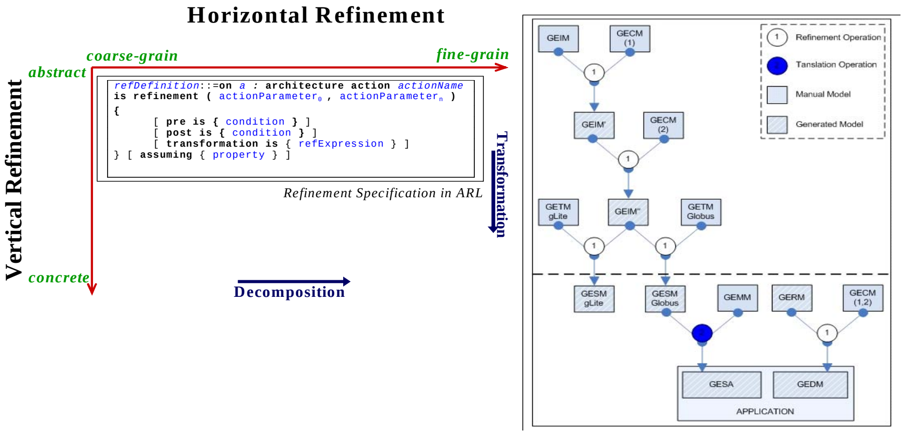
Section 2 of this paper introduces our proposal so-called *gMDE*, as well as its foundations in sections 3 and 4. Finally, section 4 and 5 illustrate the presented paradigms with an example.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 308 – 312, 2007.

2 A Formal Architecture-Centric MDE Approach

The presented approach, of so-called “grid Model-Driven Engineering” (*gMDE*), aims at enacting the model-driven paradigm based on formally defined architectural models dedicated to grid-based application development. While most existing MDE implementations provide only model to source code transformations where the *PIMs* are translated to *PSMs*, the problem of Grids engineering requires more elaborated models transformations – i.e. model to model - to fill the conceptual gap between the abstract model and its concrete (more detailed) representation. Moreover, interesting modelling aspects such as model optimization require the generation of intermediate models to compute and synchronize different views of the system. Thus, the proposed approach is based on the combination of the MDE vision [2] with the architecture-centric approach [3]..In Grid engineering, design is largely affected by constraints, which are introduced either by the targeted Quality of Services (QoS) or by the targeted execution platform. As presented in [4], our *gMDE* approach exhibits several models (see the right part of the Figure 1). Each model represents an accurate aspect of the system, useful for conceptual understanding (separation of concerns), analysis and refinement. Unlike the software engineering process, where the system architecture is iteratively refined by the architect, most of the transformations in *gMDE* are semi-automated. Thus, Grid applications architects only concentrate on applications functional building blocks and their interactions, and let the system address non-functional issues such as QoS. Figure 1 below, introduces the cascade of architectural models in the *gMDE* engineering process. In the presented process, a distinction is made between two major levels:

- Transformations of architectural models, which take place at the same level of abstraction (i.e. architectural structure, behaviour and properties) shown above the broken line in figure 1 and
- Transformations of abstract models to more concrete ones (including deployment in an infrastructure) shown below the broken line in Figure 1.



```
refDefinition::=on a : architecture action actionName
is refinement ( actionParameter0 , actionParametern )
{
      [ pre is { condition } ]
      [ post is { condition } ]
      [ transformation is { refExpression } ]
} [ assuming { property } ]
```

Fig. 1. The gMDE Design Process as a Cascade of Refined Models

3 gMDE as a Grid-Based Application Development Framework

ArchWare [3] is an architecture-centric engineering environment supporting the development of complex systems. It enables the support of critical correctness requirements and provides languages for expressing architecture structure, behaviour and properties. ArchWare provides a set of formal languages amongst which the Architecture Refinement Language (ARL [5]). This latter is used to describe software architectures (based on the Component and Connector paradigm) and to refine them according to transformation rules. This language is based on the π-calculus [6] and μ-calculus [7] allowing the specification of architectures structure, behaviour and properties. Our *gMDE* approach uses ARL as the basis language (a refinement calculus) for expressing architectural models and transformation rules.

The *gMDE* approach focuses on both directions of refinement i.e. "vertical" and "horizontal". The intention is not only to refine an architecture to a concrete and "close to final" code form, but also to adapt it according to constraints. *gMDE* proposes two ways of using the model-driven process. The first consists of optimizing a given Grid-based application abstract architecture according to expressed developers' QoS. The second consists of adapting an architecture according to the target Grid middleware Respectively:

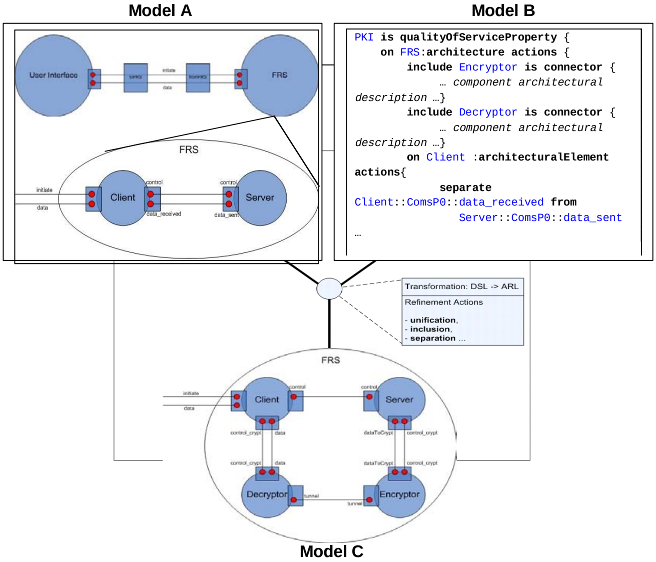
- QoS. Each QoS is represented by an architectural model (considered as an architectural pattern, which can be re-used in other Grid-based application architectures). This QoS representation is then incorporated into the current Grid-based application architecture through a set of refinement actions.
- Target Grid platform. Each Grid platform is represented by another architectural pattern. The Grid-based application abstract architecture is adapted to the platform representation through a set of refinement actions too.

To address the specificities of Grids, the ARL expressiveness has to be extended: the *gMDE* approach features a Domain Specific Language (DSL) allowing the description of proper Grid services and their associated constraints. This language is based on a Grid SOA paradigm promoting simplicity and facilitating the model comprehension, architectures being naturally expressed in terms of services and their properties.

4 *gMDE* Architectural Models Transformation Principles

Using *gMDE*, an architect formalizes on one hand the architecture of the grid-based application (model A) and on another hand, a QoS attribute (model B). The first model is expressed by using our DSL built on top of ARL.

To engage in the weaving process, the constraint definition model (model B) is transformed into refinement actions, as illustrated in the left part of Figure 2. During the weaving process, the Grid-based application architectural model (model A) is translated in ARL and the model B is interpreted and decomposed into a series of refinement actions in ARL too. The refinement actions are applied one by one on the model A until completion, resulting in a model C satisfying the specified QoS (right part of Figure 2).



```
PKI is qualityOfServiceProperty {
    on FRS:architecture actions {
        include Encryptor is connector {
            … component architectural
description …}
        include Decryptor is connector {
            … component architectural
description …}
        on Client :architecturalElement
actions{
            separate
Client::ComsP0::data_received from
                Server::ComsP0::data_sent
…
```

Fig. 2. *gMDE* architectural models transformation

5 Conclusion

The *gMDE* approach and environment are currently in use to evaluate potential advantages in the development process of various Grid applications. There are clearly identified challenges in the development of systems such as MammoGrid [8], which can be addressed by using *gMDE*. From these case studies, preliminary conclusions are encouraging and highlight the approach relevance.

