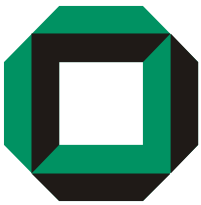
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Perspectives in Software Architecture Quality – Short papers of the 2nd International Conference on the Quality of Software Architectures (QoSA 2006), June 27-29, 2006, Västeras, Sweden

Herausgeber:
Christine Hofmeister, Ivica Crnkovic,
Ralf Reussner, Steffen Becker

Interner Bericht 2006-10



Universität Karlsruhe
Fakultät für Informatik

ISSN 1432 - 7864

Supplement to the Proceedings of the 2nd International Conference on the Quality of Software Architectures

Preface

Although the quality of a system's software architecture is one of the critical factors in its overall quality, the architecture is simply a means to an end, the end being the implemented system. Thus the ultimate measure of the quality of the software architecture lies in the implemented system, in how well it satisfies the requirements and constraints of the project and whether it can be maintained and evolved successfully. But in order to treat design as science rather than an art, we need the ability to address the quality of the software architecture directly, not simply as it is reflected in the implemented system.

This is the goal of QoSA: to address software architecture quality directly by addressing the problems of:

- designing software architectures of good quality,
- defining, measuring, evaluating architecture quality, and
- managing architecture quality, tying it upstream to requirements and downstream to implementation, and preserving architecture quality throughout the lifetime of the system.

Cross-cutting these problems is the question of the nature of software architecture. Software architecture organizes a system, partitioning it into elements and defining relationships among the elements. For this we often use multiple views, each with a different organizing principle.

But software architecture must also support properties that are emergent, that cannot be ascribed to particular elements. For this we often use the language of quality attributes. Quality attributes cover both internal properties, exhibited only in the development process (e.g. maintainability, portability, testability, etc.), and external properties, exhibited in the executing system (e.g. performance, resource consumption, availability, etc.). Quality attributes cover properties that are emergent, that have a pervasive impact, that are difficult to reverse, and that interact, thereby precluding or constraining other properties.

Thus in addition to examining software architecture quality, QoSA also aims to investigate quality attributes in the context of the problems of the design, evaluation, and management of software architecture. The papers selected for QoSA 2006 describe research and experience on these topics. Architecture evaluation is the most prevalent theme of the papers. The approaches varies from formal models to support evaluation to experience with process-centered approaches. The focus of the evaluation varies from evaluation of a particular quality attribute, such as performance or safety, to approaches where the evaluation covers a number of quality attributes, determined by the evaluator.

Another theme for QoSA 2006 is processes for supporting architecture quality. These papers go beyond the problem of evaluation to address software architecture quality at the process level. A final significant theme is the problem of managing and applying architectural knowledge.

Nineteen papers were presented at QoSA 2006. Twelve of these were long papers published in the conference proceedings. Seven of these describe interesting emerging results or novel case studies, and these are contained in this supplement to the proceedings. These papers have a stronger focus on novelty than on being long-running validated research. Therefore, they are part of this supplement, rather than the regular proceedings which are more concerned with established results. Also presented at QoSA were keynote addresses from Jan Bosch and Clemens Szyperski, and three tutorials on the QoSA themes.

We thank the members of the program committee and additional reviewers for their thorough, thoughtful, and timely reviews of the submitted papers. We also thank Sven Overhage and Judith Stafford for their work in supporting QoSA. Finally, we thank the generous sponsors of QoSA 2006: University of Karlsruhe (TH), Mälardalen Unverity, and Västerås City. These conference would not be possible without the support of all the above people and sponsors.

Ivica Crnkovic
Christine Hofmeister
Ralf Reussner
Steffen Becker

QoSA 2006 Organization

General Chair:
Ivica Crnkovic, Mälardalen University, Sweden

Program Committee Chair:
Christine Hofmeister, Lehigh University, USA

Steering Committee:
Ralf Reussner, University of Karlsruhe (TH), Germany
Judith Stafford, Tufts University, USA
Sven Overhage, Augsburg University, Germany
Steffen Becker, University of Karlsruhe (TH), Germany

Program committee:
Colin Atkinson, University of Mannheim, Germany
Len Bass, Software Engineering Institute, USA
Don Batory, University of Texas at Austin, USA
PerOlof Bengtsson, University of Karlskrona/Ronneby, Sweden
Jan Bosch, Nokia Research Center, The Netherlands
Alexander Brandle, Microsoft Research, United Kingdom
Michel Chaudron, Technische Universiteit Eindhoven, The Netherlands
Viktoria Firus, University of Oldenburg, Germany
Hassan Gomaa, George Mason University, USA
Ian Gorton, National ICT, Australia
Volker Gruhn, University of Leipzig, Germany
Wilhelm Hasselbring, University of Oldenburg / OFFIS, Germany
Jean-Marc Jezequel, University of Rennes / INRIA, France
Philippe Kruchten, University of British Columbia, Canada
Patricia Lago, Vrije Universiteit, The Netherlands
Nicole Levy, University of Versailles, France
Tomi Mannisto, Helsinki University of Technology, Finland
Raffaela Mirandola, Politecnico di Milano, Italy
Robert Nord, Software Engineering Institute, USA
Frantisek Plasil, Charles University, Czech Republic
Iman Poernomo, King's College, United Kingdom
Sasikumar Punnekkat, Märlardalen University, Sweden
Andreas Rausch, University of Kaiserslautern, Germany
Matthias Riebisch, Technical University of Ilmenau, Germany
Bernhard Rumpe, University of Technology Braunschweig, Germany
Chris Salzmann, BMW Car-IT
Jean-Guy Schneider, Swinburne University, Australia
Michael Stal, Siemens, Germany
Clemens Szyperski, Microsoft, USA
Hans van Vliet, Vrije Universiteit, The Netherlands

Wolfgang Weck, Independent Software Architect, Switzerland

Reviewers:
Sven Apel, University of Magdeburg, Germany
Roberto Lopez-Herrejon, University of Oxford, United Kingdom
Moreno Marzolla, INFN, Italy

Sponsors:
University of Karlsruhe (TH)
Mälardalen Unverity
Västerås City

Table of Contents

Jan Bosch, VP and head of the Software and Application Technologies Laboratory at Nokia Research Center

Expanding the scope of software product families: problems and alternative approaches

Abstract:
Software product families have found broad adoption in the embedded systems industry. Product family thinking has been prevalent in this context for mechanics and hardware and adopting the same for software has been viewed as a logical approach. During recent years, however, the trends of convergence, end-to-end solutions, shortened innovation and R&D cycles and differentiaton through software engineering capabilities have lead to a development where organizations are stretching the scope of their product families far beyond the initial design. Failing to adjust the product family approach, including the architectural and process dimensions when the business strategy is changing is leading to several challenging problems that can be viewed as symptons of this approach. The keynote discusses the key symptoms, the underlying causes for these symptons as well as solutions for realigning the product family approach with the business strategy.

Prof. dr. ir. **Jan Bosch** is a VP and head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Earlier, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include software architecture design, software product families, software variability management and component-oriented programming.
He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of several books and volumes in, among others, the Springer LNCS series and (co-)author of a significant number of research articles. He has been guest editor for journal issues, chaired several conferences as general and program chair, served on many program committees and organized numerous workshops. Finally, he is and has been a member of the steering groups of the GCSE and WICSA conferences.

Clemens Szyperski, Software Architect,Microsoft Research

Composing with Style - Components meet Architecture

Abstract:
Composability itself is probably the least composable term in the theory of computer science. In this talk, I'll explore some of thetroubling reasons why we have succeeded only so-so when it comes to the creation of composable software - and thus software components. Architecture can often come to the rescue, but only when applied with great style.

Clemens Szyperski joined Microsoft Research as a Software Architect in 1999. His team moved into a product incubation phase in 2001 and began production development in early 2003. A first product developed in an entirely new way will be released together with the upcoming Office System 2007. Since late 2005 he is now working on driving novel platform technology in Microsoft's new Connected Systems Division. His focus is on the end-to-end issues of leveraging component software to effectively build new kinds of software. He maintains an affiliation with Microsoft Research and continues his general activities in the wider research arena. His Jolt-award-winning book Component Software (Addison Wesley) appeared in a fully revised and extended second edition in late 2002. Software Ecosystem (MIT Press), co-authored with Dave Messerschmitt of UC Berkeley, was published in mid 2003. Clemens serves on numerous program committees, including ECOOP, ESEC/FSE, ICSE, and OOPSLA. He served as assessor and panelist for national funding bodies in Australia, Canada, Ireland, the Netherlands, and USA. He is a cofounder of Oberon microsystems, Zurich, Switzerland, and its now-public spin-off esmertec.

From 1994 to 1999, he was an associate professor at the School of Computer Science, Queensland University of Technology, Australia, where he retains an adjunct professorship. He held a postdoc scholarship at ICSI, affiliated with UC Berkeley in 1992/93. In 1992, he received his PhD in computer science from the Swiss Federal Institute of Technology (ETH) in Zurich under Prof. Niklaus Wirth and in 1987 his Masters in electrical engineering/ computer engineering from Aachen University of Technology (RWTH).

Judith Stafford, Senior Lecturer, Tufts University, Boston

Documentation Principles and Practices that You Can Live With

Abstract:
Software architecture has become a widely-accepted conceptual basis for the development of non-trivial software in all application areas and by organizations of all sizes. Effectively documenting an architecture is as important as crafting it; if the architecture is not understood, or worse, misunderstood, it cannot meet its goals as the unifying vision for software development. Development-based architecture strategies, such as Rational's Unified Process, stop short of prescribing documentation standards. The Views and Beyond approach to software architecture provides practical guidance on the what, why, and how of creating IEEE 1471-2000 compliant documentation for your software architecture that will be used for years to come. The approach is based on the well-known concept of views and is presented in the context of prevailing prescriptive models for architecture, including the Unified Process and UML 2.0, which has improved support for representing key architectural elements over its predecessors.

Attendee Background:
Participants should have experience with creating or using descriptions of large software systems and some knowledge of the Unified Modeling Language.

Tutorial Objectives:
The primary aim of this tutorial is to teach developers what constitutes good documentation of a software architecture, why it is worth the effort to create and maintain a documentation package, and how to write it down. A secondary aim is to teach other stakeholders why they should care about architectural documentation and how they can use it to make their life easier, increase productivity, and decrease overall system development and maintenance costs.

Judith Stafford is a Senior Lecturer in the Department of Computer Science at Tufts University, and is also a visiting scientist at the Software Engineering Institute, Carnegie Mellon University. Dr. Stafford has worked for several years in the area of compositional reasoning and its application to software architectures and component-based systems. She has organized workshops, given invited talks, taught tutorials, and written widely in these areas including co-authoring the book that inspired this tutorial, Documenting Software Archtiectures: Views and Beyond, Addison Wesley, 2002 and several book chapters on software architecture and component-based software engineering.

Prof. Dr. Ralf Reussner and Steffen Becker, University of Karlsruhe

Model-based Software Development with Eclipse

Abstract:
The tutorial consists of two parts. In the first part (45 min), Ralf Reussner focuses on the importance of an explicitly modelled software architecture. Besides an introduction into common architectural views, the role of the software architect is compared to "classical building" architects. As part of this, the often used comparison between building architecture and software architecture is critically reviewed. In particular, the role of an architect is discussed in model-driven software projects.

During the second part of the tutorial (135 min), Steffen Becker demonstrates online model driven development tools based on Eclipse. First, an introduction is given on the meta-modelling tools of the Eclipse Modelling Framework (EMF) and on the Graphical Modelling Framework (GMF) used to generate a domain specific editors for user defined (meta-)models. Additionally, the MDA framework of the OMG is presented and the concepts are applied to the introduced tools.

A live demonstration of the capabilities of the introduced tools for model transformations shows finally how a domain specific modelling tool can be generated to a large extend automatically using an EMF-model instance and the generator of GMF. As a result, an editor based on the Eclipse Graphical Editing Framework (GEF) can be deployed and run using Eclipse.

Professor **Ralf Reussner** holds the Chair for Software-Design and -Quality at the University of Karlsruhe since 2006. His research group is well established in the area of component based software design, software architecture and predictable software quality. Professor Reussner shaped this field not only by over 60 peer-reviewed publications in Journals and Conferences, but also by establishing various conferences and workshops. In addition, he acts as a PC member or reviewer of several conferences and journals. As Director of Software Engineering at the Informatics Research Centre in Karlsruhe (FZI) he consults various industrial partners in the areas of component based software, architectures and software quality. He is principal investigator or chief coodinator in several grants from industrial and governmental funding agencies. He graduated from University of Karlsruhe with a PhD in 2001. After this, Ralf was a Senior Research Scientist and project-leader at the Distributed Systems Technology Centre (DSTC Pty Ltd), Melbourne, Australia. From March 2003 till January held the Juniorprofessorship for Software Engineering at the University of Oldenburg, Germany, and was awarded with a 1 Mio EUR grant of the prestigious Emmy-Noether young researchers excellence programme of the National German Science Foundation.

Steffen Becker is a member of the research staff at the Chair for Software-Design and -Quality at the University of Karlsruhe since 2006. In his PhD thesis he concerned with combining model driven software development and prediction of the resulting Quality of Service properties of component based software systems. As part of his work he is working on a component model enabling the prediction of component based software systems. He is known in his field of research by several scientific publications and also as a member of the steering committee of the QoSA conference and the WCAT workshop series at ECOOP. He gained

practical experiences during his internship as software engineer in Johannesburg, ZA in 2000 as well as during consulting activities at the OFFIS in Oldenburg, Germany. He holds a diploma in business administration and computer science combined (Dipl.-Wirschaftsinformatik) from the Darmstadt University of Technology.

Heinz Züllighoven, Carola Lilienthal (University of Hamburg) and and Marcel Bennicke (Brandenburg University of Technology Cottbus)

Software Architecture Analysis and Evaluation

Abstract:
A software architecture describes the structure of a software system on an abstract implementation independent level. In forward engineering it serves as a blueprint to prescribe the intended software structure (so-called architecture model). In reverse engineering it can provide an abstract view of the actual code structure of the existing software system (so-called code architecture). Architecture models and actual code architectures play a vital role for all comprehension and communication tasks during the development and evolution of large software systems. Therefore, architecture models and code architectures have to be explicitly represented and consistently maintained during the development, maintenance, and reengineering processes.

The need to insure compliance of the architecture model and the actual code architecture has drawn considerable attention in recent years. In order to facilitate maintainability and enhancement of a software system the compliance of the architecture model and the actual code architecture is essential. Various tools have been developed to analyse and evaluate the deviation of code architecture and architecture model. In this tutorial we present static analysis tools that may be used for architectural analyses. We demonstrate how these tools can create useful architectural views for different evaluation tasks such as identification of reconstruction scope, critical architectural elements and potential design irregularities. If possible we will analyse a software system provided by a participant of the workshop in a life demonstration.

Heinz Züllighoven, graduted in Mathematics and German Language and Literature, holds a PhD in Computer Science. Since October 1991 he is professor at the Computer Science Department of the University of Hamburg and head of the attached Software Technology Centre. He is one of the original designers of the Tools & Materials approach to object-oriented application software and the Java framework JWAM, supporting this approach. Since 2000, Heinz Züllighoven is also one of the managing directors of C1 Workplace Solutions Ltd. He is consulting industrial software development projects in the area of object-oriented design, among which are several major banks. Heinz Züllighoven has published a number of papers and books on various software engineering topics. An English construction handbook for the Tools & Materials approach has been published by Morgan Kaufmann in 2004. Among his current research interests are agile object-oriented development strategies, migration processes and the architecture of large industrial interactive software systems. In addition, he an his co-researchers are further developing a light-weight modeling concept for business processes which is tool-supported.

Carola Lilienthal holds a Diploma degree in computer science from University of Hamburg (1995). She is a research assistant at the University of Hamburg and is working in the Software Engineering Group of Christiane Floyd and Heinz Züllighoven. Since 1995 she is also working as a consultant for object oriented design, software architecture, software quality, agile software development and participatory design in several industrial projects.

She has published a number of papers on various software engineering topics. Her research interests are the construction and analysis of large software systems, software architecture, software quality analysis and agile software development.

Marcel Bennicke holds a Diploma degree in computer science from Brandenburg University of Technology (2002). He is a research associate with the Software Systems Engineering Group at the same university. His research interests are software architecture, software quality and software quality analysis. Between 2004 and 2005 he has been working as a consultant in several industrial projects doing software quality analyses and introducing measurement programs in software development projects.

Merging In-House Developed Software Systems – A Method for Exploring Alternatives

Rikard Land, Jan Carlson, Ivica Crnković, Stig Larsson

Mälardalen University, Department of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
+46 21 10 70 35

{rikard.land, jan.carlson, ivica.crnkovic, stig.larsson}@mdh.se, http://www.idt.mdh.se/{~rld, ~jcn, ~icc}

Abstract

An increasing form of software evolution is software merge – when two or more software systems are being merged. The reason may be to achieve new integrated functions, but also remove duplication of services, code, data, etc. This situation might occur as systems are evolved in-house, or after a company acquisition or merger. One potential solution is to merge the systems by taking components from the two (or more) existing systems and assemble them into an existing system. The paper presents a method for exploring merge alternatives at the architectural level, and evaluates the implications in terms of system features and quality, and the effort needed for the implementation. The method builds on previous observations from several case studies. The method includes well-defined core model with a layer of heuristics in terms of a loosely defined process on top. As an illustration of the method usage a case study is discussed using the method.

1. Introduction

When organizations merge, or collaborate very closely, they often bring a legacy of in-house developed software systems. Often these systems address similar problems within the same business and there is usually some overlap in functionality and purpose. A new system, combining the functionality of the existing systems, would improve the situation from an economical and maintenance point of view, as well as from the point of view of users, marketing and customers. During a previous study involving nine cases of such in-house integration [10], we saw some drastic strategies, involving retiring (some of) the existing systems and reusing some parts, or only reutilizing knowledge and building a new system from scratch. We also saw another strategy of resolving this situation, which is the focus of the present paper: to merge the systems, by reassembling various parts from several existing system into a new system. From many points of view, this is a desirable solution, but based on previous research this is typically very difficult and is not so common in practice; there seem to be some prerequisites for this to be possible and feasible [10].