References

[1] Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid – Enabling Scalable Virtual Organisations. International Journal of Supercomputer Applications (2001)

[2] Schmidt, D.C.: Guest Editor’s Introduction: Model-Driven Engineering. Computer 39(2), 25–31 (2006)

[3] Archware: The EU funded ArchWare IST 2001-32360 – Architecting Evolvable Software - project. http://www.arch-ware.org

[4] Manset, D., Verjus, H., McClatchey, R., Oquendo, F.: A Formal Architecture-Centric Model-Driven Approach For The Automatic Generation Of Grid Applications. In: ICEIS’06. Actes de 8th International Conference on Enterprise Information Systems, Paphos, Chyprus (2006)

[5] Oquendo, F.: π-ARL: an Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures. ACM SIGSOFT Software Engineering Notes archive 29(5) (September 2004)

[6] Milner, R.: Communicating and Mobile Systems: the pi-calculus. Cambridge University Press, Cambridge (1999)

[7] Kozen, D.: Results on the Propositional Mu-Calculus. Theoretical Computer Science 27, 333–354 (1983)

[8] Amendolia, S.R., et al.: Deployment of a Grid-based Medical Imaging Application. In: Proceedings of the 2005 HealthGrid Conference, UK (2005)

Aqueducts: A Layered Pipeline-Based Architecture for XML Processing[⋆]

Miguel A. Martínez-Prieto[1], Carlos E. Cuesta[2], and Pablo de la Fuente[1]

[1] GRINBD, Depto. de Informática, E.T.S. de Ingeniería Informática
Universidad de Valladolid, 47011 Valladolid (Spain)
{migumar2, pfuente}@infor.uva.es
[2] Kybele, Depto. Lenguajes y Sistemas Informáticos II, E.T.S. de Ingeniería Informática
Universidad Rey Juan Carlos, 28933 Móstoles, Madrid (Spain)
carlos.cuesta@urjc.es

Abstract. *Aqueducts* define a variant of the pipe-filter style designed to handle and manage semi-structured data streams, including those describing system structures themselves, such as XML-based architecture descriptions. This style is based on the concept of *aqueduct*, a higher-order filter which comprises a sequence of filters able to define a process logic using flow control constructs. Those filters can be expanded, then defining an inner layer in the *Aqueducts* hierarchy.

1 Introduction

Metalanguages define a mechanism to generate specific markup grammars which are used to describe the structure of different types of documents; nowadays, XML [2] is the reference metalanguage and then its documents can be considered as semi-structured data streams. Taking into account that many applications describe processes operating on these XML documents, our approach is designed to manage how these applications might interact since this need raise many issues related to interoperability [5].

Currently, the W3C is working on this issue through the *XProc* [8] definition. This initiative defines a language for describing operations to be performed on XML documents with an underlying model (based on *pipelines*) which implements basic processing requisites through different classes of component with a technological orientation.

In this paper, we present *Aqueducts* as an architectural vision of outlined above issues. For this purpose, we design a specific processing model partially inspired on the underlying model of XProc, in which communications are restricted to XML streams.

2 Aqueducts

An *architectural style* characterizes a family of systems by means of a small set of mappings from the syntactic to the semantic domain. The use of architectural styles has a number of significant benefits [4], hence we design *Aqueducts* model as a specific extension of the well-known *pipe-filter* [1] style for XML processing.

[⋆] This work has been partially funded by Projects TIN2006-15071-C03-02, TIN2006-15175-C05-01 (META) and TIN2005-25826-E from the Spanish Ministry of Education and Science.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 313–316, 2007.

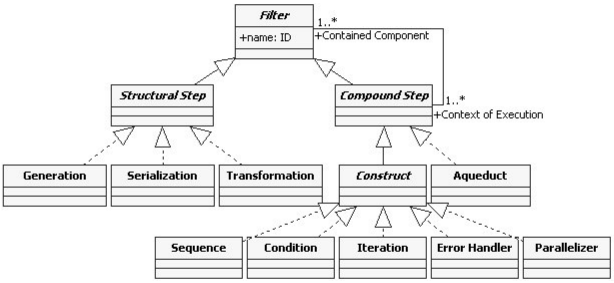


Fig. 1. Hierarchy of Components Structural Model

Aqueducts uses XML as data type for all component input and output, therefore our approach gets more flexibility than a classic pipe-filter thanks to its rich features for data representation. Considering this property, we design a specific processing model by means of a hierarchy of components (*Figure 1*) which allows managing XML streams.

This hierarchy is built from an abstract component (*filter*) that we understand as a composition unit of software applications, which has a set of interfaces and a set of requirements [7]. This concept abstracts syntactic and semantic features which are inherited by all components considered in the *Aqueduct*'s hierarchy.

2.1 Filter

We define a *filter* as *a processor element which performs a specialized program that transforms, in a specific way, the XML input data in a XML output result.*

Conceptually, this element is designed from the original definition of *filter* [6] and so, it is considered as an independent entity that performs its program in a context in which interacts with other filters. Therefore, a filter defines two concerns: the *program* and the *interface* in its context of execution in which is identified by a unique name.

The **program** concern defines the logic of process that the filter executes. We consider the program as finite state machine [1] which can be internally built (*structural steps*) or can be represented by an inner filters configuration (*compound steps*). This last case is based on hierarchical features of pipe-filter style that allows viewing a system, recursively, as a filter in another system [1]. From this property, we define the *context of execution* (for a fixed filter) as *the top level filter which contains the filter*.

The **interface** concern is responsible to allow the filter's communication within its context of execution. This process allows the filter to interact with other filters contained in its context of execution by means of a two disjoint set of ports: "input" and "output". *Input ports* are used to introduce contents in the filter. We consider two types: on one hand, a single *data reception* in which the filter receives the main content that it processes. On the other hand, multiple *input configuration* ports used for receiving complementary content required in the filter's computation. *Output ports* are used to deliver contents generated in the filter's execution. We consider two types: a single *result deliver* used for delivering filter's main result and multiple *output configuration* ports that the filter can use to deliver complementary content generated in its computation results.

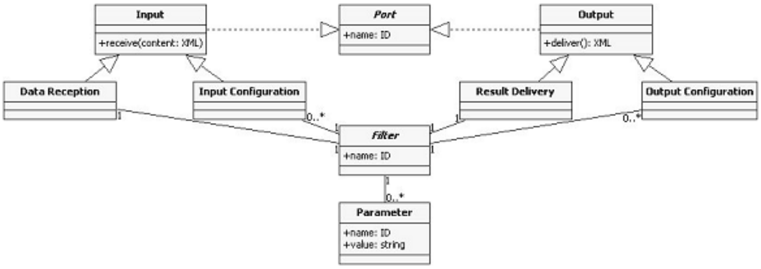


Fig. 2. Interface for the Communication Structural Model

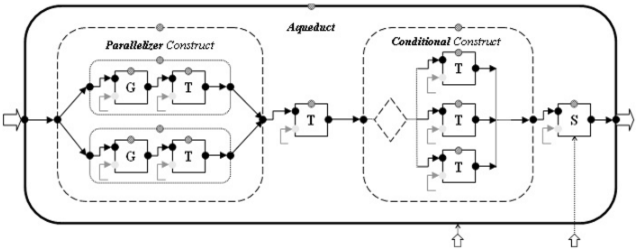


Fig. 3. A working *aqueduct* in a real environment

Moreover, all filters define an implicit *error* port which reports to its context possible errors produced in their execution.

Furthermore, the interface of communication allows to customizing the filter's program with literal values which are added by means of different *parameters*. *Figure 2* shows a global vision of the structural model that represents all interface concepts .

2.2 Hierarchy of Components

The basic difference between the components of this hierarchy is the way that use for implementing their program; furthermore, all components are characterized by a specific computational semantics. *Figure 1* shows a global vision of this hierarchy.

Structural Steps are defined as *atomic filters with an indivisible logic of process within its context of execution*. So, this type of components is responsible to perform basic operations required in *Aqueducts*, in order to reuse them in different contexts. Considering the pipe-filter pattern [3], we define three classes of structural steps: *generator* (as the aqueduct *data source*), *transformer* (as generic filters in the aqueduct) and *serializer* (as the aqueduct *data sink*).

Compound Steps are complex filters that define its program through an internal configuration of pipes and filters; this way, a compound step plays a *context of execution* role for all contained filters which share its interface of communication according to several restrictions. There are two classes: constructs and aqueducts.

We define a *construct* as a *compound step responsible to control and manage the data flow inside an aqueduct*. We consider the three basic control structures (*sequence*, *condition* and *iteration*), an *error handling* and a *parallelizer* construct designed in accordance with the concurrence features of the pipe-filter style.

An *aqueduct* is defined as the higher-level filter in the hierarchy. It is a service provider which design tasks as sequences of processing stages based on precedence among operations. This precedence establishes a generator and a serializer as, respectively, the first and the last steps executed in an aqueduct; between them, we consider transformers distributed according to a data flow designed by means of constructs. *Figure 3* shows an aqueduct with a specific configuration of structural steps and constructs.

3 Conclusions and Future Work

Aqueducts define an innovative vision of the pipe-filter style adapted for specific processing of XML. This model allows building several types of systems, since *Aqueducts* is able to process XML streams of any kind independently of their semantic.

These systems build their specific functionality through *aqueduct* structures that are understood as specialized *service providers* defined in accordance with a semantic model based on precedence among basic XML processing operations. Therefore, these services are highly extensible and adaptable, due to the model of specialized components that defines the *aqueduct* structure.

Currently, we are employing *Aqueducts* as technological basis in web applications. Our main development project is centred on a digital library which publishes electronic contents in heterogeneous environments through several services developed with *Aqueducts* technology. Plans of future are centred on the evolution of *Aqueducts* towards a SOA representation that allows us to build B2B systems with this processing model.

References

1. Allen, R., Garlan, D.: Towards Formalized Software Architectures. Technical Report CMU-CS-92-163, Carnegie Mellon University, School of Computer Science (July 1992)
2. Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0, 4th edn. W3C Recommendation (August 2006), available at http://www.w3.org/TR/REC-xml/
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented software architecture: a system of patterns. John Wiley & Sons, Inc., New York, USA (1996)
4. Garlan, D., Allen, R., Ockerbloom, J.: Exploiting Style in Architectural Design Environments. In: Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, pp. 175–188. ACM Press, New York (1994)
5. Milowski, A.: XML Processing Model Requirements and Use Cases. W3C Working Draft (April 2006), available at http://www.w3.org/TR/xproc-requirements/
6. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, New Jersey (1996)
7. Szyperski, C.: Component software: beyond object-oriented programming. ACM Press/Addison-Wesley Publishing Co., New York, USA (1998)
8. Walsh, N., Milowski, A.: XProc: An XML Pipeline Language. W3C Working Draft (April 2007), available at http://www.w3.org/TR/xproc/

On the Interplay of Crosscutting and MAS-Specific Styles

Ambra Molesini[1], Alessandro Garcia[2], Christina Chavez[3], and Thais Batista[4]

[1] Alma Mater Studiorum—Università di Bologna, Italy
ambra.molesini@unibo.it
[2] Lancaster University, UK
a.garcia@lancaster.ac.uk
[3] UFBA—Universidade Federal de Bahia, Brazil
flach@dcc.ufba.br
[4] UFRN—Universidade Federal do Rio Grande do Norte, Brazil
thais@ufrnet.br

Abstract. This paper presents a systematic case study that analyzes the influence exerted by different styles over the nature of architectural crosscutting concerns in an evolving multi-agent system. The analysis encompassed the systematic comparison of alternative architecture decompositions for the same application that changed over time to address different stakeholders' concerns.

1 Introduction

Architectural aspects are expected to "modularize widely scoped concerns that naturally crosscut the boundaries of system components at the software architecture level" [1]. However, the current knowledge on architecturally-relevant crosscuttings tends to neglect other well established architecture design abstractions such as architectural styles. In fact, different architectural styles applied to the same problem can lead to designs with significantly different modularity properties [2]. Hence, with the wide recognition of the software lifecycle impairments caused by crosscutting concerns, architects need to improve their knowledge on the relationship of such concerns and pointcuts stylistic design choices to date. Unfortunately, there is little knowledge and reflection about the influence exerted by specific architectural styles on the nature of crosscutting concerns in early design stages.

In this context, the motivation for studying the role played by domain-specific styles in the manifestation of crosscuttings seems appealing. Our research provides a first, significant stepping stone on addressing this gap. We describe a case study where we have observed the influence of distinct stylistic choices on the nature of crosscutting concerns in a multi-agent system (MAS) architecture, such as error handling and coordination. We have used an evolving conference management system (CMS) as case study (Section 2.1). Different architectural designs for this system have been produced (Section 2.2), and a number of crosscutting concerns emerged from these different decomposition (Section 2.3). We

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 317–320, 2007.

have made an analysis upon the variability of the crosscutting concerns as different styles have been applied. Section 3 provides an overview of our findings.

2 Architectural Crosscutting and MAS-Specific Styles

In a case study, we have observed phenomena related to crosscutting concerns when different architectural designs have been defined over time for the same application [3,4], name Conference Management System (CMS) (Section 2.1). This section discusses two alternative designs subsequently generated for the CMS system, with each of them based on a distinct architectural style (Section 2.2) and argues about the nature of the heterogeneity of the crosscutting concerns that manifested in such designs (Section 2.3).

2.1 The Conference Management System

The CMS application supports the management of paper submissions to scientific conferences [5]. Setting up and running a conference program is a multi-phase process, involving several individuals and groups. The agents enrolled in this system represent a number of people involved, such as chairs and reviewers. In the review phase, the program committee has to handle the review of the papers, i.e. contacting potential referees and asking them to review a number of the papers. In the final phase authors are notified of these decisions. There is a number of architectural concerns in the CMS design, like exception handling and coordination. Coordination protocols are essential in such an architecture in order to regulate the interaction-intensive activities involving the system's agents. Error handling is also a concern in the CMS in order to enhance system robustness.

2.2 MAS-Specific Styles

The following text summarizes the instances of the two employed styles to address coordination issues, namely: *(i)* Reflective Blackboard, *(ii)* and Stigmergic Coordination. Figure 1 provides a partial description of both styles' instances in CMS; it illustrates the key elements in each style used.

Reflective Blackboard. The Reflective Blackboard style [4] is built from the composition of two well-known architectural styles [6]: the *Blackboard* and the *Reflection* patterns. The basic idea of this pattern is that since communication is centralized on the blackboard, systematic control properties are modularly introduced to it in a finer-grained fashion. Control is transparently inserted in the desired points of inter-agent communications by using reflective features (1a). The components of the Reflection pattern are used to refine the overall structure of the Blackboard: elements of the blackboard are associated via meta-object protocol (MOP) with meta-objects defined in the control components. The instantiation of this style to the CMS architecture is sketched in 1a.

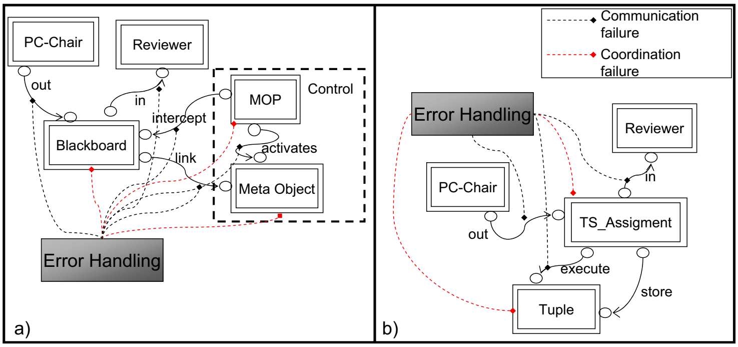


Fig. 1. *a)* Reflective Blackboard; *b)* Stigmergic Coordination

Stigmergic Coordination. The Stigmergic Coordination style is another evolution of the Blackboard style and it is particularly suited for self-organizing systems. Stigmergy [3] refers to all those kinds of indirect interactions occurring among situated agents that reciprocally affect each others' behavior. So far, the most widely used stigmergic mechanism in MAS include pheromone-based behaviors, relying on agents depositing markers that the environment can diffuse and evaporate. This pattern presents a pheromone-based approach: tuples can be seen as pheromones. The tuple space acts only as a shared data-space where the tuples are stored and executed. The instance of this style is shown in 1b where the assignment process of CMS is sketched.

2.3 Crosscuttings in MAS-Specific CMS Architectures

This section reviews MAS-specific concerns in both CMS architectures.

Error handling. Exception handling was a crosscutting concern in all the MAS-specific styles (Figure 1). Figure 1 presents the widely-scoped effect of handlers associated with two different kind of exceptions: communication error and coordination error. Communication failures are caused by network errors. Coordination failures are caused by error manifestation either in the tuple space for the Stigmergic Coordination patterns or in the control for Reflective Blackboard pattern. In the Stigmergic Coordination architecture, the agents need to collaborate for addressing error handling. Hence, behaviors for managing errors end up being spread all over several agents and tuples. In the Reflective Blackboard pattern, both agents and control are also involved in error treatments.

Coordination is a crosscutting concern in the Stigmergic Coordination pattern only. In fact, this pattern adopts tuples as places to put in the coordination code, but the complex coordination protocol is also realized by agents that generate the tuples. As a result tasks associated with the coordination protocol is scattered over agents involved in the protocol. In the Reflective Blackboard pattern, coordination is not a crosscutting concern because coordination is modularized in specific meta-objects defined inside the control component.