There is a need to relatively fast and accurately find and evaluate merge solutions, and our starting point to address this need has been the following previous observations [10]:

1. Similar high-level structures seem to be a prerequisite for merge. Thus, if the structures of the existing systems are not similar, a merge seems in practice unfeasible.
2. A development-time view of the system is a simple and powerful system representation, which lends itself to reasoning about project characteristics, such as division of work and effort estimations.
3. A suggested beneficial practice is to assemble the architects of the existing systems in a meeting early in the process, where various solutions are outlined and discussed. During this type of meeting, many alternatives are partly developed and evaluated until (hopefully) one or a few high-level alternatives are fully elaborated.
4. The merge will probably take a long time. To sustain commitment within the organization, and avoid too much of parallel development, there is a need to perform an evolutionary merge with stepwise deliveries. To enable this, the existing systems should be delivered separately, sharing more and more parts until the systems are identical.

This paper presents a systematic method for exploring merge alternatives, which takes these observations into account: by 1) assuming similar high-level structures, 2) utilizing static views of the systems, 3) being simple enough to be able to learn and use during the architects' meetings, and 4) by focusing not only on an ideal future system but also stepwise deliveries of the existing systems. The information gathered from nine

case studies was generalized into the method presented in this paper. To refine the method, we made further interviews with participants in one of the previous cases, which implemented the merge strategy most clearly.

The rest of the paper is organized as follows. We define the method in Section 2 and discuss it by means of an example in Section 3. Section 4 discusses important observations from the case and argues for some general advices based on this. Section 5 surveys related work. Section 6 summarizes and concludes the paper and outlines future work.

2. Software Merge Exploration Method

Our software merge exploration method consists of two parts: (i) a model, i.e., a set of formal concepts and definitions, and (ii) a process, i.e., a set of human activities that utilizes the model. The model is designed to be simple but should reflect reality as much as possible, and the process describes higher-level reasoning and heuristics that are suggested as useful practices.

To help explaining the method, we start with a simple example in Section 2.1, followed by a description of the method's underlying model (Section 2.2) and the suggested process (Section 2.3).

2.1 An Explanatory Example

Figure 1a shows two simple music sequencer software systems structured according to the "Model-View-Controller" pattern [2]. The recorded music would be the model, which can be viewed as a note score or as a list of detailed events, and controlled by mouse clicks or by playing a keyboard.

The method uses the module view [3,5] (or development view [8]), which describes modules and "use" dependencies between them. Parnas defined the "use" dependency so that module α is said to use module β if module α relies on the correct behavior of β to accomplish its task [14].

In our method, the term *module* refers to an encapsulation of a particular functionality, purpose or responsibility on an abstract level. A concrete implementation of this functionality is called a *module instance*. In the example, both systems have a EventView module, meaning that both systems provide this particular type of functionality (e.g., a note score view of the music). The details are probably different in the two systems, though, since the functionality is provided by different concrete implementations (the module instances $\mathsf{EventView}_A$ and $\mathsf{EventView}_B$, respectively). The method is not restricted to module instances that are present in the existing systems but also those that are possible in a future system; such new module instances could be either a planned implementation (e.g., $\mathsf{EventView}_{new\_impl}$), an already existing module to be reused in-house from some other program (e.g., $\mathsf{EventView}_{pgm\_name}$), or an open source or commercial component ($\mathsf{EventView}_{component\_name}$).

2.2 The Model

Our proposed method builds on a model consisting of three parts: a set of model elements, a definition of inconsistency in terms of the systems' structures, and a set of permissible user operations.

2.2.1 Concepts and Notation

The following concepts are used in the model:

- We assume there are two or more existing *systems*, (named with capital letters, and parameterized by X, Y, etc.).
- A *module* represents a conceptual system part with a specific purpose (e.g., EventView in Figure 1). Modules are designated with capital first letter; in the general case we use Greek letters α and β.
- A *module instance* represents a realization of a module. It is denoted α_X where α is a module and X is either an existing system (as in $\mathsf{EventView}_A$) or an indication that the module is new to the systems (as in $\mathsf{EventView}_{pgm\_name}$ or $\mathsf{EventView}_{component\_name}$).
- A *"use" dependency* (or *dependency* for short) from module instance α_X to module instance β_Y means that α_X relies on the correct behavior of β_Y to accomplish its task. We use the textual notation $\alpha_X \rightarrow \beta_Y$ to represent this.
- A *dependency graph* captures the structure of a system. It is a directed graph where each node in the graph represents a module instance and the edges (arrows) represent use dependencies. In Figure 1a, we have for example the dependencies $\mathsf{NoteView}_A \rightarrow \mathsf{MusicModel}_A$ and $\mathsf{MouseCtrl}_B \rightarrow \mathsf{MusicModel}_B$.
- An *adaptation* describes that a modification is made to α_X in order for it to be compatible, or *consistent* with β_Y, and is denoted $\langle \alpha_X, \beta_Y \rangle$ (see 2.2.2 below).
- A *scenario* consists of a dependency graph for each existing system and a single set of adaptations.

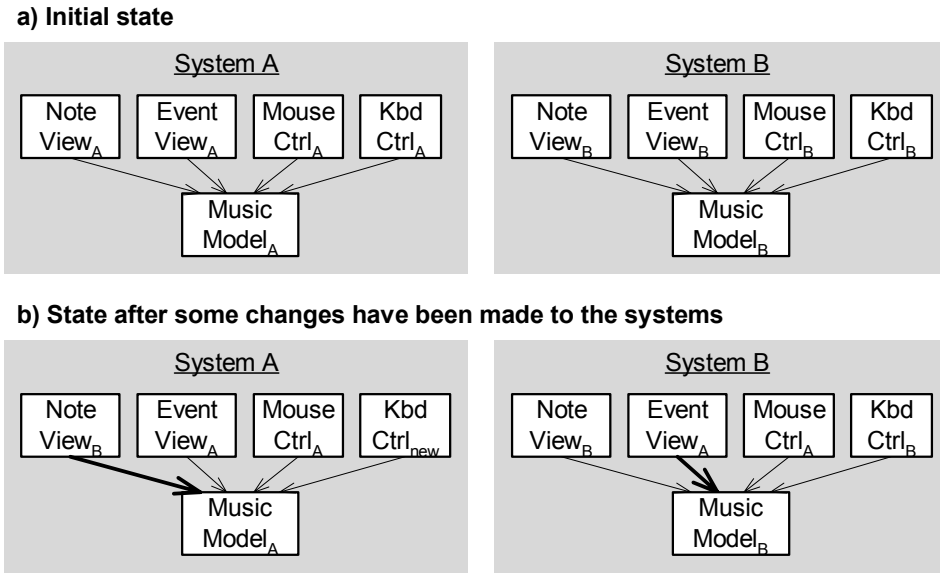


Figure 1. Two example systems with the same structure being merged.

2.2.2 Inconsistency

A dependency from α_X to β_Y can be *inconsistent*, meaning that β_Y cannot be used by α_X. Trivially, the dependency between two module instances from the same system is consistent without further adaptation. For the dependency between two modules from different systems we cannot say whether they are consistent or not. Most probably they are inconsistent, which has to be resolved by some kind of adaptation if we want to use them together in a new system. The actual adaptations made could in practice be of many kinds: some wrapping or bridging code, or modifications of individual lines of code; see further discussion in 4.1.

Formally, a dependency $\alpha_X \rightarrow \beta_Y$ is *consistent* if $X = Y$ or if the adaptation set contains $\langle \alpha_X, \beta_Y \rangle$ or $\langle \beta_Y, \alpha_X \rangle$. Otherwise, the dependency is *inconsistent*. A dependency graph is consistent if all dependencies are consistent; otherwise it is inconsistent. A scenario is consistent if all dependency graphs are consistent; otherwise it is inconsistent.

Example: The scenario in Figure 1b is inconsistent, because of the inconsistent dependencies from NoteView$_B$ to MusicModel$_A$ (in System A) and from EventView$_A$ to MusicModel$_B$ (in System B). The dependencies from KbdCtrl$_{new}$ to MusicModel$_A$ (in System A) and from MouseCtrl$_A$ to MusicModel$_B$ (in System B) on the other hand are consistent, since there are adaptations $\langle$KbdCtrl$_{new}$, MusicModel$_A\rangle$ and $\langle$MusicModel$_B$, MouseCtrl$_A\rangle$ representing that KbdCtrl$_{new}$ and MusicModel$_B$ have been modified to be consistent with MusicModel$_A$ and MouseCtrl$_A$ respectively.

2.2.3 Scenario Operations

The following operations can be performed on a scenario:

1. Add an adaptation to the adaptation set.
2. Remove an adaptation from the adaptation set.
3. Add the module instance α_X to one of the dependency graphs, if there exists an α_Y in the graph. Additionally, for each module β, such that there is a dependency $\alpha_Y \rightarrow \beta_Z$ in the graph, a dependency $\alpha_X \rightarrow \beta_W$ must be added for some β_W in the graph.
4. Add the dependency $\alpha_X \rightarrow \beta_W$ if there exist a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.
5. Remove the dependency $\alpha_X \rightarrow \beta_W$ if there exists a dependency $\alpha_X \rightarrow \beta_Z$ (with $Z \neq W$) in the graph.
6. Remove the module instance α_X from one of the dependency graphs, if there are no edges to α_X in the graph, and if the graph contains another module instance α_Y (i.e., with $X \neq Y$).

Note that these operations never change the participating modules of the graphs (if there is an α_X in the initial systems, they will always contain some α_Y). Similarly, dependencies between modules are also preserved Note also that we allow two or more instances for the same module in a system; when this could be suitable for a real system is discussed in 4.2.

2.3 The Process

The suggested process consists of two phases, the first consisting of two simple preparatory activities (P-I and P-II), and the second being recursive and exploratory (E-I – E-IV).

The scope of the method is within an early meeting of architects, where they (among other tasks) outline various merge solutions. To be able to evaluate various alternatives, some evaluation criteria should be provided by management, product owners, or similar stakeholders. Such criteria can include quality attributes for the system, but also considerations regarding development parameters such as cost and time limits. Other boundary conditions are the strategy for the future architecture and anticipated changes in the development organization. Depending on the circumstances, evaluation criteria and boundary conditions could be renegotiated to some extent, once concrete alternatives are developed.

2.3.1 Preparatory Phase

The *Preparatory* phase consists of two activities:

Activity P-I: Describe Existing Systems. First, the dependency graphs of the existing systems must be prepared, and common modules must be identified. These graphs could be found in existing models or documentation, or extracted by reverse engineering methods, or simply created by the architects themselves.

Activity P-II: Describe Desired Future Architecture. The dependency graph of the future system has the same structure, in terms of modules, as the existing systems. For some modules it may be imperative to use some specific module instance (e.g., α_X because it has richer functionality than α_Y, or a new implementation α_{new} because there have been quality problems with the existing α_X and α_Y). For other modules, α_X might be preferred over α_Y, but the final choice will also depend on other implications of the choice, which is not known until different alternatives are explored. The result of this activity is an outline of a desired future system, with some annotations, that serve as a guide during the exploratory phase. This should include some quality goals for the system as a whole.

2.3.2 Exploratory Phase

The result of the preparatory phase is a single scenario corresponding to the structure and module instances of the existing systems. The exploratory phase can then be described in terms of four activities: E-I "Introduce Desired Changes", E-II "Resolve Inconsistencies", E-III "Branch Scenarios", and E-IV "Evaluate Scenarios".

The order between them is not pre-determined; any activity could be performed after any of the others. They are however not completely arbitrary: early in the process, there will be an emphasis on activity E-I, where *desired changes* are introduced. These changes will lead to *inconsistencies that need to be resolved* in activity E-II. As the exploration continues, one will need to *branch scenarios* in order to explore different choices; this is done in activity E-III. One also wants to continually *evaluate the scenarios* and compare them, which is done in activity E-IV. Towards the end when there are a number of consistent scenarios there will be an emphasis on evaluating these deliveries of the existing systems. For all these activities, decisions should be described so they are motivated by, and traceable to, the specified evaluation criteria and boundary conditions. These activities describe high-level operations that are often useful, but nothing prohibits the user from carrying out any of the primitive operations defined above at any time.

Activity E-I: Introduce Desired Changes. Some module instances, desired in the future system, should be introduced into the existing systems. In some cases, it is imperative where to start (as described for activity P-II); the choice may e.g., depend on the local priorities for each system (e.g., "we need to improve the MusicModel of system A"), and/or some strategic considerations concerning how to make the envisioned merge succeed (e.g., "the MusicModel should be made a common module as soon as possible").

Activity E-II: Resolve Inconsistencies. As modules are exchanged in the graphs, dependencies $\alpha_X \rightarrow \beta_Y$ might become inconsistent. There are several ways of resolving these inconsistencies:

- Either of the two module instances could be modified to be consistent with the interface of the other. In the model, this means adding an adaptation to the adaptation set. In the example of Figure 1b, the inconsistency between NoteView_B and MusicModel_A in System A can be solved by adding either of the adaptations $\langle \text{NoteView}_B, \text{MusicModel}_A \rangle$ or $\langle \text{MusicModel}_A, \text{NoteView}_B \rangle$ to the adaptation set. (Different types of possible modifications in practice are discussed in Section 4.1.)
- Either of the two module instances could be exchanged for another. There are several variations on this:
  - A module instance is chosen so that the new pair of components is already consistent. This means that α_X is exchanged either for α_Y (which is consistent with β_Y as they come from the same system Y) or for some other α_Z for which there is an adaptation $\langle \alpha_Z, \beta_Y \rangle$ or $\langle \beta_Y,$

$\alpha_Z\rangle$. Alternatively, β_Y is exchanged for β_X or some other β_Z for which there is an adaptation $\langle\beta_Z, \alpha_X\rangle$ or $\langle\alpha_X , \beta_Z\rangle$. In the example of Figure 1b, MusicModel_A could be replaced by MusicModel_B to resolve the inconsistent dependency $\text{NoteView}_B \rightarrow \text{MusicModel}_A$ in System A.

  - A module instance is chosen that did not exist in either of the previous systems. This could be either of:
    i) a module reused in-house from some other program (which would come with an adaptation cost),
    ii) a planned or hypothesized new development (which would have an implementation cost, but low or no adaptation cost), or
    iii) an open source or commercial component (which involves acquirement costs as well as adaptation costs, which one would like to keep separate).

- One more module instance could be introduced for one of the modules, to exist in parallel with the existing; the new module instance would be chosen so that it already is consistent with the instance of the other module (as described for exchanging components). The previous example in Figure 1a and b is too simple to illustrate the need for this, but in Section 4 the industrial case will illustrate when this might be needed and feasible. Coexisting modules are also further discussed in Section 4.1.

Some introduced changes will cause new inconsistencies, that need to be resolved (i.e., this activity need to be performed iteratively).

Activity E-III: Branch Scenarios. As a scenario is evolved by applying the operations to it (most often according to either of the high-level approaches of activities E-I and E-II), there will be occasions where it is desired to explore two or more different choices in parallel. For example, several of the resolutions suggested in activity E-II might make intuitive sense, and both choices should be explored. It is then possible to copy the scenario, and treat the two copies as branches of the same tree, having some choices in common but also some different choices.

Activity E-IV: Evaluate Scenarios. As scenarios evolve, they need to be evaluated in order to decide which branches to evolve further and which to abandon. Towards the end of the process, one will also want to evaluate the final alternatives more thoroughly, and compare them – both with each other and with the pre-specified evaluation criteria and boundary conditions (which might at this point be reconsidered to some extent). The actual state of the systems must be evaluated, i.e., the actually chosen module instances plus the modifications to reduce inconsistencies). Do the systems contain many shared modules? Are the chosen modules the ones desired for the future system (richest functionality, highest quality, etc.)? Can the system as a whole be expected to meet its quality goals?

2.3.3 Accumulating Information

As these activities are carried out, there is some information that should be stored for use in later activities. As operations are performed, information is accumulated. Although this information is created as part of an operation within a specific scenario, the information can be used in all other scenarios; this idea would be particularly useful when implemented in a tool. We envision that any particular project or tool would define its own formats and types of information; in the following we give some suggestions of such useful information and how it would be used.

Throughout the exploratory activities, it would be useful to have some ranking of modules readily available, such as "EventView_A is preferred over EventView_B because it has higher quality". A tool could use this information to color the chosen modules to show how well the outlined alternatives fit the desired future system.

For activity E-II "Resolve Inconsistencies", it would be useful to have information about e.g., which module could or could not coexist in parallel. Also, some information should be stored that is related to how the inconsistencies are solved. There should at least be a short textual description of what an adaptation means in practice. Other useful information would be the efforts and costs associated with each acquirement and adaptation; if this information is collected by a tool, it becomes possible to extract a list of actions required per scenario, including the textual descriptions of adaptations and effort estimates. It is also possible to reason about how much of the efforts required that are "wasted", that is: is most of the effort related to modifications that actually lead towards the desired future system, or is much effort required to make modules fit only for the next delivery and then discarded? The evaluation criteria and boundary conditions mentioned in Section 2.2 could also be used by a tool to aid or guide the evaluation in the activity E-IV.

3. An Industrial Case Study

In a previous multiple case study on the topic of in-house integration, the nine cases in six organizations had implemented different integration solutions [10]. We returned to the one case that had clearly chosen the merge strategy and successfully implemented it

(although it is not formally released yet); in previous publications this case is labelled "case F2". The fact that this was one case out of nine indicates that the prerequisites for a merge are not always fulfilled, but also that they are not unrealistic (two more cases involved reusing parts from several existing systems in a way that could be described as a merge). To motivate the applicability of the proposed method, this section describes the events of an industrial case and places them in the context of our method.

3.1 Research Method

This part of the research is thus a single case study [17]. Our sources of information have been face-to-face interviews with the three main developers on the US side (there is no title "architect" within the company) and the two main developers on the Swedish side, as well as the high-level documentation of the Swedish system. All discussion questions and answers are published together with more details on the study's design in a technical report [9].

Although the reasoning of the case follows the method closely, the case also demonstrates some inefficiency due to not exploring the technical implications of the merge fully beforehand. It therefore supports the idea of the method being employed to analyze and explore merge alternatives early, before committing to a particular strategy for the in-house integration (merge or some other strategy).

3.2 The Case

The organization in the case is a US-based global company that acquired a slightly smaller global company in the same business domain, based in Sweden. To support the core business, computer simulations are conducted. Both sites have developed software for simulating 3D physics, containing state-of-the-art physics models, many of the models also developed in-house.