3 Conclusions

The main outcomes of our analysis indicated that some system concerns are well modularized while others are not, depending on the choice of domain-specific architectural style(s). We have also observed that styles have directly interfered with the nature of the crosscutting concerns at the architectural description of the CMS system (Section 2.3). Specific styles also typically determine the level of tangling or scattering of a particular concern. In addition, there are some concerns that are recurrently crosscutting independently of the dominant architecture stylistic choice or specific style compositions. For instance error handling is a typical example of architectural concern that tends to be crosscutting in all the architectural styles. We have also observed that default error handling policies, such as exception propagation or error-specific information logging [7], are consistently present in almost all types of architectural decomposition. In fact, error handling is widely recognized as a global design issue and has been extensively referred in the literature as a classical crosscutting concern in systems following different kinds of architecture decompositions. Furthermore, it may affect almost all the system modules and their interfaces representing an anti-modularity factor in several architectural styles [6] as well. Our investigation confirms some findings we have detected in the analysis of a web-based information system [8,9].

References

1. Garcia, A., et al.: On the modular representation of architectural aspects. In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg (2006)
2. Shaw, M.: Comparing architectural design styles. IEEE Software 12, 27–41 (1995)
3. Mamei, M., et al.: Programming stigmergic coordination with the tota middleware. In: Dignum, F., et al. (eds.) Proc. of AAMAS'05, pp. 415–422. ACM Press, New York (2005)
4. Silva, O., et al.: The reflective blackboard pattern: Architecting large multi-agent systems. In: Garcia, A.F., de Lucena, C.J.P., Zambonelli, F., Omicini, A., Castro, J. (eds.) Software Engineering for Large-Scale Multi-Agent Systems. LNCS, vol. 2603, pp. 73–93. Springer, Heidelberg (2003)
5. Ciancarini, P., et al.: A case study in coordination: Conference management on internet (1996), `ftp://ftp.cs.unibo.it/pub/cianca/coordina.ps.gz`
6. Buschmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. Wiley, Chichester (1996)
7. Filho, F.C., et al.: Exceptions and aspects: the devil is in the details. In: SIGSOFT FSE 2005, pp. 152–162. ACM Press, New York (2006)
8. Greenwood, P., et al.: On the impact of aspectual decompositions on design stability: An empirical study. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 176–200. Springer, Heidelberg (2007)
9. Chavez, C., et al.: Are architectural aspects style dependent? In: 1st Workshop on Aspects in Architectural Description (2007)

Processes for Creating and Exploiting Architectural Design Decisions with Tool Support

Francisco Nava[1], Rafael Capilla[1], and Juan C. Dueñas[2]

[1] Department of Computer Science, Universidad Rey Juan Carlos,
c/ Tulipán s/n, 28933, Madrid, Spain
{francisco.nava,rafael.capilla}@urjc.es
[2] Department of Engineering of Telematic Systems, ETSI Telecomunicación
Ciudad Universitaria s/n, 28040, Madrid, Spain
jcduenas@dit.upm.es

Abstract. Software architectures suffer of a serious lack of documented design decisions, but also an explicit definition of the processes needed to create and exploit such architectural knowledge. To address these issues, we focus on the specification of those activities that we believe should be implemented to support the creation and use of design rationale with tool support.

1 Design Decisions in Software Architecture

Software architectures have been widely used to describe the main parts of a software system. Recently, this traditional perspective is changing to include design rationale, which understands the architecture as the result of a set of architectural design decisions [2]. The "decision view" was proposed in [6] as a "new" architectural view which crosscuts the information of other architecture views as it documents explicitly the design decisions made. Also, as the system evolves, these decisions have to be maintained accordingly to the changes made. The importance of recording design decisions has been recognized as a way to bridge the gap between requirements and architectural products and for traceability issues as well [14]. Perry and Wolf [11] mentioned the rationale and principles that guide the design and evolution of software architectures to justify the final shape of the architecture. This idea is reflected in [4], which discuss the importance of documenting such rationale, but the processes to create and use such knowledge are not described. In 2005, Tyree and Akerman [13] proposed a list of attributes for charactering design decisions, while in [12] the authors discuss that recording architectural descriptions, mainly based on diagrams describing the relationships between components and connectors, is not enough. The need to reduce the maintenance effort and avoid architecture erosion because decisions were never recorded motivates the need to replay them. Hence, the processes to achieve this goal should be clearly described.

2 Processes for Architecting with Design Rationale

Previous efforts tried to characterize design decisions in architecture, but also important are the processes that use such knowledge under typical software engineering

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 321 – 324, 2007.

activities. A recent report [10] distinguishes between *architecture as a product* from *architecture as a process*. The former considers design decisions "as a product" while the latter deals with the processes used in the decision-making activity. The classification proposed in [10] distinguishes four main types of activities: *architecting and sharing knowledge* in the producer side and *assessment and learning knowledge* in the consumer side. Our goal in this work is to detail the sub-activities that happen under these four main categories as a refinement outcome. Those activities labelled with an asterisk (*) have been defined in [10].

Architecting*: It concerns with those process that create and store architectural knowledge. The sub-tasks we define for this activity are discussed below.

- **Make decisions:** Software architects make decisions motivated by a set of goals. Decisions are stored and characterized by a set of attributes that describe the rationale that motivated each decision. As knowledge producers, decisions are stored and documented for future (re)use, but before a decision is selected as valid, we have to evaluate the impact of other alternatives and store these as well. Because decisions can be made at different times and before the final choices are made, it seems useful to label the status of the decisions. Often, decisions are made based on previous knowledge such as design patterns, architectural styles, or previous decisions. Hence, knowledge searching activities can be carried out.
    - **Knowledge search / discovery / reuse:** Knowledge search processes and discovery methods are particularly welcome [5] to extract stored knowledge. Different forms of search like, navigation, browsing, queries, keyword search, etc, can be employed to reuse codified knowledge.
- **Evaluate decisions:** The reasoning activity is usually based on the consideration of analysis of the implications for a set of similar decisions. Therefore, an evaluation process based on simulation, scenario analysis, or impact analysis should be carried out to decide the best alternative among several ones. The expertise of the architect plays a key role in the evaluation activity.
- **Validate decisions:** Before decisions are stored we should check the integrity of such knowledge (e.g.: removing a dependency between two design decisions may cause a loss of information or the existence of incompatible decisions).

Sharing*: Knowledge sharing makes architectural knowledge (AK) available to others [10]. Several ways can be used for transferring AK [7], but there is a need to determine which AK is worthy to be shared. We define the following sub-activities.

- **Passive sharing:** The architect retrieves and reviews existing and structured AK (e.g.: documentation, books, web pages, codified knowledge) stored by others.
- **Active sharing:** Meetings with other stakeholders promote an active way of knowledge sharing, which can be improved using publisher-subscriber strategies (e.g.: RSS, contents for distributed teams) under a collaborative environment.

Assessing*: Assessment is used to make recommendations about viable an unviable decisions and two types of assessment can be defined.

- **(Pre) Assessment for architecting:** The results of evaluation and simulation tasks are used to assess the architect for future decisions. Impact analysis provides recommendations about the viability of a decision before it is made.

- **(Post) Assessment for learning:** This task performs assessment during post-architecting activities (e.g.: maintaining) as it learns from past decisions and documented experiences.

Learning*: Past experiences are valuable knowledge assets for architects as consumers of such knowledge. E-learning and personal training can be used.

3 Tool Support with ADDSS

Prior work describes ADDSS 1.0, a web-based tool for recording, managing and documenting architectural design decisions [3] and this section outlines the processes discussed in section 2 and supported by the new version ADDDS 2.0.

Architecting: ADDSS 2.0 supports the decision making process and stores design decisions which can be associated to requirements and design objects. The rationale of the decisions is described as a free text description. ADDSS 2.0 stores the date of the decision, the responsible, and a status indicating if the decision is pending, rejected, approved, etc. ADDSS users can navigate and browse through the decisions following the iterations of the architecting process. A query mechanism retrieves information about the trace links between decisions to requirements and to architectures. The user can browse stored design patterns and apply them in new decisions, but no mechanism is defined to reuse past decisions. The evaluation of design decisions is partially supported by adding alternative decisions that can be evaluated externally, but no specific fields have been defined to store the pros and the cons of the decisions being considered. ADDSS 1.0 doesn't provide by the moment validation mechanisms.

Sharing: Sharing mechanisms are limited to passive sharing were the user can examine the documentation generated automatically by the tool.

Assessing: Assessment is a human task difficult to automate. ADDSS facilitates assessment for architecting because alternative decisions can be stored and labelled adequately. Assessment for learning in ADDSS 2.0 has at this stage no direct support as a guidance based on experience.

Learning: ADDSS only provides PDF documents from the decisions made but no further recommendations as a learning guide.