As the results are used for real-world decisions potentially affecting the environment and human lives, the simulation results must be accurate (i.e., the output must correspond closely to reality). As the simulations are carried out off-line and the users are physics specialists, many other runtime quality properties of the simulation programs are not crucial, such as reliability (if the program crashes for a certain input, the bug is located and removed), user-friendliness, performance, or portability. On the other side, the accuracy of the results are crucial.

Both systems are written in Fortran and consist of several hundreds of thousands lines of code, and the staff responsible for evolving these simulators are the interviewees, i.e., less than a handful on each site. There was a strategic decision to integrate or merge the systems in the long term. This should be done through cooperation whenever possible, rather than as a separate up-front project.

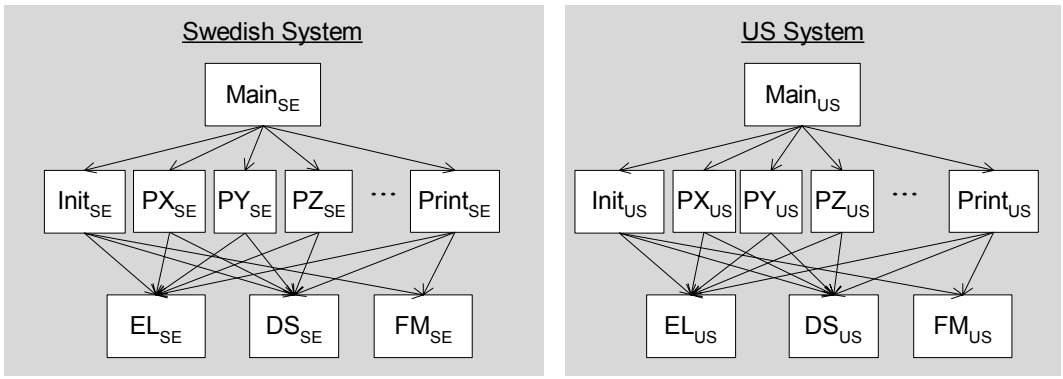
The rest of this section describes the events of the case in terms of the proposed activities of the method. It should be noted that although the interviewees met in a small group to discuss alternatives, they did not follow the proposed method strictly (which is natural, as the method has been formulated after, and partly influenced by, these events).

Activity P-I: Describe Existing Systems. Both existing systems are written in the same programming language (Fortran), and it was realized early that the two systems have very similar structure, see Figure 2a). There is a main program (Main) invoking a number of physics modules (PX, PY, PZ, …) at appropriate times, within two main loops. Before any calculations, an initialization module (Init) reads data from input files and the internal data structures (DS) are initialized. The physics modeled is complex, leading to complex interactions where the solution of one module affects others in a non-hierarchical manner. After the physics calculations are finished, a file management module (FM) is invoked, which collects and prints the results to file. All these modules use a common error handling and logging library (EL), and share the same data structures (DS). A merge seemed plausible also thanks to the similarities of the data models; the two programs model the same reality in similar ways.

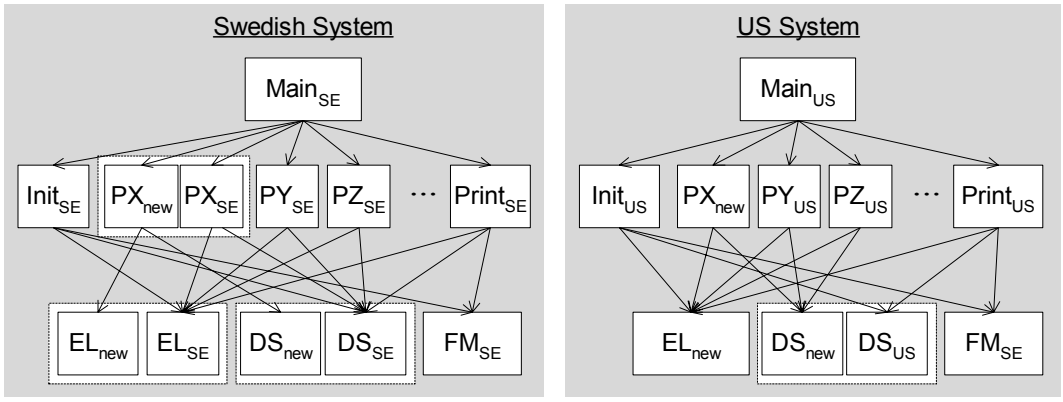
Activity P-II: Describe Desired Future Architecture. The starting point was to develop a common module for one particular aspect of the physics (PX_{new}), as both sides had experienced some limitations of their respective current physics models. Now being in the same company, it was imperative that they would join efforts and develop a new module that would be common to both programs; this project received some extra integration funding. Independent of the integration efforts, there was a common wish on both sides to take advantage of newer Fortran constructs to improve encapsulation and enforce stronger static checks.

Activity E-I: Introduce Desired Changes. As said, the starting point for integration was the module PX. Both sides wanted a fundamentally new physics model, so the implementation was also completely new (no reuse), written by one of the Swedish developers. The two systems also used different formats for input and output files, managed by file handling modules (FM_{SE} and FM_{US}). The US system chose to incorporate the Swedish module for this, which has required some changes to the modules using the file handling module.

a) Initial state



b) Current state



Adaptation set: <$Main_{SE}$, PX_{new}> <PX_{new}, EL_{new}> <PX_{new}, DS_{new}> <$Main_{US}$, PX_{new}> <PY_{US}, EL_{new}> <PY_{US}, DS_{new}> <PZ_{US}, EL_{new}> <PZ_{US}, DS_{new}>

c) Future System

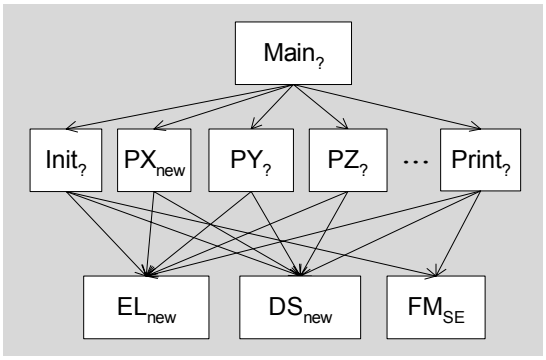


Figure 2: The current status of the systems of the case.

Activity E-II: Resolve Inconsistencies. The PX module of both systems accesses large data structures (DS) in global memory, shared with the other physics modules. An approach was tried where adapters were introduced between a commonly defined interface and the old implementations, but was abandoned as this solution became too complex. Instead, a new implementation of data structures was introduced. This was partially chosen because it gave the opportunity to use newer Fortran constructs which made the code more structured, and it enabled some encapsulation and access control as well as stronger type checking than before.

This led to new inconsistencies that needed to be resolved. In the US system, six man-months were spent on modifying the existing code to use the new data structures. The initialization and printout modules remained untouched however; instead a solution was

chosen where data is moved from the old structures (DS_{SE} and DS_{US}) to the new (DS_{new}) after the initialization module has populated the old structures, and data is moved back to the old structures before the printout module executes. In the Swedish system, only the parts of the data structures that are used by the PX module are utilized, the other parts of the program uses the old structures; the few data that are used both by the PX module and others had to be handled separately.

The existing libraries for error handling and logging (EL) would also need some improvements in the future. Instead of implementing the new PX module to fit the old EL module, a new EL module was implemented. The new PX module was built to use the new EL module, but the developers saw no major problems to let the old EL module continue to be used by other modules (otherwise there would be an undesirable ripple effect). However, for each internal shipment of the PX module, the US staff commented away the calls to the EL library; this was the fastest way to make it fit. In the short term this was perfectly sensible, since the next US release would only be used for validating the new model together with the old system. However, spending time commenting away code was an inefficient way of working, and eventually the US site incorporated the EL library and modified all other modules to use it; this was not too difficult as it basically involved replacing certain subroutine calls with others. In the Swedish system, the new EL library was used by the new PX module, while the existing EL module was used in parallel, to avoid modifying other modules that used it. Having two parallel EL libraries was not considered a major quality risk in the short run.

Modifying the main loop of each system, to make it call the new PX module instead of the old, was trivial. In the Swedish system there will be a startup switch for some years to come, allowing users to choose between the old and the new PX module for each execution. This is useful for validation of PX_{new} and is presented as a feature for customers.

E-III Branch Scenarios. As we are describing the actual sequence of events, this activity cannot be reported as such, although different alternatives were certainly discussed – and even attempted and abandoned, as for the data structure adapters.

E-IV Evaluate Scenarios. This activity is also difficult to isolate after the fact, as we have no available reports on considerations made. It appears as functionality was a much more important factor than non-functional (quality) attributes at the module level. At system level, concerns about development time qualities (e.g., discussions about parallel module instances and the impact on maintenance) seem to have been discussed more than runtime qualities (possibly because runtime qualities in this case are not crucial).

Figure 2 shows the initial and current state of the systems, as well as the desired outlined future system. (It is still discussed whether to reuse the module from either of the systems or create a new implementation, hence the question marks).

4. Discussion

This section discusses various considerations to be made during the exploration and evaluation, as highlighted by the case.

4.1 Coexisting Modules

To resolve an inconsistency between two module instances, there is the option of allowing two module instances (operation 2). Replacing the module completely will have cascading effects on the consistencies for all edges connected to it (both "used-by" and "using"), so having several instances has the least direct impact in the model (potentially the least modification efforts). However, it is not always feasible in practice to allow two implementations with the same purpose. The installation and runtime costs associated with having several modules for the same task might be prohibiting if resources are scarce. It might also be fundamentally assumed that there is only one single instance responsible for a certain functionality, e.g., for managing central resources. Examples could be thread creation and allocation, access control to various resources (hardware or software), security, etc. Finally, at development time, coexisting components violates the conceptual integrity of the system, and results in a larger code base and a larger number of interfaces to keep consistent during further evolution and maintenance. From this point of view, coexisting modules might be allowed as a temporary solution for an intermediate delivery, while planning for a future system with a single instance of each module (as in the case for modules EL and DS). However, the case also illustrates how the ability to choose either of the two modules for each new execution was considered useful (PX_{SE} and PX_{new} in the Swedish system).

We can see the following types of relationships between two particular module instances of the same module:

- **Arbitrary usage.** Any of the two parallel modules may be invoked at any time. This seems applicable for library type modules, i.e., modules that retains no state but only performs some action and returns, as the EL module in the case.
- **Alternating usage.** If arbitrary usage cannot be allowed, it might be possible to define some rules

for synchronization that will allow both modules to exist in the system. In the case, we saw accesses to old and new data structures in a pre-defined order, which required some means of synchronizing data at the appropriate points in time. One could also imagine other, more dynamic types of synchronization mechanisms useful for other types of systems: a rule stating which module to be called depending on the current mode of the system, or two parallel processes that are synchronized via some shared variables. (Although these kinds of solutions could be seen as a new module, the current version of the method only allows this to be specified as text associated to an adaptation.)

- **Initial choice.** The services of the modules may be infeasible to share between two modules, even over time. Someone will need to select which module instance to use, e.g., at compile time by means of compilation switches, or with an initialization parameter provided by the user at run-time. This was the case for the PX_{SE} and PX_{new} modules in the Swedish system.

The last two types of relationships requires some principle decision and rules at the system (architectural) level, while the signifying feature of the first is that the correct overall behaviour of the program is totally independent of which module instance is used at any particular time.

4.2 Similarity of Systems

As described in 2.1.1, the model requires that the structures of the existing systems are identical, which may seem a rather strong assumption. It is motivated by the following three arguments [10]:

- The previous multiple case study mentioned in Section 3.1 strongly suggests that similar structures is a prerequisite for merge to make sense in practice. That means that if the structures are dissimilar, practice has shown that some other strategy will very likely be more feasible (e.g., involving the retirement of some systems). Consequently, there is little motivation to devise a method that covers also this situation.
- We also observed that it is not so unlikely that systems in the same domain, built during the same era, indeed have similar structures.
- If the structures are not very similar at a detailed level, it might be possible to find a higher level of abstraction where the systems are similar.

A common type of difference, that should not pose large difficulties in practice, is if some modules and dependencies are similar, and the systems have some modules that are only extensions to a common architecture. For example, in the example system one of the systems could have an additional View module (say, a piano roll visualization of the music); in the industrial case we could imagine one of the systems to have a module modeling one more aspect of physics (PW) than the other. However, a simple workaround solution in the current version of the method is to introduce virtual module instances, i.e., modules that do not exist in the real system (which are of course not desired in the future system).

5. Related Work

There is much literature to be found on the topic of *software integration*. Three major fields of software integration are component-based software [16], open systems [13], and Enterprise Application Integration, EAI [15]. However, we have found no existing literature that directly addresses the context of the present research: integration or merge of software *controlled and owned within an organization*. These existing fields address somewhat different problems than ours, as these fields concern components or systems *complementing* each other rather than systems that *overlap* functionally. Also, it is typically assumed that components or systems are acquired from third parties and that modifying them is not an option, a constraint that does not apply to the in-house situation. *Software reuse* typically assumes that components are initially built to be reused in various contexts, as COTS components or as a reuse program implemented throughout an organization [7], but in our context the system components were likely not being built with reuse in mind.

It is commonly expressed that a software architecture should be documented and described according to different views [3,5,6,8]. One frequently proposed view is the module view [3,5] (or development view [8]), describing development abstractions such as layers and modules and their relationships. The dependencies between the development time artifacts were first defined by Parnas [14] and are during ordinary software evolution the natural tool to understand how modifications made to one component propagate to other.

The notion of "architectural mismatch" is well known, meaning the many types of incompatibilities that may occur when assembling components built under different assumptions and using different technologies [4]. There are some methods for automatically merging software, mainly source code [1], not least in the context of configuration management systems [12]. However, these approaches are unfeasible for merging large systems with complex requirements, functionality, quality, and stakeholder interests. The abstraction level must be higher.

6. Conclusions and Future Work

The problem of integrating and merging large complex software systems owned in-house is essentially unexplored. The method presented in this paper addresses the problem of rapidly outlining various merge alternatives, i.e., exploring how modules could be reused across existing systems to enable an evolutionary merge. The method makes visible various merge alternatives and enables reasoning about the resulting functionality of the merged system as well as about the quality attributes of interest (including both development time and runtime qualities).

The method consists of a formal model with a loosely defined heuristics-based process on top. The goal has been to keep the underlying model as simple as possible while being powerful enough to capture the events of a real industrial case. One of the main drivers during its development has been simplicity, envisioned to be used as a decision support tool at a meeting early in the integration process, with architects of the existing systems. As such, it allows rapid exploration of multiple scenarios in parallel. We have chosen the simplest possible representation of structure, the module view. For simplicity, the method in its current version mandates that the systems have identical structures. This assumption we have shown is not unreasonable but can also be worked around for minor discrepancies. The method is designed so that stepwise deliveries of the existing systems are made, sharing more and more modules, to enable a true evolutionary merge.

Assisted by a tool, it would be possible to conveniently record information concerning all decisions made during the exploration, for later processing and presentation, thus giving an advantage over only paper and pen. We are implementing such a tool, which already exist as a prototype [11]. It displays the graphs of the systems, allows user-friendly operations, highlights inconsistencies with colors, and is highly interactive to support the explorative process suggested. The information collected, in the form of short text descriptions and effort estimations, enables reasoning about subsequent implementation activities. For example, how much effort is the minimum for a first delivery where some module is shared? What parts of a stepwise delivery are only intermediate, and how much effort is thus wasted in the long term?

There are several directions for extending the method: First, understanding and bridging differences in existing data models and technology frameworks of the existing systems is crucial for success and should be part of a merge method. Second, the model could be extended to allow a certain amount of structural differences between systems. Third, the module view is intended to reveal only static dependencies, but other types of relationships are arguably important to consider in reality. Therefore, we intend to investigate how the method can be extended to include more powerful languages, including e.g., different dependency types and different adaptation types, and extended also to other views.

6.1 Acknowledgements

We would like to thank all interviewees and their organization for sharing their experiences and allowing us to publish them. Also thanks to Laurens Blankers for previous collaboration that has led to the present paper, and for our discussions on architectural compatibility.

7. References

[1] Berzins V., "Software merge: semantics of combining changes to programs", In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16, issue 6, pp. 1875-1903, 1994.

[2] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, John Wiley & Sons, 1996.

[3] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., and Stafford J., *Documenting Software Architectures: Views and Beyond*, ISBN 0-201-70372-6, Addison-Wesley, 2002.

[4] Garlan D., Allen R., and Ockerbloom J., "Architectural Mismatch: Why Reuse is so Hard", In *IEEE Software*, volume 12, issue 6, pp. 17-26, 1995.

[5] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, 2000.

[6] IEEE Architecture Working Group, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE, 2000.

[7] Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.

[8] Kruchten P., "The 4+1 View Model of Architecture", In *IEEE Software*, volume 12, issue 6, pp. 42-50, 1995.

[9] Land R., *Interviews on Software Systems Merge*, MRTC report, Mälardalen Real-Time Research Centre, Mälardalen University, 2006.

[10] Land R. and Crnkovic I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", In *Information & Software Technology*, Accepted for publication, 2006.

[11] Land R. and Lakotic M., "A Tool for Exploring Software Systems Merge Alternatives", In *Proceedings of International ERCIM Workshop on Software Evolution* , 2006.

[12] Mens T., "A state-of-the-art survey on software merging", In *IEEE Transactions on Software Engineering*, volume 28, issue 5, pp. 449-462, 2002.

[13] Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.

[14] Parnas D. L., "Designing Software for Ease of Extension and Contraction", In *IEEE Transaction on Software Engineering*, volume SE-5, issue 2, pp. 128-138, 1979.

[15] Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.

[16] Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.