4 Conclusions

This paper extends the activities described in [10] and details those implemented in ADDSS 2.0. Similar efforts with tool supports can be found in [1] [8]. From a process perspective, we have analyzed the sub-activities that would help architects to create and exploit the rationale that guides the reasoning activity [9], but it is difficult to illustrate in 4 pages how these activities should carried out. A preliminary evaluation was done in 2006-2007 with ADDSS 1.0 (22 students of a master course organized in 11 teams participated in creating the architecture of a real system). Most of the teams spent their time in architecting and evaluation, but they had some difficulties to make explicit the decisions made. Most of the teams perceived useful to record design decisions because the architecture was developed in several iterations in different days, so they could remember the decisions made. Because not much interaction happened between the teams, sharing was limited, and we didn't spend time in learning as it was an isolated project. For next months we expect to have additional results with the new capabilities implemented in ADDSS 2.0.

References

1. Babar, M.A., Gorton, I.: A Tool for Managing Software Architecture Knowledge. In: Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops (2007)
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. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: A Web-based Tool for Managing Architectural Design Decisions. In: Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge. ACM Digital Library, Software Engineering Notes, vol. 31(5)
4. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures. Views and Beyond. Addison-Wesley, Reading (2003)
5. de Boer, R.C.: Architectural Knowledge Discovery, Why and How? In: Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge, ACM Digital Library, Software Engineering Notes, vol. 31(5)
6. Dueñas, J.C., Capilla, R.: The Decision View of Software Architecture. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 222–230. Springer, Heidelberg (2005)
7. Farenhorst, R.: Tailoring Knowledge Sharing to the Architecting Process. In: Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge, ACM Digital Library, Software Engineering Notes, vol. 31(5)
8. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: 5th IEEE/IFIP Working Conference on Software Architecture, pp. 109–118 (2005)
9. Kruchten, P., Lago, P., van Vliet, H., Wolf, T.: Building up and Exploiting Architectural Knowledge. In: 5th IEEE/IFIP Working Conference on Software Architecture (2005)
10. Lago, P., Avgeriou, P.: First Workshop on Sharing and Reusing Architectural Knowledge. ACM SIGSOFT Software Engineering Notes 3(5), 32–36
11. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. Software Engineering Notes, ACM SIGSOFT, 40–52 (October 1992)
12. Tang, A., Babar, M.A., Gorton, I., Han, J.A.: A Survey of the Use and Documentation of Architecture Design Rationale. In: 5th IEEE/IFIP Working Conference on Software Architecture (2005)
13. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22(2), 19–27 (2005)
14. Wang, A., Sherdil, K., Madhavji, N.H.: ACCA: An Architecture-centric Concern Analysis Method. In: 5th IEEE/IFIP Working Conference on Software Architecture (2005)

Supporting the Automatic Generation of Proto-Architectures

Elena Navarro[1], Patricio Letelier[2], Javier Jaén[2], and Isidro Ramos[2]

[1] Computing Systems Department, UCLM, Albacete, Spain
enavarro@dsi.uclm.es
[2] Department of Information Systems and Computation, UPV, Valencia, Spain
{letelier,fjaen,iramos}@dsic.upv.es

Many issues must be taken into account in order to provide a right specification of the system-to-be to meet properly the established requirements. In this sense, the introduction of proper supporting techniques able to automate as much as possible the process means a clear advantage. In this work, we introduce a tool called MORPHEUS that gives support to our proposal by providing traceability throughout the process of generation of proto-architecture from requirements.

1 Introduction

The specification of the Software Architecture of the system-to-be meeting the established requirements is always a challenging activity. Many decisions at the architectural level must be made to meet both the functional and non-functional requirements. However, it must be taken into account the impact of these decisions and the relations they have with other decisions and requirements before realizing them in the system-to-be. ATRIUM [6] is a methodology that proposes a support in this context (see Fig.1). It is an Aspect-Oriented Software Development (AOSD) proposal that provides support to the specification, analysis and traceability from early-aspects in the Requirements stage to their realization in the architecture by using an Aspect-Oriented Architecture Description Language (AO-ADL).

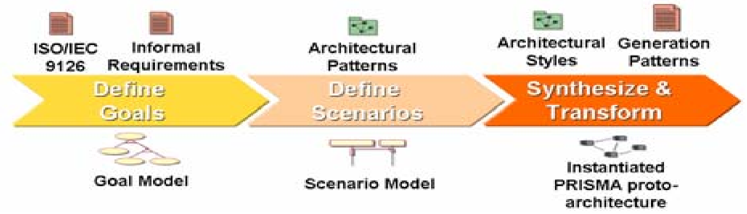


Fig. 1. A sketched view of ATRIUM

During the description of ATRIUM, several tasks, which could be cumbersome and error-prone for the analysts, have been automated in order to speed up the process and improve the results. One of these tasks is the automatic generation of the proto-architecture from the set of aspectual scenarios. In order to provide a proper solution to

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 325–329, 2007.

carry out these tasks, we have developed a tool called MORPHEUS whose main features are described in the following.

2 MORPHEUS: A Supporting Tool Support to ATRIUM

ATRIUM is defined by means of three different models: requirements model, scenario model and architectural model. For this reason, MORPHEUS has been developed providing the user with three different environments:

–Requirements Environment [4] provides analysts with a requirements metamodelling tool for describing Requirement Metamodels customized according to the project's semantic needs. This environment automatically provides a tool for modelling according to the active Metamodel along functionality for its analysis. This analysis can be customized to the specific needs as well. This environment is used to describe and exploit the ATRIUM Goal Model [5] that is in charge of describing the system-to-be requirements and analysing architectural alternatives.

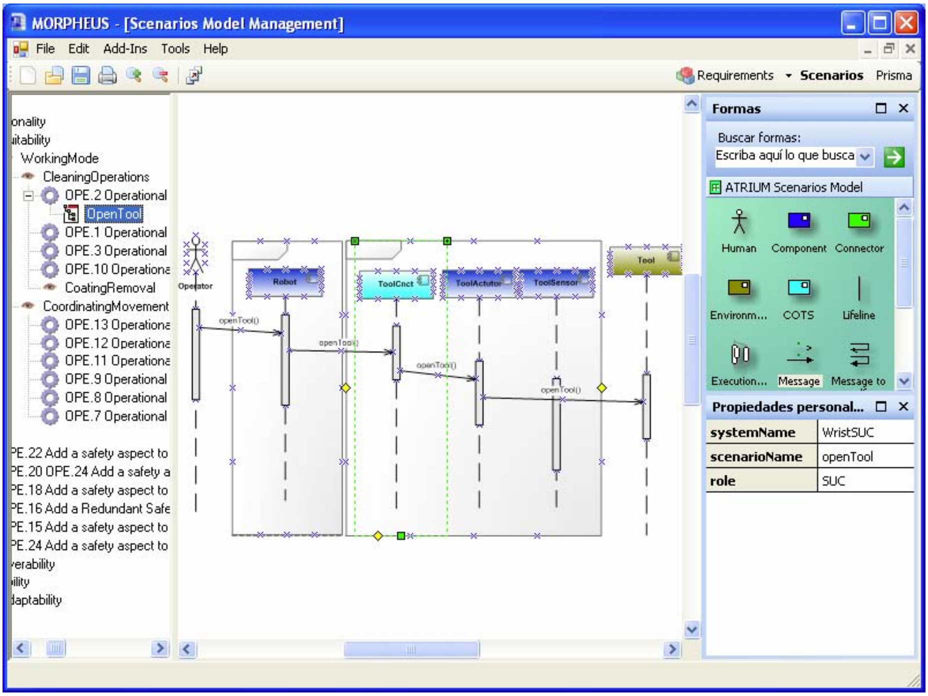
–*Scenarios Environment* has been expressly developed to describe architectural scenarios by means of a UML 2.0 profile defined as an extension of the Sequence Diagrams. It facilitates the description of architectural and environmental elements along with specific expressiveness to support the aspectual approach. It also provides facilities to synthesize the proto-architecture from the set of scenarios. More details about this environment are provided in section 2.1.

–*Software Architecture Environment* [7] makes available a whole graphical environment for the PRISMA AO-ADL. This means that if PRISMA was the selected target architectural model to be generated by the *Scenarios Environment*, the proto-architecture obtained from the Scenarios Model could be refined. Once the software architecture has been completed, the analyst can proceed to generate code for its execution in the PRISMANET Middleware.

It is worthy of note that traceability is implicitly maintained among the different environments thanks to the automatic links established connecting elements belonging to the different models.

2.1 Scenarios Environment

In ATRIUM, the Scenario Model is exploited to carry out the generation of the PRISMA proto-architecture. This Model provides us with partial views of the architecture, where only shallow-components, shallow-connectors and shallow-systems have been identified along with their behavior expressed through their interaction. They are called shallow because we do not need their complete definition but an initial one that can be refined in later stages of development if it is necessary. In order to describe these Scenarios, a UML 2.0 profile has been defined that extends the Sequence Diagrams to provide the necessary expressiveness for modeling concepts such as Systems, Aspectual Messages, etc (more details in [3]). Fig. 2 shows an example of an ATRIUM Scenario where several architectural and environmental elements are collaborating by means of a sequence of messages. It can be observed that different colors and shapes are used to facilitate its comprehension.



systemName	WristSUC
scenarioName	openTool
role	SUC

Fig. 2. What MORPHEUS looks like when the Scenarios Editor is loaded

Fig. 2 shows that the *Scenarios Editor* provides the user with different functionalities to facilitate the modeling. The *Model Explorer* facilitates the navigation through the Scenario Model being defined in an easy an intuitive way and manage (creation, modification and deletion) the defined scenarios. It is pre-loaded with part of the information of the Goal Model being defined. For this reason, the selected *operationalizations*, catalogued by their dimensions, are displayed. It facilitates to maintain the traceability between the Goal Model and the Scenarios Model. Associated to each operationalizations one or several scenarios can be specified to describe how the shallow architectural elements collaborate to realize that operationalization (see scenario "OpenTool" in Fig.2). In the middle of the environment is situated the *Graphical View* where the elements of the scenarios can be graphically specified.

On the right side it can be seen the *Stencil* that makes available the different shapes to describe graphically the ATRIUM scenarios. The user only has to drag and drop on the *Graphical View* the necessary shapes. In addition, below the stencil a control allows the user to introduce the properties necessary for each element being defined.

The second context is the *Synthesis processor*. It is in charged of the generation of proto-architecture. For its development, the alternative selected was the integration of one of the existing model transformations engines considering that it has to provide support to the transformations described in [2]. They have been defined to synthesize the proto-architecture from the set of architectural scenarios. Specifically, ModelMorf [1] was selected because it supports the QVT-Relations language. It should be mention that this engine supports multi-directional transformation specification and incremental transformations. It is worthy of note that it also provides support to traceability by means of the generation of trace class associated to each relation. This means that it can be exploited to maintain the traceability both bottom-up and top-down. This engine accepts as inputs the Metamodels and their corresponding models in XMI format to perform the transformation. For this reason, the *Synthesis processor* proceeds in several steps. First, it stores the Scenario Model being defined in XMI. Second, it provides the user with a graphical control to select the destination target architectural model, the QVT rules for its transformation and the name of the protoarchitecture to be generated. By default, PRISMA is the selected target architectural model because the QVT rules for its generation have been defined. However, the user can defined its own rules and architectural Metamodel to synthesize the Scenario Model according to his/her specific needs. Finally, the *Synthesis processor* performs the transformation by invoking ModelMorf. The result is an XMI file describing the proto-architecture. The Software Architecture environment can load this XMI file for its refinement if PRISMA was the selected target architectural model.

3 Conclusions

In this work a brief introduction to MORPHEUS, the supporting tool of ATRIUM, has been presented. Special emphasis has been put on the Scenarios Environment. It has been described how it facilitates the description of the ATRIUM Scenario Model and how it is synthesized to generate the target proto-architecture. It must be highlighted that as far as we know this is the only tool that performs this synthesize using QVT and considering the aspectual approach from early stages of development.

Acknowledgements. This work is funded by the Dept. of Science and Technology (Spain) under the National Program I+D+I, META project TIN2006-15175-C05-01.

References

[1] ModelMorf (2007), http://www.tcs-trddc.com/ModelMorf/index.htm

[2] Navarro, E.: Architecture Traced from RequIrements by applying a Unified Methodology. PhD thesis, Computing Systems Department, UCLM (May 30, 2007)

[3] Navarro, E., Letelier, P., Ramos, I.: Requirements and Scenarios: playing Aspect Oriented Software Architectures. In: WICSA 2007. Sixth Working IEEE/IFIP Conference on Software Architecture, Mumbai, India (short paper, January 6-9 2007)

[4] Navarro, E., Letelier, P., Reolid, D., Ramos, I.: Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques. In: 15th Int. Conference on Information System Development, Budapest, Hungary (August 31-September 2, 2006)

[5] Navarro, E., Letelier, P., Mocholí, J.A., Ramos, I.: A Metamodeling Approach for Requirements Specification. Journal of Computer Information Systems 47(5), 67–77 (2006)

[6] Navarro, E., Ramos, I., Pérez, J.: Software Requirements for Architectured Systems. In: RE'03. Proceedings of 11th IEEE International Requirements Engineering Conference (short paper), Monterey, California, USA, September 8-12, 2003, pp. 365–366. IEEE Computer Society Press, Los Alamitos (2003)

[7] Pérez, J., Navarro, E., Letelier, P., Ramos, I.: A Modelling Proposal for Aspect-Oriented Software Architectures. In: ECBS. 13th IEEE Int. Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March 27th-30th, 2006, IEEE Computer Society Press, Los Alamitos (2006)

[8] QVT: MOF Query/Views/Transformations final adopted specification. OMG document ptc/05-11-01, 2005 (November 5, 2005)

AspectLEDA: Extending an ADL with Aspectual Concepts*

Amparo Navasa, Miguel A. Pérez, and Juan M. Murillo

Quercus Software Engineering Group
Department of Computer Science, University of Extremadura. Spain
{amparonm,toledano,juanmamu}@unex.es

Abstract. When increasing the complexity of software systems new techniques allowing for their adequate manipulation are required. In the last ten years, AOSD has been proposed to manage the systems complexity by considering concepts of AO programming extended along the software life cycle. The suitability of the existence of an AO architectural design appears when AO concepts are extended to the whole life cycle. In order to adequately specify the AO design, Aspect-Oriented ADLs are needed. In this paper *AspectLEDA*, an ADL to support architectural descriptions treating aspects, is presented.

Keywords: Software Architecture, AOSD, ADL, AO-ADL.

1 Introduction and Motivations

Software Architecture combined with Aspect-Orientation (AO) is concerned with representing concerns that cut across regular architectural components. Suitable ADLs to describe architectures including aspects (AO-ADLs) are needed to endow with linguistic support for architecture descriptions that integrate aspects. They should also provide a formal basis to analyze the architecture properties and to detect errors.

Several proposals for the integration of Software Architecture and Aspect Oriented Software Development (AOSD) have been developed in order to join the advantages of both approaches. In this scenario, not many AO-ADLs have been proposed. They face the introduction of aspects into ADLs in different ways: some of them are extensions of regular ADL, but others are new languages; most of them introduce the aspects as components but others as connectors or as architectural views. Besides, each AO-ADL follows the symmetric or asymmetric (most of them) approach to deal with aspects. Some of them provide a strong formal basis to check system properties. In addition, several of the ADL allow for code generation and thus obtaining an architectural prototype. Finally, most of them provide the designer with a tool to define the architecture. References can be found in [2].

In this paper *AspectLEDA*, an AO-ADL, is proposed which is aligned with those criteria in the following way. It is based on a regular ADL, LEDA[1], which has been extended. *AspectLEDA* follows the asymmetric model which provides the advantage

* This research work is partially supported by project number TIN 2005-09405-C02-02 and 2PR04B011.*

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 330 – 334, 2007.

of supporting system evolution. Besides, it incorporates dynamic reconfiguration by means the reflective architecture model. Aspects in *AspectLEDA* are turned into components which remain separated (not weaved) during the whole process. As additional property it is supported by a tool that assist the architect during the design providing him with the possibility of generating an executable Java code to obtain the simulation of the architecture.

Thus, the reasons why *AspectLEDA* is proposed are twofold: first to provide support for an architectural description in which aspects are considered; and secondly, to provide a base to check whether the architecture will support the behaviour expected in the system.

In section 2 how to describe AO architectures with *AspectLEDA* is shown.

2 AspectLEDA

AspectLEDA is an extension of LEDA with primitives for describing AO concepts. LEDA is an ADL for the description and verification of structural and behavioural properties of software systems which has its semantics specified in Pi-calculus. In addition, LEDA language is structured in two levels: components and roles.