[17] Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

Using Architectural Decisions

Jan S. van der Ven, Anton Jansen, Paris Avgeriou, and Dieter K. Hammer
University of Groningen, Department of Mathematics and Computing Science,
PO Box 800, 9700AV Groningen, The Netherlands,
[salvador|anton|paris|dieter]@cs.rug.nl,
WWW home page: http://search.cs.rug.nl

***Abstract*— There are increasing demands for the explicit representation and subsequent sharing and usage of architectural decisions in the software architecting process. However, there is little known on how to use these architectural decisions, or what type of stakeholders need to use them. This paper presents a use case model that arose from industrial needs, and is meant to explore how these needs can be satisfied through the effective usage of architectural decisions by the relevant stakeholders. The use cases are currently being validated in practice through industrial case studies. As a result of this validation, we argue that the usage of architectural decisions by the corresponding stakeholders can enhance the quality of software architecture.**

I. Introduction

One of the proposed ways to advance the quality of software architecture is the treatment of architectural decisions [1], [2], [3], [4] as first-class entities and their explicit representation in the architectural documentation. From this point of view, a software system's architecture is no longer perceived as interacting components and connectors, but rather as a set of architectural decisions [5]. The main reason that this paradigm shift improves the quality of software architecture is that it reduces *Architectural Knowledge Vaporization* [1], [5]. It is presently not possible to completely eliminate vaporization, as the result still depends on the judgment and the chosen tradeoffs that the architect makes.

Architectural knowledge vaporizes because most of the *architectural decisions*, which are the most significant form of architectural knowledge [6], are lost during the development and evolution cycles. This is due to the fact that architectural decisions are neither documented in the architectural document, nor can they be explicitly derived from the architectural models. They merely exist in the form of tacit knowledge in the heads of architects or other stakeholders, and thus inevitably dissipate. The only way to resolve this problem is to grant architectural decisions first-class status and properly integrate them within the discipline of software architecture.

Although the domain of architectural decisions is receiving increasing attention by the software architecture community, there is little guidance as to how architectural decisions can be used during both architecting and in the general development process. In fact, in order to give architectural decisions first-class status there should be a systematic approach that can support their explicit documentation and usage by the architect and the rest of the stakeholders. We believe that it is too early at this stage to introduce methods or processes, and even more so supporting systems for creating and subsequently using architectural decisions. We argue that, first, we need to understand the complex nature of architectural decisions, and their role in the software development process and within an organization.

To achieve this goal, we present in this paper a use case model, that elaborates on two important issues: first, which are the stakeholders that need to use architectural decisions; second, how can the decisions be used by the relevant stakeholders. We have worked with industrial partners to understand the exact problems they face with respect to loss of architectural decisions. We have thus compiled a wish list from practitioners on the use of architectural decisions. Furthermore we have combined this wishlist with what we consider as the ideal requirements for a system that supports the usage of architectural decisions. We have thus followed both a bottom-down and a top-down approach and finally merged them into a use case model that represents real and ideal industrial needs for effective usage of architectural decisions. Finally, we validated this use case model in practice, by applying it at the industrial partners in small case studies.

The idea of a use case model for using architecture knowledge was introduced in [2], which forms the foundation work for our paper. This idea is further elaborated in [7]. It discusses the concept of architectural knowledge, from an ontological perspective, and typical usages of architectural knowledge in a broad context.

The rest of the paper is structured as follows: in Section 2 we give an overview of how our industrial partners defined the needs for using and sharing architectural decisions. Section 3 presents the use case model, including the actors and system boundary. The ongoing validation of the use cases is conducted in Section 4. Section 5 discusses related work in this field and Section 6 sums up with conclusions and future work.

II. From industrial needs to Use Cases

The use cases that are described in the rest of the paper refer to a potential system that would support the management of architectural decisions. To the best of our knowledge, there is no such system implemented yet. We envision this system as a Knowledge Grid [8]: "an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources".

Before pinpointing the specific requirements for a knowledge grid in the next sections, it is useful to consider the more

generic requirements by combining the areas of knowledge grids and architectural decisions. First, this system should support the effective collaboration of teams, problem solving, and decision making. It should also use ontologies to represent the complex nature of architectural decisions, as well as their dense inter-dependencies. Furthermore, it must effectively visualize architectural decisions and their relations from a number of different viewpoints, depending on the stakeholders' concerns. Finally, it must be integrated with the tools used by architects, as it must connect the architectural decisions to documents written in the various tools or design environments, and thus provide traceability between them.

We are currently participating in the Griffin project [9] that is working on tools, techniques and methods that will perform the various tasks needed for building this knowledge grid. Until now, the project has produced two main results: a use case model, and a domain model. The domain model describes the basic concepts for storing, sharing, and using architectural decisions and the relationships between those concepts [10]. The use case model describes the required usages of the envisioned knowledge grid. The use cases are expressed in terms of the domain model, in order to establish a direct link between the concepts relevant to architectural decisions (the domain model), and how the architectural decisions should be used (use cases). The focus in this paper is on the use case model.

Four different industrial partners participate in the Griffin project. They are all facing challenges associated to architectural knowledge vaporization. Although the companies are of different nature, they all are involved in constructing large software-intensive systems. They consider software architecture of paramount importance to their projects, and they all use highly sophisticated techniques for maintaining, sharing and assessing software architectures. Still, some challenges remain.

We conducted qualitative interviews with 14 employees of these industrial partners. Our goal was to analyze the problems they faced concerning sharing architectural knowledge, and to identify possible solutions to these problems. People with different roles were interviewed: architects (SA), project managers (PM), architecture reviewers (AR), and software engineers (SE). A questionaire (see the appendix at the end of this paper) was used to streamline and direct the interviews. The questionaire was not directly shown to the employees, but used as a starting point and checklist for the interviewers.

The results from the interviews were wrapped up in interview reports that described the current challenges and envisioned solutions by these companies. The interview reports contained some needs from the interviewees, which included:

1) Find relevant information in large architectural descriptions (SA, PM, AR).
2) Add architectural decisions, relate them to other architectural knowledge like architectural documentation, or requirement documentation (SA, PM).
3) Search architectural decisions and the underlying reasons, construct (multiple) views where the decisions are represented (SA, PM, AR).
4) Identify what knowledge should minimally be made available to let developers work effectively (SA).
5) Identify the changes in architectural documentation (PM).
6) Identify what architectural decisions have been made in the past, to avoid re-doing the decision process. This include identifying what alternatives were evaluated and the issues that played some critical role at that time (SA, PM, AR, SE).
7) Reuse architectural decisions (SA, PM, SE).
8) Keep architecture up-to-date during development and evolution (SA, PM).
9) Get an overview of the architecture (SA, PM, AR).

The interview reports and the needs stated in these reports form the starting point for constructing the use cases. Except for this bottom-up approach we also followed a top-down approach: we thought about the ideal usages of a system that supports the usage of architectural knowledge. This was necessary as most of the interviewees had a rather implicit and vague notion of architectural decisions and had not thought of using architectural decisions, represented as first-class entities. Both real needs from the interviewees and ideal needs proposed by the research group were merged into a use case model presented in the next section.

III. The Use Case Model

This section elaborates on a set of use cases that roughly define the requirements for a potential knowledge grid. First, we describe the actors of the knowledge grid, starting from the roles of our interviewees. After this, the primary actor and the scope are discussed. To understand the dependencies between the use cases a use case model consisting of 27 use cases, including the relations, is presented in figure 1. Besides presenting the dependencies among the use cases, the figure also relates the use cases to the identified needs described in the previous section. Note that this is not a one-to-one mapping; some needs resulted in multiple use cases and few use cases do not originate from needs but from 'ideal' requirements.

A. Actors

We identified the following actors being relevant for the use cases, based on the roles of the interviewees.

- *Architect.* Architects should be able to create and manage an architecture, and get an overview of the status of the architecture. This results in demands for views that show the coverage of requirements or describe the consistency of the design. Also, the architect is responsible for providing stakeholders with sufficient information, to ensure that their concerns are met in the architecture design.
- *Architecture Reviewer.* Architecture reviewers are often interested in a specific view on the architecture. They can be colleagues, experts from a certain field, or reviewers from an external organization. They want to understand the architecture quickly and want to identify potential pitfalls in the architecture, like poorly founded architectural

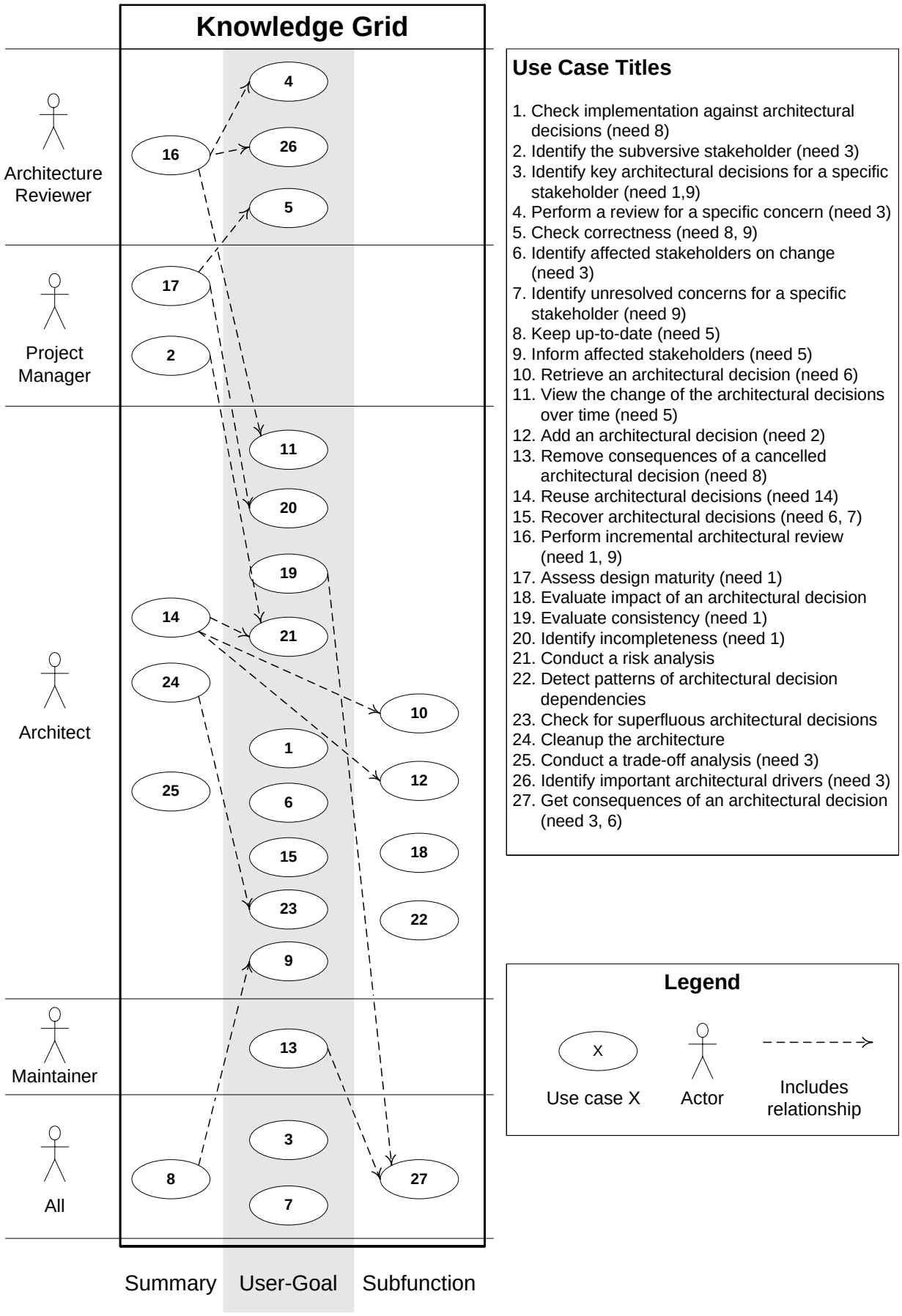


Fig. 1. Use case diagram

decisions, architectural incompleteness, or architectural inconsistency.

- *Project Manager.* The concerns of the project manager are usually driven by the planning; what is the status of the architecture, are there potential upcoming problems or risks, and how can we address them? The project manager also addresses people-related issues, e.g. which stakeholder is the biggest risk for the architecture?
- *Developer.* The primary concern of the developer is that the architecture should provide sufficient information for implementing the system. The descriptions must be unambiguous. Also, the developer must know where to look for the necessary knowledge; this can be in the architectural documentation, or by knowing which person to contact.
- *Maintainer.* The maintainer is often ignored as a stakeholder of an architecture. However, the maintainer is one of the most important actors when the architecture has to evolve. The maintainer has interest in the evolution of the architecture (up-to date information), and the consequences of changes in the architecture.

We encountered that the different companies used different terms for the roles they have in the software development process. The list of actors presented above is an abstraction of those different roles.

B. Describing the use cases

We present the use cases, as mandated in [11], using the following elements:

- *Scope.* All the use cases are defined as an interaction on a knowledge grid type of system (see section 2.1). From the use case model perspective, this system is considered a black-box system.
- *Goal level.* The descriptions from the interviews were very diverse in detail. As a consequence, some use cases describe a single interaction on the system (e.g. add an architectural decision), while others are very high-level demands of the system (e.g. perform an incremental architectural review). We adopted three goal levels from [11] of a decreasing abstraction: Summary, User-goal and Subfunction, for describing this difference. A Summary goal use case can involve multiple User-goals use cases, and often have a longer time span (hours, days). A User-goal use case involves a primary actor using the system (in Business Process Management often called elementary business process), often in one session of using the system. Subfunction use cases are required to carry out User-goal use cases. They typically represent an elementary action on the system, which is used by multiple User-goal use cases.
- *Primary actor.* The list of actors described in section III-A are used to determine the primary actor for a specific use case. Sometimes, a use case can be performed by all actors (e.g. identify key architectural decisions). In these cases, the term All is used as a substitute for the primary actor. In other cases, when the type of actor affects the use case, the most suitable actor was selected as primary actor, and the others were left out.
- *Main success scenario and steps.* First, a short description of the use case was constructed. From this, a set of steps was defined, describing the main success scenario. Due to space constraints, this is not shown for all the use cases. In the next section, four use cases are described in detail.
- *Includes relationships.* The "include" relationships between the use cases are based on the steps defined for these use cases. This relationship expresses that a use case contains behavior defined in another use case, as defined in UML 2.0 [12]. When a use case includes another use case with a different primary actor, this typically means that the first actor will ask the second actor to perform the specified use case. For example, in use case 2 (Identify the subversive stakeholder), the project manager will ask the architect to conduct a risk analysis (use case 21). Off course one person can also have multiple roles, and thus perform as the primary actor in both use cases.

Figure 1 presents the characteristics (Primary actor, goal level, and name) of the use case model, which consists of 27 use cases. Note that to enhance the readability, the uses relationship (between the actor and the use case) is not visualized with arrows, but by horizontal alignment. For example, the architecture reviewer acts as a primary actor for use cases 16, 4, 26, and 5. The use cases are vertically divided in the three goal levels: Summary, User-goal and Subfunction. For example, use case 16 is a Summary use-case and use case 4 an User-goal.

IV. Use Case Validation

A use case model like the one presented in section III can not be directly validated in a formal, mathematical sense. Instead, the use cases need to be applied in practice and their effect on the development process should be evaluated. However, before the use cases can be applied, the use cases need further refinement to become usefull. In this validation section, we present these refinements, demonstrate the relevance of the use cases in an industrial setting, and present the improvement these use-cases have made on the development process.

Currently, the Griffin project is involved in conducting case studies at our industrial partners to validate the use cases. In this section we briefly present the Astron Foundation case study. Astron is currently engaged in the development of the LOw Frequency ARray (LOFAR) for radio astronomy [13]. LOFAR pioneers the next generation of radio telescope and will be the most sensitive radio observatory in the world. It uses many inexpensive antennas combined with software, instead of huge parabolic dishes, to observe the sky. This makes LOFAR a software intensive telescope. LOFAR will consists of around 15.000 antenna's distributed over 77 different stations. Each antenna will generate around 2 Gbps of raw data. The challenge for LOFAR is to communicate and process the resulting 30Tbps data stream in real-time for interested scientists.

In the LOFAR system, architectural decisions need to be shared and used over a time span of over 25 years. This is due to the long development time (more then 10 years), and a required operational lifetime of at least 15 years. Astron is judged by external reviewers on the quality of the architecture. The outcome of these reviews influences the funding, and consequently the continuation of the project. Therefore, it is evident that the architecture has to hold high quality standards.

Together with Astron, we identified eight use cases being of primary concern for the LOFAR case study: 5, 7, 10, 12, 15, 17, 19, and 20. This section focuses on assessing the design maturity, which is a major concern for Astron. Assessing the design maturity is a specialization of the earlier identified need for getting an overview of the architecture (see section II, need 9). The following use-cases are relevant with regard to this assessment:

- Asses design maturity (UC 17, see figure 2)
- Identify incompleteness (UC 20, see figure 3)
- Check correctness (UC 5, see figure 4)
- Evaluate consistency (UC 19, see figure 5)

Of these four use cases, use case 17 is the only *Summary level* use case (see figure 1). Use cases 5, 19, and 20 are used by this use case. In the remainder of this section, these use cases are presented in more detail. For each use case, the following is presented: the relevance to Astron, the current practice at Astron, a more elaborate description of the use case, and the envisioned concrete realization within Astron.

The elaborated use case descriptions make use of the concept of **knowledge entity**. All the domain concepts defined within the knowledge grid are assumed as being knowledge entities. For this case study, this includes among others: architectural decisions, rationales, decision topics, alternatives, requirements, specifications, assumptions, rules, constraints, risks, artifacts, and the relationships among them.

A. UC 17: Assess design maturity

Relevance The design maturity is an important part of the quality of the LOFAR architecture. LOFAR is constructed using cutting edge technology to enquire maximum performance. However, due to the long development time, these cutting edge technologies are typically emerging when the initial architecture design is being made. So, the architecture has to be made with components that do not yet exist. It is therefore hard for Astron to make a judgment whether the architecture is sufficiently matured to start actual construction.
Current Practice Within the LOFAR case study, the design maturity is first assessed for the various subsystems by each responsible architect. For each subsystem the main issues with regard to incompleteness, correctness, and consistency are reported. Based on these reports, the opinions of the architects and project management it is decided whether the system is mature enough to be proposed to the external reviewers to proceed to the next project phase.
Use case realization The design maturity use case is presented in figure 2. This use case consists of three other use cases that in turn are used to check the architecture for completeness, correctness, and consistency. These use cases are presented in the remainder of this section.