The first step when describing an *AspectLEDA* architecture is generating an intuitive description. Fig. 1 (partially) represents the system components (*C1* and *C2*), and the aspects to be added (as components). In addition, a new kind of connector extending UML is defined allowing us to represent interactions between components and aspects. It will intercept the interact tions previously defined between the system components. Such connector represent complex artefacts including the point, in which the aspect must act, the conditions under which the aspect must be activated and the moment in which it must be done.

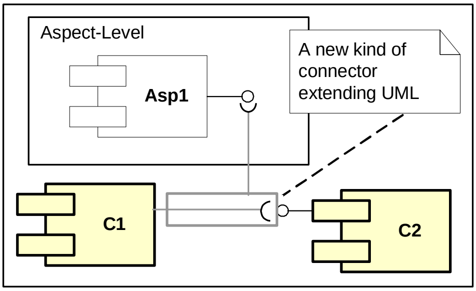


Fig. 1. Intuitive Graphical Description

Detailed Architecture Description. After the first intuitive approach of the system the new connector must be described. To do it, the scenarios of the use case affected by the aspect are studied and the sequence diagrams representing them are extended: a new architectural component, which contains the aspectual component and manages its interactions, has been included. It will intercept the methods call defining the Join Points. The objective is to produce the *Common Items* (*CI*) structure for the aspect which contains the required details of the interactions between aspects and components; it will be used later. For each scenario in which the aspect is involved a *CI* is produced. Table 1 shows its contents.

***AspectLEDA* Language Description.** At this point, the required information to proceed with the detailed description of the system is available. The first step is describing the base system. Since the obliviousness principle must be observed [3], the base system includes neither reference to aspects nor special primitives supporting aspects. Thus, the base system is described in LEDA. Fig. 2 shows the architecture, which is described as a composed component including generic components *Client* and *Server,* which are defined by their interfaces. The roles description specifies the interaction protocols. The connection between components is specified in the *attachments* section.

Table 1. Common Items structure

Element	Description
Aspect Name	It specifies the name for the aspect to which this CI structure is associated
Insertion Point - IP	Each cutting point identified in the Sequence Diagram. JP in AOP
Extension Point	Identified point in the use case
Aspect Operation	Name of the aspect component to be applied
Component	Name of the component affected by the IP
Event	Event type triggering the aspect application
Application Condition	Conditions that must be satisfied allowing the aspect to be executed
When Clause	When the aspect can/must be applied

```
Component Basesystem {
interface none;
composition
    ccli: Client;
    cserv: Server;
attachments
ccli.requireM(serverop,param)<>cserv.provideM
(serverop,param);
}
component Client {
interface
   requireM:RequireM;
}
component Server {
interface
   provideM:ProvideM;
}
role ProvideM(serverop,param){
 spec is
   serverop?(answer).(value)answer!(value).
   ProvideM(serverop,param);
   }
role RequireM(serverop,param){
 spec is
   (answer)serverop!(answer).answer?(value).
   RequireM(serverop,param);
   }

instance basesys:Basesystem;
```

Fig. 2. Basic System Architecture in LEDA

```
component Aspect {
interface
   asprole:AspectRole;
}
role AspectRole(aspectop,param){
 spec is
   aspectop?(ans).(val)answer!(val).
   AspectRole(aspectop,param);
   }
```

Fig. 3. Aspect Definition in LEDA

```
Component Extendedsystem {
composition
    basesystem: System;
    asp: Aspect;
attachements
  basesystem.cserv.serverop()<<RMA,
  condvalue,when_cond>>asp.aspectop();
}
component System {
interface
    none;
}
component Aspect {
interface
    asprole: AspectRole;
}
instance extsys: Extendedsystem;
```

Fig. 4. *AspectLEDA* architecture, one aspect

Next, aspects must be described. They are expressed in LEDA as regular components (Fig. 3). Now the extended system can be described in *AspectLEDA*. Its description is simple (Fig. 4): A system in *AspectLEDA* is defined as a composed component -*composition* section-, constituted by two kinds of elements: system and aspects. They are defined as architectural components. The interaction between the system element and the aspects is described in the *attachments* section. Some parameters describing the new system architecture have been included between the attached elements representing the needed information (from the *CI* structure). Finally components and their interfaces are included in the *AspectLEDA* description.

Architecture Execution. *AspectLEDA* is supported by the Aspect Oriented Software Architecture Tool for LEDA (*AOSATool/LEDA*) which assists the software architect during the design process. He specifies the aspects and the *CI* structure to be included in the architecture; then, the tool generates *AspectLEDA* code. The tool also translates the *AspectLEDA* architecture into an equivalent LEDA one. Then it is possible to generate and run a Java program executing this architecture and thus to check if the obtained behaviour is the expected one. A formal Pi-calculus specification of the system can be obtained as well.

***AspectLEDA* Semantic.** The obtained AO-architecture is based on the use of the AO System Architecture Model (*AOSA Model*) [4], which allows aspects remain separated in the LEDA description. This approach is based on the principle which state that the separation of cross- cutting concerns can be reduced to a coordination prob lem. Thus, a coordination component can be used to coordinate the execution of the base and aspect codes sorting out all the actions performed by the system.

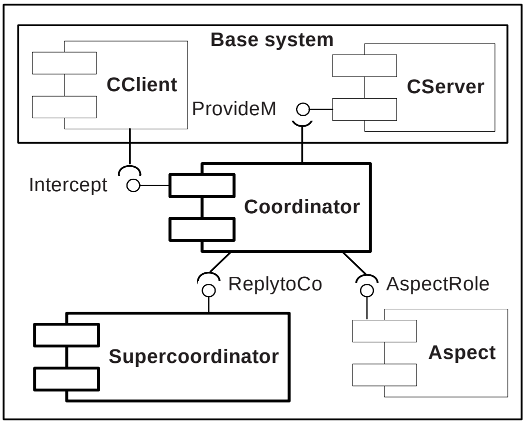


Fig. 5. AO-Architecture Graphical Representation

Fig. 5 shows a representation of the obtained LEDA architect- ure in which are represented the elements of the base system, the aspectual components and the elements proposed by the model:

1 A *Coordinator* component, which is generated for each Insertion Point (IP) on the base components, that is, for each operation of the base components that will be affected for an aspect. The *Coordinator,* on the one hand, detects when the operation designed by its IP is wanted to be executed and, on the other, orders to the associated aspect the execution of the corresponding operation[1]. The *Coordinator* manages the interactions considering the satisfied condition.

2 The *SuperCoordinator* is coordinating the coordinator, determining whether the conditions under which aspect can be executed are true or not. Before proceeding with the aspect execution, each *coordinator* asks the *SuperCoordinator*.

Acknowledgements

We would like to thank to the anonymous referees for the useful comments on this paper and the suggestions given by them.

[1] When several aspects affect the same IP, the coordinator manages the aspect priority.

References

1. Canal, C., et al.: Compatibility and Inheritance in Software Architecture. Science of Computing Programming 41(2), 105–130 (2001)
2. Chitchyan, R., et al.: Survey of Aspect-Oriented Analisis and Design Approaches. Document ID AOSD-Europe-ULANC-9 (May 2005)
3. Filman, R., et al.: Aspect-Oriented Software Development. Addison-Wesley, Reading (2005)
4. Navasa, A., et al.: Aspect Modelling at Architecture Design. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 41–48. Springer, Heidelberg (2005)

Experiences Using a Component-Oriented Architectural Framework for Robots and Its Improvement with a MDE Approach*

Francisco J. Ortiz, Juan A. Pastor, Diego Alonso, Bárbara Álvarez,
and Pedro Sánchez

Division of Electronics Engineering and Systems (DSIE)
Universidad Politécnica de Cartagena, Campus Muralla del Mar s/n
30202 Cartagena, Murcia Spain
francisco.ortiz@upct.es

Abstract. This paper describes the experience of the DSIE research group in the developing of the EFTCoR family of robots using an abstract architectural framework ACRoSeT, following the component-based paradigm. Using abstract components allow us to define very different architectures in a platform independent way. The translation of the abstract components to platform specific code is a hard and difficult task that can be partially automated with the help of the model transformation tools provided by the MDE approach.

Keywords: MDE, component-based software architecture, teleoperated robot.

1 Introduction

This paper describes the authors' experiences using software architectures in the development of teleoperated cranes and vehicles for ship hull cleaning in the context of the EFTCoR project. This development was specially challenging due to:

- The use of different execution platforms and different programming languages.
- Different functional requirements makes impossible to use a single architecture.

We needed a way to define different architectures sharing common components. With these ideas in mind, we defined *ACRoSeT* [1], an abstract architectural framework for the domain of teleoperated robots.