UC 17: Assess design maturity

Description: This use case verifies whether a system conforming to the architecture can be made or bought. The architect wants to know when the architecture can be considered as finished, complete, and consistent.
Primary actor: Project Manager.
Scope: Knowledge grid
Level: Summary.
Precondition: None.
Postcondition: The knowledge grid provides an overview of the matureness, and reports potential risks.
Main success scenario:

1) Identify incompleteness (UC 20)
2) Check correctness (UC 5)
3) Evaluate consistency (UC 19)
4) The grid generates a report based on the knowledge of the previous steps

Extensions: None

Fig. 2. Use case 17

B. Use Case 20: Identify incompleteness

Relevance Use case 20 determines whether the architecture covers all (essential) requirements. For Astron this is relevant from a management perspective; incompleteness gives pointers to directions where additional effort should be concentrated.
Current practice Astron checks for the completeness of the architecture description by peer review and risk assessment. The peer review is done iteratively; fellow architects give feedback (among others) on the completeness of the architectural descriptions. A risk assessment is performed before every external review. The result of this process, a risk matrix, is used for the next iteration of the architectural description. During the design phase, the architect signifies specific points of incompleteness, typically by putting keywords like ‘tbd’ (to be determined), or ‘tbw’ (to be written) in the architecture documents.

For example, in the central processor, the signals of the antenna’s should be correlated with each other. Therefore, the signals of all the stations should be routed all-to-all. However, the architectural decision on what network topology to use for this task is still incomplete, as some alternatives have been evaluated, but no suitable (cost-effective) solution can be selected so far.
Use case realization The general use case is described in figure 3. For Astron this is realized by the following:

Risks Currently, the relationships between the identified risks (for example in the risk matrix) and the design (in the architectural documentation) are not explicitly clear. The knowledge grid allows the architect to relate risks to particular parts of the design and to architectural decisions. This use case enables the architect to *partially* check the completeness of

Use Case 20: Identify incompleteness

Description: In this use case, the system provides a report about the structure of the architectural decisions.
Primary actor: Architect.
Scope: Knowledge grid
Level: User-goal.
Precondition: The user is known within the knowledge grid.
Postcondition: The knowledge provides an overview of the incomplete knowledge entities.
Main success scenario:

1) The architect selects a part of the architecture.
2) The knowledge grid identifies the knowledge entities in the part.
3) The grid reports about the incompleteness of these knowledge entities.

Extensions: None

Fig. 3. Use case 20

the mitigation of risks, as every risk should be addressed by at least one architectural decision. Whether the risk is actually addressed by the decision, is checked by UC 5, presented in the next subsection.

Requirements As an example of inconsistency indicators every requirement should lead to one or more (mostly non-formal, usually textual) specifications. It can thus be automatically determined which requirements are not covered by any specifications.

Visualization Possibilities of visualization for incompleteness can be visualized by a "to-do" list of open decision topics, or visual indicators in the documentation (e.g. icons, or coloring of text pieces).

C. Use Case 5: Check correctness

Relevance Besides completeness, it is also important to know whether the architectural decisions actually address the requirements. In this sense, correctness is complimentary to completeness. For example, completeness only means that there exist decisions taken with respect to all requirements, while correctness means that these decisions actually lead to a solution that meets these requirements. Astron spends considerable effort in verifying the correctness of the design. Prototypes of major hardware and software components are made and evaluated. Simulations and models are used as well. For example, to deal with the major concern of the enormous amounts data to be processed, Astron has developed an elaborate performance model. This model allows the architects to simulate and validate the correctness of many different concepts for distributed data processing.
Current practice Similar to the check for completeness, peer reviews are used to verify the correctness of the design description. Domain experts verify the design description created by the architect. Based on this feedback, the architect adapts the design description. Doubts about the correctness of parts of the design are typically annotated with the key word 'tbc' (to be confirmed) or placed in a separate open issue sections. If there is any doubt about the *way* in which the correctness is verified, keywords like 'under discussion' are typically used in the architectural documentation.

Use Case 5: Check correctness

Description: In this use case, the knowledge grid supports the user in validating the correctness of the architectural decisions addressing the requirements.
Primary actor: Architect.
Scope: Knowledge grid
Level: Subfunction.
Precondition: The knowledge grid contains incompleteness information of the design.
Postcondition: The knowledge grid contains markings about the correctness; An overview of incorrect knowledge entities is provided.
Main success scenario:

1) The architect selects a set of requirements in the knowledge grid.
2) The knowledge grid provides a list of related architectural decisions and other related knowledge entities (e.g. assumptions, rules, constraints).
3) The architect evaluates related elements and marks the incorrect elements.
4) The architect continues with the next requirement.
5) The knowledge grid provides an overview of incorrect architectural decisions and requirements.

Extensions:
3a. The elements are correct, the architect marks them as such.

Fig. 4. Use case 5

For example, there has been an incorrect assessment of the distributed behavior of the used calibration algorithm. It was expected that each node used 80% local data, and that for the remaining 20% all the data on the other nodes was needed. Based on this assessment the architectural decision was made to use a distributed database grid. However, during performance tests it turned out that for this 20% the data of only one or two other nodes was needed, instead of *all* the other nodes. Consequently, the architectural decision turned out to be wrong, as the architectural decision for a centralized database is a significant better alternative. In retrospect, verification of the architectural decision by the correct domain expert could have prevented this situation from arising in the first place.
Use Case realization The knowledge grid itself cannot determine the correctness of the architectural decisions without in-depth semantic knowledge of the underlying architectural models. Therefore, this use case makes provision for assisting the architect in determining the correctness of the design, rather than that the knowledge grid determines the correctness itself.

Requirements For each requirement or risk, the architect needs to find out whether the involved architectural decisions correctly address the requirement or risk. This use case describes how this process can be supported.

Visualize The visualization of the incompleteness can subsequently be used to visualize incorrect elements. However, since the checking of correctness is mostly manual job for the architect, the results may vary when different people are checking the correctness. Integration of this information is then needed.

D. Use Case 19: Evaluate consistency

Relevance This use case is concerned with the consistency between the architectural decisions themselves. As the LOFAR project consists of many components that are developed in parallel, detecting contradictions is important, as this provides an early warning for mistakes in the overall design. Inconsistencies make the design of the system harder to understand and create problems during the realization of the system.

Current practice Checking for inconsistencies in textual descriptions is largely a manual job. The part of the design that is modeled (e.g. in the performance models) can automatically be checked for inconsistencies. However, they only cover a very small part of the overall design, and therefore a small part of the architectural decisions. Most of the inconsistencies are found by inspection, either by the architect or reviewer.

For example, there has been an inconsistency in LOFAR between the protocol used by the central processor (the correlator of the radio signals) and the stations (the locations where the antenna's are residing). Although large efforts have been put in a consistent definition of the data packet header, versioning etc., the used definition of how to interpret the subband data turned out to be inconsistent. For the station a subband was defined starting with the lowest frequency leading to the highest frequency of the subband, while for the central processor it was defined the other way around.

Use case realization The architect is supported with relevant context information in the decision making process. For Astron, this will include the visualization of relevant requirements, and closely related architectural decisions and specifications. Techniques similar to the work of [14] could be used for this. Furthermore, once an inconsistency is detected, the architect is supported with a visualization of the relevant architectural decisions. This allows the architect not only to confirm an inconsistency, but also to detect its cause and consequently resolve it.

Use Case 19: Evaluate consistency

Description: In this use case, the knowledge grid supports the user in detecting inconsistencies in the architecture design.
Primary actor: Architect.
Scope: Knowledge grid.
Level: User-goal.
Precondition: The user is known within the knowledge grid.
Postcondition: The knowledge grid contains markings about the consistency; An overview of inconsistent knowledge entities is provided.
Main success scenario:

1) The architect selects a subset of architectural knowledge in the grid.
2) Architect selects a specific knowledge entity or a part of the design, and asks the knowledge grid for consistency assistance.
3) The knowledge grid provides a list of related (and potentially inconsistent) knowledge entities.
4) The architect marks the inconsistent knowledge entities.
5) The architect repeats steps 3 and 4 for the remaining knowledge entities.
6) The knowledge grid provides an overview of inconsistent knowledge.

Extensions:
4a. The knowledge entities are consistent, the architect marks them as such.

Fig. 5. Use case 19

V. Related Work

Software architecture design methods [15], [16] focus on describing how sound architectural decisions can be made. Architecture assessment methods, like ATAM [15], assess the quality attributes of a software architecture. The use cases presented in this paper describe some assessment scenarios that could be reused from these design and assessment methods.

Software documentation approaches [17], [18] provide guidelines for the documentation of software architectures. However, these approaches do not explicitly capture the way to take architectural decisions and the rationale behind those decisions. The presented use cases describe how stakeholders would like work with this knowledge.

Architectural Description Languages (ADLs) [19] do not capture the decisions making process in software architecting either. An exception is formed by the architectural change management tool Mae [20], which tracks changes of elements in an architectural model using a revision management system. However, this approach lacks the notion of architectural decisions and does not capture the considered alternatives or rationales, something the knowledge grid does.

Architectural styles and patterns [21], [22] describe common (collections of) architectural decisions, with known benefits and drawbacks. Tactics [15] are similar, as they provide clues and hints about what kind of techniques can help in certain situations. Use case 22 (Detect patterns of architectural decision dependencies), can be used to find these kinds of decisions.

Currently, there is more attention in the software architecture community for the decisions behind the architectural model. Tyree and Akerman [3] provide a first approach on documenting design decisions for software architectures. Concepts and guidelines for explicit representations of architectural decisions can be found in the work of Babar et al. [23] and

our own work [5], [6]. Closely related to this is the work of Lago and van Vliet [24]. They model assumptions on which architectural decisions are often based, but not the architectural decisions themselves. Kruchten et al. [2], stress the importance of architectural decisions, and show classifications of architectural decisions and the relationship between them. They define some rough outlines for the use cases for describing how to use architectural knowledge. Furthermore, they provide an ontology based visualization of the knowledge in the grid. We emphasize more on the explicit modeling of the use cases and are validating a set of extended use cases in the context of a case study.

Integration of rationale and design is done in the field of design rationale. SEURAT [25] maintains rationales in a RationaleExplorer, which is loosely coupled to the source code. These rationales have been transferred to the design tool, to let the rationales of the architecture and implementation level be maintained correctly. DRPG [26] couples rationale of well-known design patterns with elements in a Java implementation. Just like SEURAT, DRPG also depends on the fact that the rationale of the design patterns is added to the system in advance. The importance of having support for design rationales was emphasized by the survey conducted by Tang et al. [4]. The results emphasized the current lack of good tool support for managing design rationales. The use cases presented in this paper are an excellent start for requirements for such tools.

From the knowledge management perspective, a web based tool for managing architectural knowledge is presented in [23]. This approach uses tasks to describe the use of architectural knowledge. These tasks are much more abstract then the use cases defined in this paper (e.g. architectural knowledge use, architectural knowledge distribution).

Finally, another relevant approach is the investigation of the traceability from the architecture to the requirements [27]. Wang uses Concern Traceability maps to reengineer the relationships between the requirements, and to identify the root causes. The results from such systems could be valuable input for defining the relationships between knowledge entities, as used in our validation.

VI. Conclusions and Future Work

In order to upgrade the status of architectural decisions, we must first understand how they can be shared and used by a software development organization. For this purpose, we have proposed a use case model that came out of industrial needs and aims to fill specific gaps and in particular to alleviate the dissipation of architectural decisions. This use case model is considered as the black-box view of a knowledge grid type of system that is envisioned to enrich the architecting process with architectural decisions.

A reasonable question to reflect upon is: how exactly was the software architecture quality enhanced by the use case model proposed in this paper? Although pinpointing what exactly constitutes the quality of software architecture per se is a difficult issue, we can identify five arguments in this case:

- **Less expensive system evolution**. As the systems need to change in order to deal with new requirements, new architectural decisions need to be taken. Adding, removing and modifying architectural decisions can be based on the documentation of existing architectural decisions that reflect the original intent of the architects. Moreover, architects may be less tempted to violate or override existing decisions, and they cannot neglect to remove them. In other words the architectural decisions are enforced during evolution and the problem of *architectural erosion* [28] is reduced.
- **Enhanced stakeholder communication**. The stakeholders come from different backgrounds and have different concerns that the architecture document must address. Architectural decisions may serve the role of explaining the rationale behind the architecture to all stakeholders. Furthermore, the explicit documentation of architectural decisions makes it more effective to share them among the stakeholders, and subsequently perform tradeoffs, resolve conflicts, and set common goals.
- **Improved intrinsic characteristics of the architecture**. These concern attributes of the architecture, such as conceptual integrity, correctness, completeness and buildability [15]. Architectural decisions can support the development team to upgrade such attributes, because they give more complete knowledge, they provide a clearer and bigger picture. In other words, architectural decisions provide much richer input to the formal (or less formal) methods that will be used to evaluate these attributes.
- **Extended architectural reusability**. Reuse of architectural artifacts, such as components and connectors, can be more effectively performed when the architectural decisions are explicitly documented in the architecture. To reuse architectural artifacts, we need to know why they were chosen, what their alternatives were, and what benefits and liabilities they bring about. Such kind of reusability prevents the architects from re-making past mistakes or making new mistakes. Finally architectural decisions per se, can and should be reused, probably after slight modifications.
- **Extended traceability between requirements and architectural models**. Architectural decisions realize requirements (or stakeholders' concerns) on the one hand, and result in architectural models on the other hand. Therefore, architectural decisions are the missing link between requirements and architectural models and provide a two-way traceability between them [6]. The architect and other stakeholders can thus reason which requirements are satisfied by a specific part of the system, and vice-versa, which part of the system realizes specific requirements.

We are currently trying to validate the use case model in four industrial case studies to better understand the pragmatic industrial needs and make the use case model as relevant and

effective as possible. After this validation, we plan to perform a second iteration of validation interviews with the original interviewees from the first iteration, as well as more stakeholders with different roles, in order to fully cover the most significant roles. Furthermore external architects will also be asked to validate the use case model. In the meantime we have already attempted to implement parts of the knowledge grid in the form of tool support, which is used in the aforementioned case study of the Astron Foundation.

ACKNOWLEDGEMENTS

This research has partially been 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] J. Bosch, "Software architecture: The next step," in *Software Architecture, First European Workshop (EWSA)*, ser. LNCS, vol. 3047. Springer, May 2004, pp. 194–199.
[2] P. Kruchten, P. Lago, H. van Vliet, and T. Wolf, "Building up and exploiting architectural knowledge," in *WICSA 5*, November 2005.
[3] J. Tyree and A. Akerman, "Architecture decisions: Demystifying architecure," *IEEE Software*, vol. 22, no. 2, pp. 19–27, 2005.
[4] A. Tang, M. A. Babar, I. Gorton, and J. Han, "A survey of the use and documentation of architecture design rationale," in *Proceedings of WICSA 5*, November 2005.
[5] A. G. J. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proceedings of WICSA 5*, November 2005, pp. 109–119.
[6] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis, and J. Bosch, "Design decisions: The bridge between rationale and architecture," in *Rationale Management in Software Engineering*, A. H. D. et al., Ed. Springer-Verlag, march 2006, ch. 16, pp. 329–348.
[7] P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," in *Proceedings of the Second International Conference on the Quality if Software Architectures (QoSA 2006)*, June 2006.
[8] H. Zhuge, *The Knowledge Grid*. World Scientific Publishing Company, 2004.
[9] Griffin project website, http://griffin.cs.vu.nl.
[10] R. Farenhorst, R. C. de Boer, R. Deckers, P. Lago, and H. van Vliet, "What's in a domain model for sharing architectural knowledge?" in *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE2006)*, July 2006.
[11] A. Cockburn, *Writing Effective Use Cases*. Addison Wesley, 2001.
[12] The Unified Modeling Language (UML) website, http://www.uml.org/.
[13] Lofar project website, http://www.lofar.org/.
[14] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Improving after-the-fact tracing and mapping: Supporting software quality predictions," *IEEE Software*, vol. 22, no. 6, pp. 30–37, November/December 2005.
[15] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice 2nd ed.* Addison Wesley, 2003.
[16] J. Bosch, *Design & Use of Software Architectures, Adopting and evolving a product-line approach*. ACM Press/Addison Wesley, 2000.
[17] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures, Views and Beyond*. Addison Wesley, 2002.
[18] C. Hofmeister, R. Nord, and D. Soni, *Applied software architecture*. Addison Wesley, 2000.
[19] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.
[20] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic, "Taming architectural evolution," in *Proceedings of the 8th European software engineering conference*. ACM Press, 2001, pp. 1–10.
[21] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., 1996.
[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A system of patterns*. John Wiley & Sons, Inc., 1996.
[23] I. G. Muhammad Ali Babar and R. Jeffery, "Toward a framework for capturing and using architecture design knowledge," University of New South Wales, Australia ans National ICT Australia Ltd., Tech. Rep. UNSW-CSE-TR-0513, june 2005.
[24] P. Lago and H. van Vliet, "Explicit assumptions enrich architectural models," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005, pp. 206–214.
[25] J. E. Burge and D. C. Brown, "An integrated approach for software design checking using design rationale," in *1st International Conference on Design Computing and Cognition (DCC '04)*, July 2004, pp. 557–576.
[26] E. L. A. Baniassad, G. C. Murphy, and C. Schwanninger, "Design pattern rationale graphs: Linking design to source," in *Proceedings of the 25th ICSE*, May 2003, pp. 352–362.
[27] Z. Wang, K. Sherdil, and N. H. Madhavji, "ACCA: An architecture-centric concern analysis method," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, November 2005.
[28] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

APPENDIX
GRIFFIN QUESTIONNAIRE

This appendix contains the questionnaire that was used during the interviews with the stakeholders at the industrial partners. The questionnaire was sent to most interviewees in advance and used by the Griffin research team to see whether all relevant subjects have been discussed during the interview.

Introduction of yourself

1) Can you describe your role and responsibilities within the organization?
2) Can you give an estimate on what percentage of your time is spent on activities related to architecture? Examples include capturing architectural knowledge, communicating architectural knowledge to stakeholders, et cetera.
3) With what kind of stakeholders in the architecting process do you interact most?

Architecture

1) For the sake of clarity: what is your definition of "(software) architecture"? Does this definition differ from the generally accepted definition within your organization?
2) Can you describe the software design process, and the place the architecture takes in it?
3) Are architectures kept up-to-date during evolution? What techniques are used in keeping the architectures up-to-date?
4) For what stakeholders are architecture documents created? What are (generally spoken) the most important stakeholders?
5) Are tools, methods, templates, or architectural description languages (ADLs) used in constructing an architecture?
6) Are you satisfied about the way these tools, methods, templates, or ADLs are being utilized during the architecture construction process? Can you mention any improvement points?

Architectural knowledge

1) What architectural decisions are documented, and how?
2) Can you quantify the impact of Architectural Knowledge that is lost / not present / too implicit? Can you give examples?
3) Could you provide a top-3 list of burdens in modelling architectures? What are your ideas on this?

Your architectures of today and in the past

1) What are the most important quality characteristics of your architecture (or architectures)?
2) What kind of solutions do you provide in your design? Do you reuse certain solutions (e.g. architectural patterns) in your architectures?
3) (If possible to mention commonalities): With what kinds of design aspects do you deal explicitly in your architectures? Examples of design aspects include interfaces, error handling, execution architecture, data consistency, and robustness.
4) From what sources do you obtain information for these design aspects?
5) Is there a topic on which you foresee a big change in the use of architectures in the future?

A. Architecting in daily practice

1) How is the availability of architectural information planned, managed, and reviewed?
2) What will be (in your opinion) a big change in the architect's job in the future?
3) Looking back on the last few years, what would you reckon as a significant step forward in architecting support?
4) How would you prepare for this?
5) How is this change planned?

Using Generator Feature Diagrams to Enhance Performance Prediction of Software Component Adaptation - A Case Study

Niels Streekmann
OFFIS
Escherweg 2, D-26121 Oldenburg, Germany
Email: niels.streekmann@offis.de

Steffen Becker
Institute for Programm Structures and Data Organization
Faculty of Informatics, University of Karlsruhe (TH)
Am Fasanengarten 5, D-76131 Karlsruhe, Germany
Email: sbecker@ipd.uka.de

***Abstract*— In order to put component based software engineering into practice we have to consider the effect of software component adaptation. Adaptation is used in existing systems to bridge interoperability problems between bound interfaces, i.e., to integrate existing legacy systems into new software architectures. In CBSE, one of the aims is to predict the quality properties of the assembled system from its basic parts in order to avoid insufficient quality attributes of the final software system. Adaptation is a special case of such a composition and should be treated consequently in a special way. However, today this is not the case. In our approach, we integrate explicit knowledge on software component adapters to increase the accuracy of performance predictions. Hence, this work examines the use of adapter generators which simultaneously produce prediction models. An experiment comparing predicted performance figures with measured values of the generated adapter is presented. The results show the increase in accuracy when using feature dependent prediction models.**

I. Introduction

Software Component Adaptation is a crucial task when building component based software systems. When developing components there are always two design forces involved. On the one hand, components must be reusable in a variety of different deployment platforms. On the other hand, components must provide specialized functionality to make them applicable in specialized contexts. Hence, a trade-off has to be made to balance these forces. As a matter of fact, there are some cases where the compromise leads to limited applicability of the components. As a result, adaptation has to be performed at assembly time to compensate for the trade-off.

Agreeing on the need of adaptation as a task of the system assembler, there are two issues that have to be tackled for adaptation to become a well planned engineering activity: First, adaptation has to be done in a structured and guided way to reduce the amount of hacking unstructured glue code. Second, we need prediction methods for the impact of the adaptation on QoS.

Tasks needed for the first step include the development of appropriate adaptation methods. Such methods have to *detect mismatches* in a software architecture specification. The detection should be based solely on the available specification of the component. In the specification the provided and required interfaces of the respective components play a central role. Specifications on different interface abstraction levels allow the detection of different mismatch classes, e.g., the availability of a protocol specification allows the detection of protocol mismatches and a QoS specification allows the detecting of mismatching QoS properties. Historical and more up to date classifications of interface abstractions can be found in [1].

The method used here (which has been presented initially in [2]) is based on the use of generative or model driven development (MDD) approaches utilizing the specification and detection algorithms to generate the appropriate adapters. This has been demonstrated in literature for certain problem classes before [3], [4]. Note, that in most cases the adapter generator is semi-automatic hence requiring additional input by its users. We base our generated code on well known design patterns as they are established solutions to reccurring problems. If we take functional and extra-functional adaptation into account, there is a huge variety of patterns that can be used to bridge component mismatches (Note, that we consider a mismatch in the required and offered QoS explicitly as an adaptable interoperability problem in this paper). Nevertheless, adaptation has an impact on QoS [5]. However, the additional knowledge on the adapter can be used during the prediction of the impact on the extra-functional properties.

The contribution of this paper is to apply the method presented in [2] by implementing a generator which is capable of generating code according to the cache and replicator design patterns to overcome performance mismatches. During the implementation we gained additional experience with the application of the method which lead to a refinement of the method. The refined method is part of section II. In our implementation, an arbitrary selection of patterns and prediction models - specialized for their respective generated code - are utilized to predict the impact of the software component adapter. Afterwards, a case study is presented which is used to validate the method. The study presented here has its focus on the impact of the generator configuration. Hence, other known influences, like the underlying hardware,

are kept constant. The results demonstrate that the method can be used to increase the accuracy of todays used early performance prediction methods.

This paper is structured as follows. In section II, our approach is explained in more detail and the advantages of the approach are discussed. Succeeding that section, we give an example illustrating how the proposed process can be put into practice. This is followed by a case study in section V, where the applicability of the approach and its effects on prediction are discussed. After briefly highlighting some related work in section VI, the paper concludes and highlights some future work.

II. Generating Adapter Components based on Patterns

Generators are well known tools to simplify transformations of solutions or to reuse code fragments with certain variable parts which can even be derived by the generator [6]. A generator uses a so-called feature diagram to structure the input needed to configure the generation process. In the method outlined (see also [2]), the generator uses two sources of input: The specification of the interfaces involved in the adaptation and additional information queried from the component assembler. The information can be used in the generated adapter as well as in the prediction model.

A model is required to predict the extra-functional properties of the composition between a component and its adapter in advance. As adaptation has an impact on the extra-functional properties of the component, its influence has to be considered when predicting extra-functional properties of the system. One of the reasons for the explicit inclusion of adapters in prediction models for software architectures is the aim to improve their accuracy. A detailed analysis of the impact of component adaptation on QoS is presented in [5].

Figure 1 summarizes the presented method by showing the adapter generator and the simultaneously generated prediction model.

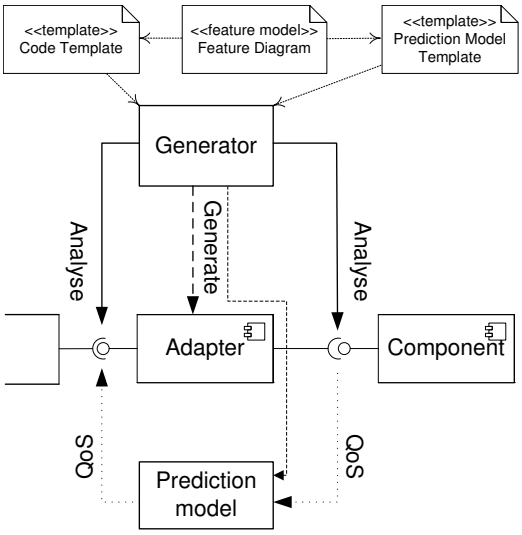


Fig. 1. Generation of Adapter and corresponding Prediction Model

The workflow for implementing the method is as follows and can be seen in the figure. First, the QoS specifications of the component providing a service (the right one in the figure) is checked against the QoS requirements of the component requiring a service (the left one in the figure). This is done by comparing their respective QML-contracts associated to the provided respective required interfaces. In the figure this is indicated by the solid "analyse" arrows.

If a mismatch is found the generator offers to generate an appropriate adapter by querying its template database. The template database contains adapter templates based on patterns. For each template there is a rough characterisation which QoS attributes are affected by using the pattern as adapter, e.g., for the cache pattern there is the information that it can be used to increase performance. All patterns found in so doing are presented to the developer who selects one of them to be applied.

For the selected pattern the generator looks up the necessary feature diagram. The feature diagram contains the variation points available in the selected template, i.e., the size of the cache for the cache template. The generator queries the user to specify the features needed. Afterwards, it does the actual generation. This is implemented as simple text generator using a template generator engine. This engine simply substitutes special marked parts of the template with the respective feature information. The result is a complete software artifact.

Two artifacts are generated. On the one hand, source code for the adapter component. This source is based on the pattern implemented in the respective template which the developer has selected, i.e., source code implementing a software cache. On the other hand, input files for any supported QoS prediction method are generated. Note, that the generator uses the same features when generating these artifacts because the features can influence both. Take the cache size for example. It is used in the source code to allocate an appropriate amount of memory and in the prediction model to estimate the cache hit and cache miss ratio. In so doing, it is ensured that the source and the respective prediction model input is in sync. Hence, we expect an increase in the accuracy of the predictions.

Note, that it is part of the decision of the developer of the generator and its template database which patterns are supported. Additionally, it is also his decision which prediction models can be used as he has to give templates for the prediction model's input. Hence, the approach presented here can be used with arbitrary prediction methods. The selection we made in section V can be changed anytime when implementing new templates. The important aspect is that the input of the prediction methods has to be derived from the features of the chosen pattern template.

For component based systems the use of parametric prediction models is needed. Parametric models take in our case the QoS of the adapted component as input parameter. In so doing, those models enable compositional reasoning taking advantage of the component based architecture of the system. Compositionality is an important property if multiple adaptations should be allowed, i.e., in case of two adaptations

the output of the analysis of the first adapter serves as input to the second.

III. Example

To illustrate the idea presented in the previous sections, consider the following example. We have a component which encapsulates a certain kind of information. The information is not being changed frequently, but its retrieval consumes a significant amount of time. Another component is going to access this component but needs a lower response time for the information retrieval service. The problem can be detected, for example, using a Quality of Service Modeling Language (QML) [7] specification of the respective interfaces as depicted in figure 2. QML is highly customizable - the possible specifications include mean values, standard deviation or a set of quantiles characterizing the distribution of any self-defined quality metric. In the example, we define a metric *delay* indicating the duration of the service call.

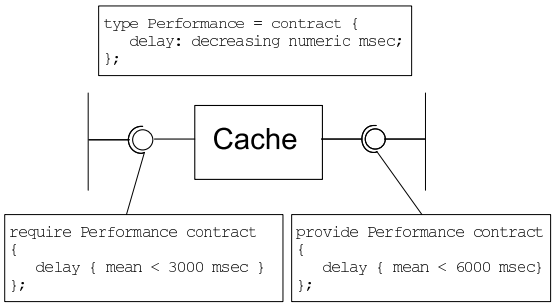


Fig. 2. A Cache to solve a Quality of Service Mismatch

In figure 2, there is also a possible solution to the presented problem: The application of a cache can fix the detected mismatch. The cache pattern is well-known in literature [8, p. 83] and has been applied for a long time in hard- and software development. In our case, we need an generator template which is able to generate the cache. When deriving such a template, one feature we can identify is the type of the information objects which need to be cached from the interface specifications.

More features can be found. In the given example, a generator can query some information taken from the pattern description and implemented in the feature list of the generator template. Referring to the description in [8] we have to

- Select resources: The resource being retrieved
- Decide on an eviction strategy: Here we can choose between well-known types like least recently used (LRU), first in - first out (FIFO), and so on.
- Ensure consistency: We need a consistency manager whose task is to invalidate cache entries as soon as the master copy is changed.
- Determine cache size: How much memory the cache is going to use. Most likely this is specified in number of cacheable resource units.

Every single decision made here can be included additionally into the prediction model of the QoS impact. In the specific example, it is especially important as a cache component is quite difficult to model in QoS todays prediction models. Caches are stateful and hence introduce the problem of stateful components [9]. This problem results from the fact that the operational profile has an impact on QoS. Consider a component storing an array of records. To search in an array containing few elements is faster than searching an array with a lot of records. Thus, a prediction method has to take the state of the component into account. The resulting complexity can be high. Hence, none of the todays applied performance prediction methods includes the runtime state of the components into the prediction. Nevertheless, we are able to predict the QoS impact based on the information available as it can be analyzed in special cases or simulated. In our example, a simulation model can utilize the eviction, locking and consistency strategy from the adapter generator's input to simulate the behaviour of the cache. During the execution of the model calls on the adapter are simulated and the response times are determined depending on the configuration of the generator.

With the resulting prediction model the impact of the adaptation can be predicted and, hence, it is possible to reason on the composition of the adapter and the adaptee.

IV. A Case Study

The example in the preceding section has been examined in a case study. The performance of generated adapters based on different feature configurations has been measured. These measurements are compared to the predictions of two applied prediction methods. Thereby the input models of the methods were generated by the same generator employing the same feature configuration. To be able to conduct the case study in a practicable period of time a set of restrictions had to be worked out to determine the scope of the study. These restrictions are listed in the following:

- The focus of the case study is the quality-based interface model as described in [10]. This is further narrowed down to performance being the only considered quality attribute.
- Not all features and variants of the cache pattern have been examined. The adopted patterns are shown in figure 3.
- The hardware influence is not considered. This results from the large impact of the influence of different hardware on the QoS properties of the component being executed on a specific hardware. To meet this restriction, all measurements were executed on the same system.
- The influence of the adapter on QoS properties other than performance has been disregarded.
- It is assumed that the adapted component has constant QoS properties. This restriction has to be made to gain a solid basis for analysis and prediction. In reality however changes to the adapted component or components it uses will affect the QoS of the component.

In the case study adapters implementing a cache were generated by a pattern-based generator application which is

described in [11]. This application is able to detect mismatching QoS properties. The basis for this detection are the QML specifications of the provided and required interfaces of the respective components. The application suggests a set of design patterns to the system assembler which are useful to solve the found mismatches. To make this possible, the design patterns have to be described in an automatically processable way including ratings respecting the affected quality properties.

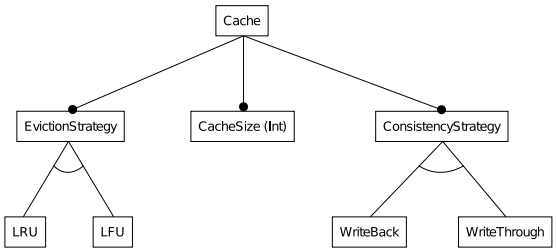


Fig. 3. Basic feature diagram of the cache pattern

The choice and configuration of an adequate pattern are delegated to the system assembler since the systems context has to be considered. The configuration of the patterns is based on feature diagrams as described in section II. Feature diagrams were introduced in [6]. Figure 3 shows the used feature diagram of the cache pattern. The diagram shows three basic features of a cache and a selection of subfeatures. The diagram is not intended to be complete, but includes a selection of mandatory features that were examined in the case study. These are the size of the cache and strategies for the eviction of resources from the cache and the preservation of consistency to the adapted component. Besides the feature diagram of the cache other feature diagrams have been used for the configuration of the generator. These diagrams contain the implementation-dependent features of the adapter itself and features of the adapted component. The latter include e.g. the retrieved resources mentioned in section III.

```
type Performance = contract {
     delay: increasing numeric msec;
}

testServiceProfile for IService = profile {
     from Set require Performance contract {
          delay{
               percentile 30 < 20;
               percentile 70 < 40;
               percentile 100 < 80;
          };
     };
     from Get require Performance contract {
          delay{
               percentile 30 < 10;
               percentile 70 < 20;
               percentile 100 < 40;
          };
     };
}
```

Fig. 4. QML specification of the adapted data service

In the case study the adapted component provides a simple data service using 100 indexed integer values as cacheable resources. The QML specification shown in figure 4 models the QoS of the provided interface of the adapted service. In this case the QoS is reduced to the performance of the service defined as its response time. The response time is defined as a set of quantiles. The response time is the only QoS property that has been examined during the case study. The influence of the adapter on other QoS properties has not been observed to preserve simplicity.

The QoS of a component C can be described by a function

$$qos(im(fm), op)$$

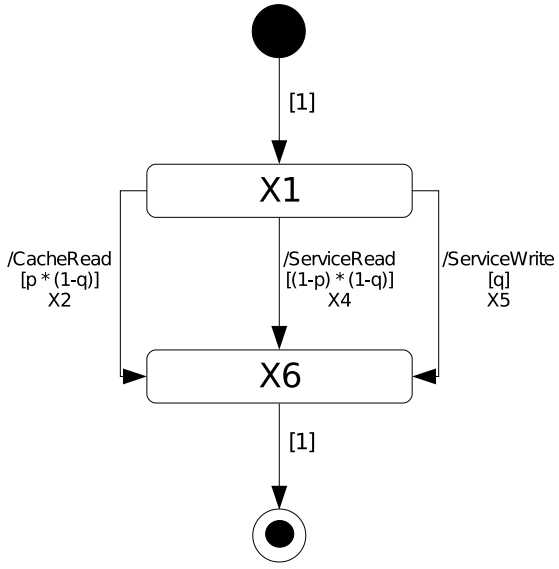
where $im(fm)$ means the implementation (im) of C depending on the feature model (fm) of C and op stands for the operational profile of C. I.e. that the QoS of a component depends on its feature-based generated implementation and on the context the component is used in. Further contextual influences as the usage of external services or the allocation environment were fixed to reduce the complexity of the case study. In our example the only external service used by the adapter is the adapted service. The performance of which is fixed to the specification given in figure 4.

From the simplified viewpoint of the case study the adapters QoS depends on its (generated) implementation and the operational profile defined in the measurement tool. For the case study a simple operational profile that is not tailored to fit into a specific application domain has been used. There were only sequential calls that queried random resources. To measure the influence of the cache locality repeated calls to the same resource were executed. Cache locality means the temporal closeness of accesses to the cache with the same query.