Teleoperated robotic systems cover a broad range of mechanisms that usually perform a small number of highly specialized tasks. Such specialization implies high variability that makes very difficult to design a single architecture flexible enough to deal with such heterogeneity. For this reason it is required a flexible and extensible architectural framework that (1) does not impose a concrete architecture, but allow defining different architectures, (2) allows reusing components in systems with

* This work was partially supported by the Spanish CICYT project MEDWSA, ref. TIC2006-15175-C05-02 and the Regional Government of Murcia Seneca Program, ref. 02998-PI-05.

F. Oquendo (Ed.): ECSA 2007, LNCS 4758, pp. 335 – 338, 2007.

different architectures, (3) allows the integration of components may be software or hardware, and (4) makes possible to integrate “intelligence,” or to interoperate with “intelligent systems”.

There have been numerous efforts to provide developers of software for robots with component frameworks to ease the development of robotic systems. Among these frameworks it is possible to highlight the following: OROCOS [3], CLARAty [9], MCA [6], ORCA [2], CARMEN [4] and PLAYER [8]. All of them make very valuable contributions that simplify the systems development.

2 Software Architecture for the Teleoperated Devices of the ETFCoR Family

The EFTCoR system comprises a family of teleoperated systems which mission is to retrieve and confine paint, oxide and marine adherences from ship hulls. The working environments are not fixed, there is a great variety of ship types, hull areas and shipyards characteristics, the systems consider different degrees of autonomy and different systems may have to work cooperatively at the same time.

ACRoSeT provides a common framework of abstract components to design software for teleoperated robots with very diverse behaviours. The subsystems defined by ACRoSeT are the following (see Fig. 1):

- *Coordination, Control and Abstraction Subsystem (CCAS): a*bstracts and encapsulates the functionality of the system physical devices.
- *Intelligence Subsystem* (*IS*): comprises the subsystems that provide intelligence to the global system. These systems are considered users of the *CCAS* functionality.
- *User Interaction Subsystem (UIS):* interprets, combines and arbitrates between orders that may come simultaneously from different users of the *CCAS*.
- *Safety, Management and Configuration Subsystem (SMCS):* Initializes, configures and manages the application.

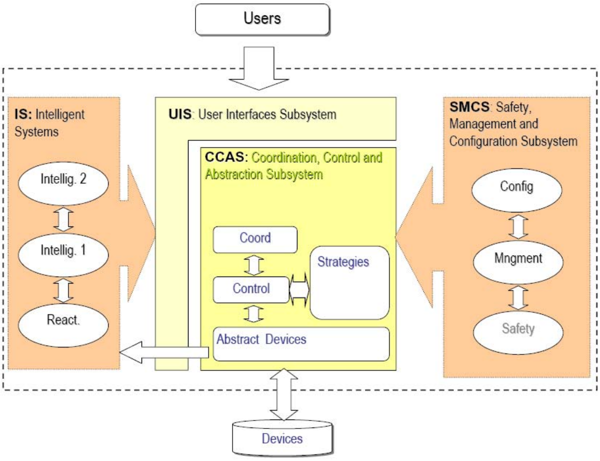

Users
IS: Intelligent Systems
Intellig. 2
Intellig. 1
React.
UIS: User Interfaces Subsystem
CCAS: Coordination, Control and Abstraction Subsystem
Coord
Control
Strategies
Abstract Devices
SMCS: Safety, Management and Configuration Subsystem
Config
Mngment
Safety
Devices

Fig. 1. An overview of the subsystems of ACRoSeT

The CCAS comprises components that are defined in four levels of granularity: (1) atomic components: abstract the characteristics of sensors and actuators, (2), Simple Controllers, (3) Mechanisms Controllers, and (3) Robot Controllers.

3 Instantiations of ACROSET for the EFTCoR Family

In response to the special industrial requirements of the EFTCoR project, the cranes (see Fig. 2) has been implemented using a PLC SIMATIC S7-300 and a Field-Bus (PROFIBUS-DP). The second instantiation is a caterpillar vehicle capable of scaling a hull thanks to permanent magnets (Fig. 3), carrying a manipulator that holds a cleaning tool. The execution platform is an on-board embedded PC with RTLinux Operating System.

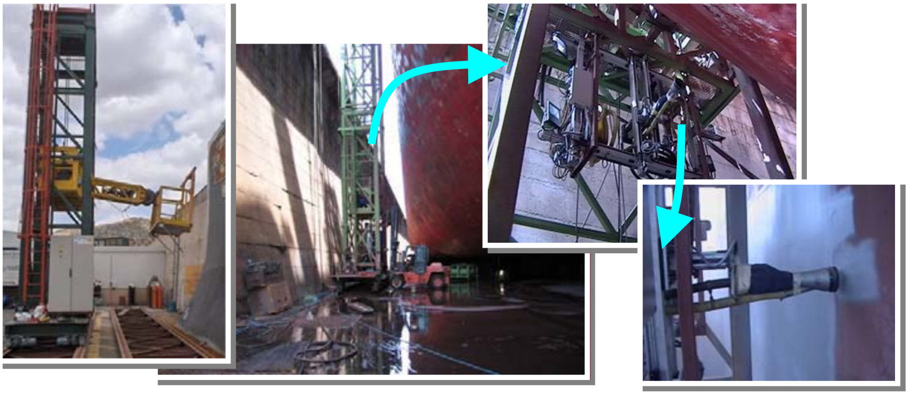


Fig. 2. XYZ table mounted on a crane. Tests in NAVANTIA shipyards.

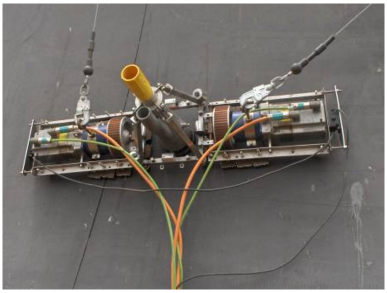


Fig. 3. Lazaro climbing vehicle

4 MDE

Model-Driven Engineering (MDE) [5] is an approach to software development in which *models* are first-class entities that guide each and every step of the design process. The other key concept in which rests MDE is *model transformation* [7].

We have adopted a MDE approach to develop the software architecture of robotic systems based on the abstract components proposed by *ACRoSeT*, using the *Eclipse* development environment and plug-ins. Different transformations make possible to map the *ACRoSeT* components to different platforms.

5 Conclusions

It is not possible to define a software architecture generic enough to be adapted to the entire domain, but usually there is no need to develop such architecture. The aim is to reuse components in different architectures and this is just what CBD and component frameworks propose.

Current component frameworks for robotic applications generally impose a concrete programming language and execution platform. As it is desirable to be able to define components that are independent of both system architecture and execution platform, ACROSET defines abstract components. However, the translation of the ACROSET abstract components into concrete, platform specific components is a difficult and error prone task. So, the ACROSET approach will only show its full potential if we are able to find a way to automatically translate abstract components into concrete components. The adoption of the MDE approach is a key step to achieve this goal.

References

1. Álvarez, B., Sánchez, P., Pastor, J.A., Ortiz, F.: An Architectural Framework for Modeling Teleoperated Service Robots. ROBOTICA 24(04), 411–418, Cambridge University Press, ISSN 0263-5747
2. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A.: Towards component-based robotics. In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, August 2-6, 2005, pp. 163–168 (2005)
3. Bruyninckx, H., Konincks, B., Soetens, P.: A Software Framework for Advanced Motion Control. Dpt. of Mechanical Engineering, K.U. Leuven. OROCOS project inside EURON. Belgium (2002)
4. Montemerlo, M., Roy, N., Thrun, S.: Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In: IEEE/RSJ Intl. Workshop on Intelligent Robots and Systems (2003)
5. Schmidt, D.: Model-Driven Engineering. IEEE Computer 39(2) (2006), doi:10.1109/MC. 2006.58, ISSN 0018-9162
6. Scholl, K.U., Albiez, J., Gassmann, B.: MCA: An Expandable Modular Controller Architecture, Karlsruhe University. 3rd Real-Time Linux Workshop, Milano, Italy (2001)
7. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20(5), 42–45 (2003), doi: 10.1109/MS. 2003.1231150
8. Vaughan, R., Gerkey, B., Howard, A.: On device abstractions for portable, reusable robot code. In: Proc. of the IEEE/RSJ Intl. Conf. On Intelligent Robots and Systems (IROS) (2003)
9. Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., Das, H.: The CLARAty architecture for robotic autonomy. In: IEEE Proceedings. Aerospace Conference, Montana, USA, vol. 1, pp. 121–132 (2001)

Author Index