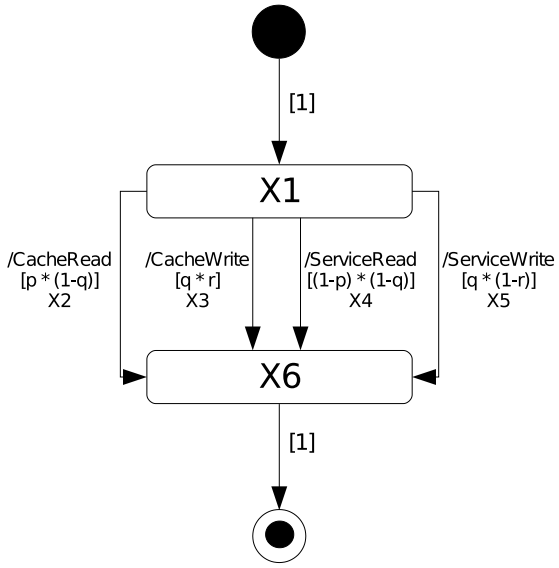
To get a reference measure for the interpretation of the results of the examined prediction models the response time of the generated adapters has been measured using a simple measurement tool. The tool simulates a customisable number of clients as threads. It generates calls in uniformly distributed intervals and with uniformly distributed index values. Inputs of the tool are the number of simulated clients, the number of calls for these clients, the probability of a writing call and the probability of a repeated call to the same resource. The adapters response times are measured using a standard timer.

To show the influences of the feature-based pattern configuration on the prediction of the response time of the adapted component, two different prediction methods were employed and compared to each other. As an example of an analytical approach the Palladio performance prediction method which is introduced in [12] has been chosen. The Palladio method computes the response time of a service based on a finite state machine model using discrete Fourier transform. As a second prediction method a self-implemented simulation model has been used.

Besides the features of the adapter further features derived from the operational profile of the adapter and the specification of the adapted component are needed to predict the response time of the adapter. The selection of certain features changes the input parameters of the parametric prediction model. Features can thereby have different influences on the prediction



(a) Write-Through



(b) Write-Back

Fig. 5. Service Effect Automata modelling a cache with different consistency strategies

model. Some change the structure of the model, others have a certain influence on variables in the model, e.g. the variable *cache-hit-probability* depends on the cache size, the eviction strategy and influences on the operational profile.

The influence of the feature-based configuration on a prediction model can be seen in the Service Effect Automata in figure 5. Service Effect Automata are used to model the generated adapters. They are the input model of the Palladio Performance Prediction Tool, that has been used as an example for analytic prediction in the case study. The analysis of Service Effect Automata has been introduced in [12]. A Service Effect Automaton is a finite state machine that models a certain service implemented by the examined component highlighting the external service calls needed to provide the service. The internal execution of the components is modelled with states while external calls are modelled as transitions. States and transitions are annotated with the distribution of their execution times (X1-X6). Furthermore transitions are annotated with the probability of their execution. This probability and the distribution of execution times of the transitions influence the analytically computed response time of the whole system. The execution time distributions are not depicted in figure 5, but serve as an input of the prediction tool in the corresponding XML-representation of the automata.

The shown automata depict the cache adapter with two different consistency strategies. Both automata have in common the execution of the adapter-specific methods (X1 and X6) and the execution of reading accesses through the cache (X2) or the adapted component (X4). The variable *p* represents the cache-hit probability. The writing accesses are handled differently in both models depending on the selected consistency strategy. The write-through strategy is modelled by a single transition (*/ServiceWrite*) that represents writing to the adapted component since every writing call affects the cache and the adapted component. The write-back strategy on the other side is modelled by two transitions. */ServiceWrite* is also used here, but supplemented by */CacheWrite* and the probability *r* that describes how often a resource is written in the cache only before it is written back to adapted component. In both cases *q* is the probability of a writing call.

V. Experimental Results

A. Comparison of the Measured Performance to the Prediction

To validate the prediction models and to show that the influences of the features on the adapter and the prediction models lead to more meaningful predictions, the measured performance of the generated adapter and the results of the prediction models have been compared in the afore described case study. Therefore adapters with differing configurations have been generated using the generator application mentioned in section IV. These adapters were installed using a test service that implements the QML specification shown in figure 4 and measured using the measurement tool described in section IV. Parallel to that the build systems were specified according to the input models of the employed prediction methods whereupon these methods were executed. Thereby the model used for the simulation has been directly implemented as part of the self-implemented simulation application and the input models of the Palladio Prediction Method consist of the XML representation of the automata in figure 5. In the following the results of the measurements and the prediction methods are compared.

Figures 6 and 7 show the results of the measurement of the generated adapter for the two examined consistency strategies and different cache sizes. The dimensions of the diagrams are the response time that has been measured and the writing call probability which is a parameter of the measurement tool described in section IV. Besides this the straight lines represent the results for different feature configurations. The continuous

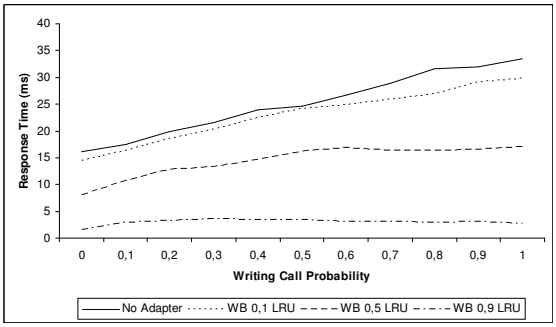


Fig. 6. Measurement results with Write-Back strategy

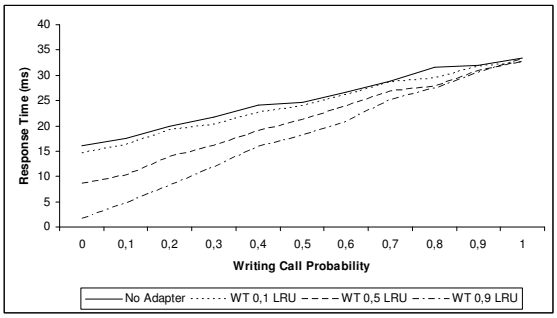


Fig. 7. Measurement results with Write-Through strategy

line in both figures represents a reference measure of the adapted service without any adapter and therefore equals the specification in figure 4 except for errors of measurement. The other lines represent the measurement results for the adapter with different consistency strategies and cache-sizes. Thereby *WB* stands for write-back, *WT* for write-through and 0.1, 0.5 and 0.9 stand for the cache size relative to the total number of resources. E.g. 0.1 means the cache is able to hold 10% of the resources. The eviction strategy in all diagrams shown in this section is *Least Recently Used*. The results for the *Least Frequently Used* strategy are not shown because they do not differ significantly. The complete results are described in [11]. It can be seen in the diagrams that the response time decreases significantly for growing cache sizes and predominant reading accesses for both strategies. They also show that this tendency lowers for increasing writing calls using the write-through strategy.

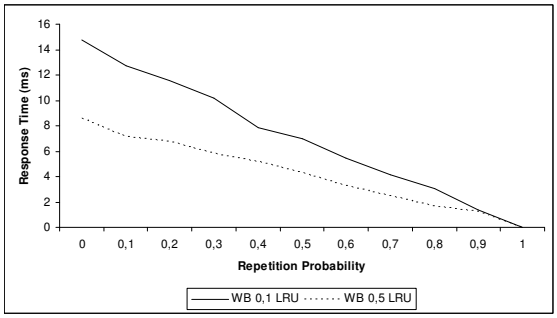
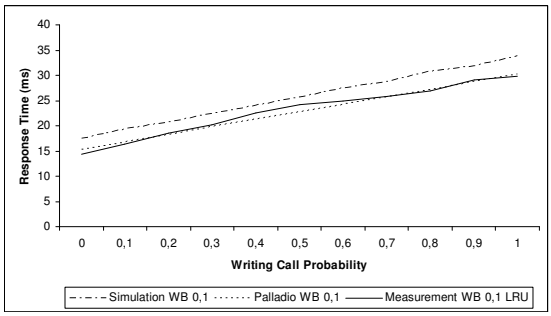
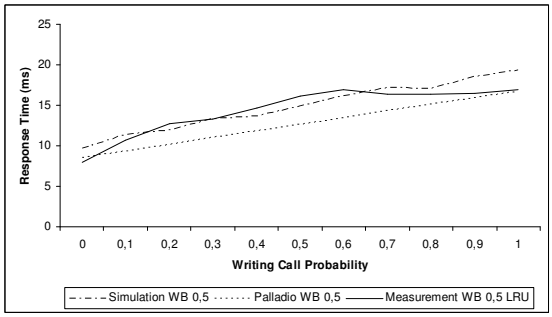


Fig. 8. Measurement results for repeated calls of the same resource

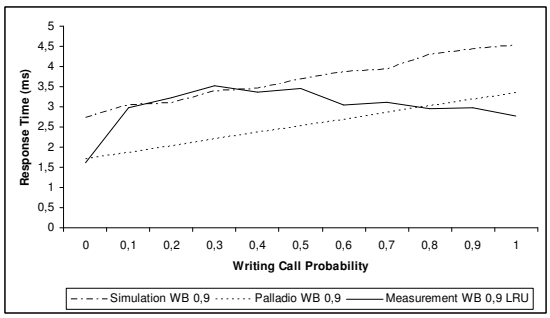
Another important detail in the shown cache example is the influence of locality effects on the prediction of the response time of the adapter. Figure 8 shows the influence of repeated calls of the same resource on the response time. It can be seen that the response time decreases with increasing repetition probability. I.e. that the response time will decrease the higher the influence of the cache locality is in the operational profile.



(a) Cache size: 10%



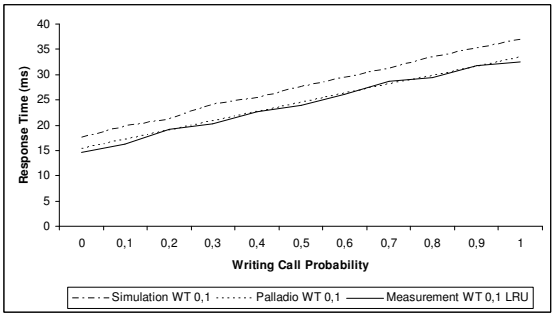
(b) Cache size: 50%



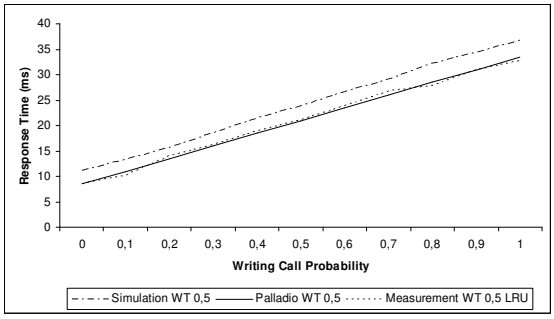
(c) Cache size: 90%

Fig. 9. Prediction results for write-back strategy

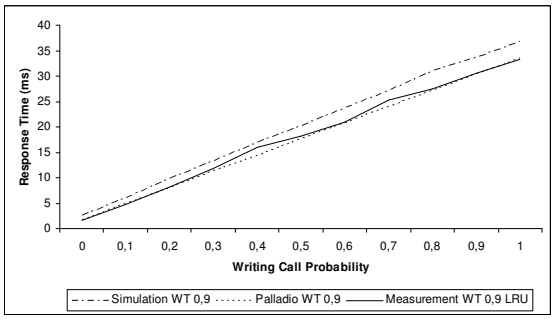
Each measurement configuration was also used to generate analytic and simulative prediction models. The results that were achieved by executing these models where compared to the measurement results. This can be seen in figure 9 for the write-back strategy and in figure 10 for the write-through strategy. The figures indicate that both prediction models and the measurements present a similar distribution of the mean response time, where the predicted response time of the simulation are always slightly higher as the predicted response time of the Palladio model. The reason for this could be that some constant factor has not been regarded in the implementation of the simulation.



(a) Cache size: 10%



(b) Cache size: 50%



(c) Cache size: 90%

Fig. 10. Prediction results for write-through strategy

Most results show a linear distribution of the response times. An exception to this scheme is the measurement for the write-back strategy and high cache sizes. It can be seen that the measured mean response times for medium writing probability values deviates from the linear predictions. A maximum deviation of up to 38% could be observed for both prediction models. The reason for this could not be found in the present case study. On the other side most of the predicted results deviate only up to 10% from the measured mean response times.

B. Interpretation of the Results

The results of the case study show that feature-based prediction is a step to component adaptation becoming a well planned engineering activity. It provides the opportunity to include the details of the implementation of an adapter into the prediction in an engineering way using well-known design patterns.

The results of the case study lead to the conclusion that configuration on the basis of a feature model of the adapter is able to change the implementation of the adapter and the prediction models in a number of ways. This could also be detected in the measurement and prediction results. The case study further illustrated that the operational profile has a decisive influence on the behaviour of the adapter. One example is the influence of the cache locality on the response time of the adapter. Another example is described in [11]. There it could be shown that the concurrent usage of a component that works sequentially leads to exponentially growing response times.

As it can be seen in the measurement of repeated calls on the same resource, the data flow has a decisive influence on the response time of a cache adapter and should be taken into account in the prediction model. Another reason to consider the data flow in the prediction models is the statefulness of a cache mentioned in section III. On the other side the consideration of the data flow causes the disadvantage of a complex model which is hard to solve analytically and will take a long time to be simulated. The importance of the operational profile for performance predictions and the problems that result from modelling the data flow is also discussed in [13].

Although the maximum deviation values of up to 38% are not regarded to be adequate for an engineering approach to performance prediction, most of the prediction results lead to the assumption that the proposed method is accurate for the purpose of improving prediction quality and introduce an engineering approach to the prediction of quality of service properties of component based software systems.

Another result of the case study is that in our case study QML was a suitable means to describe and analyse the QoS of a component. A feature of QML that turns out to be a problem in the practical application of QML is its high customizability. Since there are no established standards for the description of QoS properties using QML, this can cause that compatible specifications are not recognized or that design patterns that are able to solve a certain mismatch can not be found in an automatic way. Another problem is the interpretability of the specified distribution functions, especially if these are defined by the usage of quantiles. This leads to the need of customizing the analyzing application to the interpretation used in the given specification, if this is described in the documentation of the corresponding component.

VI. Related Work

The development of systematic approaches to adaptation of components in order to resolve interoperability problems is still a field of active research. Many papers are based on the work done by Yellin and Strom [3], who introduced an algorithm for the (semi-)automatic generation of adapters using protocol information and an external adapter specification. Bracciali et al. propose the use of some kind of process calculus to enhance this process and generate adapters using PROLOG [14]. Min et al. present an approach called Smart Connectors which allows the construction of adapters based on the provided and required interface of the components to connect [15].

QoS prediction models have to be used to predict the QoS impact. A survey has been published recently by Balsamo et al. [16]. Additionally, simulative approaches can be used and the simulation environment can be generated by the adapter generator, for example the simulator presented by Balsamo and Marzolla [17]. Nevertheless, none of these approaches has a specialized method for including adapters in their predictions.

Further patterns can be taken from the pattern literature, either on the design or the architectural level. As a starting point standard pattern literature can be used [18], [19], [20], [8]. Some of them are suited for functional or extra-functional adaptation.

VII. Conclusion

This paper presents an approach to analyze QoS related interoperability problems with the aim of generating adapters to bridge these problems. Additionally, not only the adapters are generated but also a prediction model dealing with the QoS impact of the composition of the adapter and the adapted component. We aimed at gaining a higher accuracy in these models by exploiting the input of the generator and the code templates used to generate the adapter. A case study is presented showing the application of the approach and the accuracy of the predicted figures.

The results presented here show that this approach is practically applicable for the generation and prediction of the QoS of component adapters. The predictions clearly depict the influences of feature-based configuration that can also be seen in the measurement results. This leads to the conclusion that the consideration of features of adapter generators leads to an increase in accuracy of the prediction models. Hence, the method is suited to increase the predictability of adapted components during the design phase and can be used to increase the precision of performance prediction methods.

Future work on the presented approach will include further case studies with different patterns and interface models. In addition to this the prediction methods will have to be enhanced and more prediction methods have to be evaluated. A focus should thereby be laid on the ability to model the usage profile in an adequate way. Furthermore the restrictions that had to be made to be able to conduct the case study will have to be weakened to achieve a generally applicable method.

References

[1] S. Becker, S. Overhage, and R. Reussner, "Classifying Software Component Interoperability Errors to Support Component Adaption," in *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds., vol. 3054. Berlin, Heidelberg: Springer, May 2004, pp. 68–83.

[2] S. Becker, "Using Generated Design Patterns to Support QoS Prediction of Software Component Adaptation," in *Proceedings of the Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 05)*, C. Canal, J. M. Murillo, and P. Poizat, Eds., July 2005.

[3] D. Yellin and R. Strom, "Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors," in *Proceedings of the 9th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-94)*, ser. ACM Sigplan Notices, vol. 29, 10, 1994, pp. 176–190.

[4] M. Autili, P. Inverardi, and M. Tivoli, "Automatic Adaptor Synthesis for Protocol Transformation," in *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04)*, 2004.

[5] S. Becker and R. H. Reussner, "The Impact of Software Component Adaptors on Quality of Service Properties," in *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 04)*, C. Canal, J. M. Murillo, and P. Poizat, Eds., June 2004.

[6] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Addison-Wesley, Reading, MA, USA, 2000.

[7] S. Frølund and J. Koistinen, "Quality-of-Service Specification in Distributed Object Systems," Hewlett Packard, Software Technology Laboratory, Tech. Rep. HPL-98-159, Sept. 1998.

[8] M. Kircher and P. Jain, *Pattern-Oriented Software Architecture: Patterns for Distributed Services and Components*. John Wiley and Sons Ltd, 2004.

[9] D. Hamlet, D. Mason, and D. Woit, *Component-Based Software Development: Case Studies*, ser. Series on Component-Based Software Development. World Scientific Publishing Company, March 2004, vol. 1, ch. Properties of Software Systems Synthesized from Components, pp. 129–159.

[10] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, "Towards an Engineering Approach to Component Adaptation," in *Architecting Systems with Trustworthy Components*, vol. 3938, 2006, pp. 193–215, to appear April 2006.

[11] N. Streekmann, "Einfluss von Generatorkonfigurationen auf die QoS-Vorhersage für Komponentenadapter," Diplomarbeit, University of Oldenburg, Jan. 2006.

[12] V. Firus, S. Becker, and J. Happe, "Parametric Performance Contracts for QML-specified Software Components," in *Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*, ser. Electronic Notes in Theoretical Computer Science. ETAPS 2005, 2005.

[13] H. Koziolek and S. Becker, "Transforming Operational Profiles of Software Components for Quality of Service Predictions," in *Tenth International Workshop on Component Oriented Programming (WCOP2005)*, July 2005.

[14] A. Bracciali, A. Brogi, and C. Canal, "Dynamically Adapting the Behaviour of Software Components," in *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, York, UK, April 8–11, 2002, Proceedings*, ser. Lecture Notes in Computer Science, F. Arbab and C. L. Talcott, Eds., vol. 2315. Springer-Verlag, Berlin, Germany, 2002, pp. 88–95.

[15] H. G. Min, S. W. Choi, and S. D. Kim, "Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition." in *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, Eds., vol. 3054. Springer-Verlag, Berlin, Germany, 2004, pp. 40–47.

[16] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, May 2004.

[17] S. Balsamo and M. Marzolla, "A Simulation-Based Approach to Software Performance Modeling," in *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, 2003, pp. 363–366.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley & Sons, New York, NY, USA, 1996.

[20] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects*. Wiley & Sons, New York, NY, USA, 2000.

Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability

Michael Mattsson, Håkan Grahn, and Frans Mårtensson
Department of Systems and Software Engineering
School of Engineering, Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{Michael.Mattsson, Hakan.Grahn, Frans.Martensson}@bth.se, http://www.bth.se/besq

Abstract

The software architecture has been identified as an important part of a software system. Further, the software architecture impacts the quality attributes of a system, e.g., performance and maintainability. Therefore, methods for evaluating the quality attributes of software architectures are important. In this paper, we present a survey of software architecture evaluation methods. We focus on methods for evaluating one or several of the quality attributes performance, maintainability, testability, and portability. Based on a literature search and review of 240 articles, we present and compare ten evaluation methods. We have found that most evaluation methods only address one quality attribute, and very few can evaluate several quality attributes simultaneously in the same framework or method. Further, only one of the methods includes trade-off analysis. Therefore, our results suggest an increased research focus on software architecture evaluation methods than can address several quality attributes and the possible trade-offs between different quality attributes.

1. Introduction

The software engineering discipline is becoming more wide-spread in industry and organizations due to the increased presence of software and software-related products and services in all areas. Simultaneously, this demands for new concepts and innovations in the development of the software.

During the last decades, the notion of software architecture has evolved and today, a software architecture is a key asset for any organization that builds complex software-intensive systems [5, 8, 34]. A software architecture is created early in the development and gives the developers a means to create a high level design for the system, making sure that all requirements that has to be fulfilled will be possible to implement in the system.

There exists a number of definitions of software architecture with minor differences depending on domain and people's experience. However, most definitions share common characteristics that can be exemplified by looking at the definition by Bass et al. [5]:

> *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [5]*

This means that the architecture describes which high level components a software system consists of as well as which responsibilities that these components have towards other components in the system. It also describes how these components are organized, both on a conceptual level as well as a decomposed detailed level since there can be an architectural structure inside components as well. Finally the architecture defines which interfaces the components present to other components and which interfaces and components that they use.

The architecture is created based on a set of requirements that it has to fulfil. These requirements are collected from the stakeholders of the system, e.g., users and developers. The functional requirements describe what the system should do, e.g., the functions that the system should provide to the users. Quality requirements describe a set of qualities that the stakeholders want the systems to have, e.g., how long time it may take to complete a certain operation, how easy it is to maintain the system. Other examples of quality attributes are availability, testability, and flexibility. In order to help software developers make sure that a software architecture will be able to fulfil the quality requirements, several methods for evaluating software architectures have been proposed.

In this paper we present a survey of software architecture evaluation methods. We focus our survey on methods that address one or more of the quality attributes perfor-

mance, maintainability, testability, and portability. We think that this selection of quality attributes is relevant for development of software systems that will be used and maintained over a long period of time. The methods are described and compared based on a set of criteria.

There are related evaluation methods that we have chosen to exclude from our survey. One class of related evaluation methods are targeted for components and middleware, e.g., i-Mate [27]. These methods are excluded since they do not evaluate the whole architecture of a system. Further, we have exclude many formal methods, e.g., Promela/SPIN [16, 27], which are more targeted for evaluating correctness and consistency of an architecture but not those quality attributes that we are interested in.

In addition, there are other factors then quality requirements that influence the architecture such as organizational, technical and product factors as well as risk management and project management issues. These factors and issues are not addressed since the majority of the found articles do not address these issues.

The rest of the paper is organized as follows. In the next section we introduce the concept of software architecture evaluation. In Section 3 and Section 4, we present software quality attributes in general and those that we address in this paper, respectively. In Section 5, we discuss related work and present how our survey relates to other surveys in the area. Then, in Section 6, we present the architecture evaluation methods that we include in our survey. Finally, we discuss our findings in Section 7 and conclude our study in Section 8.

2. Software Architecture Evaluation

Architecture evaluations can be performed in one or more stages of the software development process. They can be used to compare and identify strengths and weaknesses in different architecture alternatives during the early design stages. They can also be used for evaluation of existing systems before future maintenance or enhancement of the system as well as for identifying architectural drift and erosion. Software architecture evaluation methods can be divided into four main categories, i.e., experience-based, simulation-based, mathematical modelling based. Methods in the categories can be used independently but also be combined to evaluate different aspects of a software architecture, if needed [8].

Experience-based evaluations are based on the previous experience and domain knowledge of developers or consultants [2]. People who have encountered the requirements and domain of the software system before can based on the previous experience say if a software architecture will be good enough [8].

Simulation-based evaluations rely on a high level implementation of some or all of the components in the software architecture. The simulation can then be used to evaluate quality requirements such as performance and correctness of the architecture. Simulation can also be combined with prototyping, thus prototypes of an architecture can be executed in the intended context of the completed system. Examples of methods in this group are Layered Queuing Network (LQN) [1] approaches and event-based methods such as RAPIDE [28, 29].

Mathematical modelling uses mathematical proofs and methods for evaluating mainly operational quality requirements such as performance and reliability [34] of the components in the architecture. Mathematical modelling can be combined with simulation to more accurately estimate performance of components in a system.

Scenario-based architecture evaluation tries to evaluate a particular quality attribute by creating a scenario profile that forces a very concrete description of the quality requirement. The scenarios from the profile are then used to step through the software architecture and the consequences of the scenario are documented. Several scenario-based evaluation methods have been developed, e.g., Software Architecture Analysis Method (SAAM) [19], Architecture Trade-off Analysis Method (ATAM) [21], and Architecture Level Modifiability Analysis (ALMA) [6, 7].

3. Quality Attributes

Software quality is defined as the degree to which software possesses a desired combination of attributes [17]. According to [8] the quality requirements that a software architecture has to fulfil is commonly divided in two main groups based on the quality they are requesting, i.e., development and operational qualities. A development quality requirement is a requirement that is of importance for the developers work, e.g., maintainability, understandability, and flexibility. Operational quality requirements are requirements that make the system better from the users point of view, e.g. performance and usability. Depending on the domain and priorities of the users and developers, quality requirements can become both development and operational, such as performance in a real-time system.

A quality attribute can be defined as a property of a software system [5]. A quality requirement is a requirement that is placed on a software system by a stakeholder; a quality attribute is what the system actually presents once it has been implemented. During the development of the architecture it is therefore important to validate that the architecture has the required quality attributes, this is usually done using one or more architecture evaluations.

4. Quality Attributes in Focus

This survey focuses on software architecture evaluation methods that address one or more of the following quality attributes: performance, maintainability, testability, and portability. The IEEE standard 610.12-1990 [17] defines the four quality attributes as:

Maintainability. This is defined as:

> *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."*

Maintainability is a multifaceted quality requirement. It incorporates aspects such as readability and understandability of the source code. Maintainability is also concerned with testability to some extent as the system has to be re-validated during the maintenance.

Performance. Performance is defined as:

> *"The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."*

There are many aspects of performance, e.g., latency, throughput, and capacity.

Testability. Testability is defined as:

> *"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."*

We interpret this as the effort needed to validate the system against the requirements. A system with high testability can be validated quickly.

Portability. Portability is defined as:

> *"The ease with which a system or component can be transferred from one hardware or software environment to another."*

We interpret this as portability not only between different hardware platforms and operating systems, but also between different virtual machines and versions of frameworks.

These four quality attributes are selected, not only for their importance for software developing organizations in general, but also for their relevance for organizations developing software in the real-time system domain in a cost effective way, e.g., by using a product-line approach. Performance is important since a system must fulfil the performance requirements, if not, the system will be of limited use, or not used. The long-term focus forces the system to be maintainable and testable, it also makes portability important since the technical development on computer hardware technology moves quickly and it is not always the case that the initial hardware is available after a number of years.

5. Related Work

Surveying software architecture evaluation methods has, as far as we know, been done in four previous studies. In two of the cases, Dobrica and Niemelä [11] and Babar et al. [3], the software architecture evaluation methods are compared with each other in a comparison framework, specific for each study. The survey by Etxeberria and Sagardui [13] compares architecture evaluation methods with respect to the context of architectures in software product lines. The last survey, by Kazman et al. [20], does not address a large number of architecture evaluation methods but uses two evaluation methods as examples for illustrating how the methods fulfil a number of criteria the authors argue are highly needed for an architecture evaluation method to be usable.

The Dobrica and Niemelä survey [11], the earliest one, presents and compares eight of the "most representative", according to themselves, architecture evaluation methods. The discussion of the evaluation methods focus on 1) discovering differences and similarities and 2) making classifications, comparisons and appropriateness studies. The comparison and characterization framework in the survey comprises the following elements; the methods goal, which evaluation techniques are included in the method, quality attributes (what quality attributes and what number of quality attributes is considered), the software architecture description (what views are the foci and in which development phase), stakeholders' involvement, the activities of the method, support for a reusable knowledge base and the validation aspect of the evaluation method.

The objective of the Babar et al. survey [3] is to provide a classification and comparison framework by discovering commonalities and differences among eight existing scenario-based architecture evaluation methods. To a large extent, the framework comprises features that are either supported by most of the existing methods or reported as desirable by software architecture researchers and practitioners. The framework comprises the following elements; the method's Maturity stage, what definition of software architecture is required, process support, the method's activities, goals of the method, quality attributes, applicable project stage, architectural description, evaluation approaches (what types of evaluation approaches are included in the method?), stakeholders involvement, sup-

port for non-technical issue, the method's validation, tool support, experience repository, and resources required.

The survey by Etxeberria and Sagarduia [13] addresses an evaluation framework for software architecture evaluation methods addressing software product-line architectures. Since the life span of a product-line architecture is longer than for ordinary software architectures evolution is one prioritized quality attribute that deserves extra attention in an evaluation. There exist other quality attributes as well, e.g. variability. The context of software product lines imposes new requirements on architecture evaluation methods and this is discussed by Etxeberria and Sagarduia and reflects their classification framework. The framework comprises the following elements; The goal of the method, attribute types (what domain engineering and application engineering quality attributes are addressed), evaluation phase (in the product line context the evaluation can take place on different phases in application engineering and domain engineering, respectively, as well as in a synchronization phase between the two), evaluation techniques, process description, the method's validation and relation to other evaluation methods.

The purpose of the last survey, by Kazman et al. [20], is primary to provide criteria that are important for an evaluation method to address, and not to compare existing evaluation methods. The authors argue for criteria addressing what it means to be an effective method, one that produces results of real benefit to the stakeholders in a predictable repeatable way, and a usable method one that can be understood and executed by its participants, learned reasonably quickly, and performed cost effectively. Thus, the survey ends up with the following four criteria: 1) Context and goal identification, 2) Focus and properties under examination, 3) Analysis Support, and 4) Determining analysis outcomes.

The survey by Dobrica and Niemelä [11] provides an early, initial overview of the software architecture evaluation methods. This was followed up by the survey by Babar et al. [3] that presents a more detailed break-down (including requirements on detailed method activities etc.) and a more holistic perspective, e.g., process support, tool support. The survey by Kazman et al. [20] presents additional requirements on what a software architecture method should support. The software product-line context survey by Etxeberria and Sagarduia [13] addresses evaluation methods from a prescribed way of developing software. This perspective opened up some additional phases where an evaluation can take place and put product-line important quality attributes more in focus, e.g., variability and maintainability.

Our survey takes the perspective from a set of quality attributes that are of general importance for software developing organizations. This means that we are taking a more solution-oriented approach, i.e., we are focusing on finding knowledge about what existing evaluation methods can provide with respect to the identified quality attributes. We are not aiming at obtaining knowledge about general software architecture evaluation methods or pose additional requirements on the methods due to some completeness criteria or specific way of developing the software, as in the four performed surveys. We may add additional requirements on the evaluation method, but if that is the case, the requirements will have its origin from the four quality attributes addressed, performance, testability, maintainability and portability.

6. Architecture Evaluation Methods

In this survey each of the software architecture evaluation methods will be described according to a pre-defined template. The template structures the description of the architecture according to the following elements: Name and abbreviation (if any), Category of method, Reference(s) where the method are described in detail, Short description of the method, Evaluation goal of the method, How many quality attributes the method addresses, (one, many, or many where trade-off approaches exist), What specific quality attributes the method address (or if it is a more general evaluation method) and finally, the usage of the method. Table 1 summarizes the template with indication of potential values for each element.

The initial selection of research papers was made by searching through Compendex, Inspec, and IEEE Xplore. The search Compendex and Inspec resulted in 194 papers, and the search in IEEE Xplore produced an additional 46 papers. The query used for the searched used the following keywords, "software architecture" and "any of evaluation, assessment or analysis" and "at least one of performance, maintainability, testability, or portability". The keywords where truncated and stemmed when possible. In total, we had 240 papers found from the database searches. We then eliminated duplicate papers and papers that did not fulfil our criteria of addressing one or more of the quality attributes performance, maintainability, testability, or portability. After the screening we had about 25 papers that contained architecture evaluation methods and experience reports from their use. From these papers we have identified 10 methods and approaches that can be applied for architecture-level evaluation of performance, maintainability, testability, or portability.

Table 1. Method Description Template

Item	Potential values
Name and abbreviation	The method's name and abbreviation (if any)
Category of method	Experience-based, Simulation-based, Scenario-based, Mathematical modelling, or a mix of categories
Reference(s)	One or more literature source(s)
Short description of the method	Text summary of the method
Evaluation goal	Text description of goal
Number of quality attributes addressed	One, many, or many with trade-off approach
Specific quality attributes addressed	Any of Maintainability, Performance, Testability, Portability, General and any additional ones
Method usage	Has the method been used by the method developer(s) only or by some other?

6.1. SAAM — Software Architecture Analysis Method

Software Architecture Analysis Method (SAAM) [19] is a scenario-based software architecture evaluation method, targeted for evaluating a single architecture or making several architectures comparable using metrics such as coupling between architecture components.

SAAM was originally focused on comparing modifiability of different software architectures in an organization's domain. It has since then evolved to a structured method for scenario-based software architecture evaluation. Several quality attributes can be addressed, depending on the type of scenarios that are created during the evaluation process. Case-studies where maintainability and usability are evaluated have been reported in [18], and modifiability, performance, reliability, and security are explicitly stated in [21].

The method consists of five steps. It starts with the documentation of the architecture in a way that all participants of the evaluation can understand. Scenarios are then developed that describe the intended use of the system. The scenarios should represent all stakeholders that will use the system. The scenarios are then evaluated and a set of scenarios that represents the aspect that we want to evaluate is selected. Interacting scenarios are then identified as a measure of the modularity of the architecture. The scenarios are then ordered according to priority, and their expected impact on the architecture. SAAM has been used and validated in several studies [10, 12, 18, 19, 25]. There also exist methods that are extensions and/or further evolutions of SAAM, which are surveyed by Dobrica and Niemelä [11].

6.2. ATAM — Architecture Trade-off Analysis Method

Architecture Trade-off Analysis Method (ATAM) [21] is a scenario-based software architecture evaluation method. The goals of the method are to evaluate architecture-level designs that considers multiple quality attributes and to gain insight as to whether the implementation of the architecture will meet its requirements. ATAM builds on SAAM and extends it to handle trade-offs between several quality attributes.

The architecture evaluation is performed in six steps. The first one is to collect scenarios that operationalize the requirements for the system (both functional and quality requirements). The second step is to gather information regarding the constraints and environment of the system. This information is used to validate that the scenarios are relevant for the system. The third step is to describe the architecture using views that are relevant for the quality attributes that were identified in step one. Step four is to analyze the architecture with respect to the quality attributes. The quality attributes are evaluated one at a time. Step five is to identify sensitive points in the architecture, i.e., identifying those points that are affected by variations of the quality attributes. The sixth and final step is to identify and evaluate trade-off points, i.e., variation points that are common to two or more quality attributes. ATAM has been used and validated in several studies [21, 32].

6.3. ALMA — Architecture-Level Modifiability Analysis

Architecture-Level Modifiability Analysis (ALMA) [6, 7] is a scenario-based software architecture evaluation method with the following characteristics: focus on modifiability, distinguish multiple analysis goals, make important assumptions explicit, and provide repeatable techniques for performing the steps. The goal of ALMA is to provide a structured approach for evaluating three aspects of the maintainability of software architectures, i.e., maintenance prediction, risk assessment, and software architecture comparison.

ALMA is an evaluation method that follows SAAM in its organization. The method specifies five steps: 1. determine the goal of the evaluation, 2. describe the software architecture, 3. elicit a relevant set of scenarios, 4. evaluate the scenarios, and 5. interpretation of the results and draw conclusions from them. The method provides more detailed descriptions of the steps involved in the process than SAAM does, and tries to make it easier to repeat evaluations and compare different architectures. It makes use of structural metrics and base the evaluation of the scenarios on quantification of the architecture. The method has been used and validated by the authors in several studies [6, 7, 24].

6.4. RARE/ARCADE

RARE and ARCADE are part of a toolset called SEPA (Software Engineering Process Activities) [4]. RARE (Reference Architecture Representation Environment) is used to specify the software architecture and ARCADE is used for simulation-based evaluation of it. The goal is to enable automatic simulation and interpretation of a software architecture that has been specified using the RARE environment.

An architecture description is created using the RARE environment. The architecture description together with descriptions of usage scenarios are used as input to the ARCADE tool. ARCADE then interprets the description and generates a simulation model. The simulation is driven by the usage scenarios. RARE is able to perform static analysis of the architecture, e.g., coupling. ARCADE makes it possible to evaluate dynamic attributes such as performance and reliability of the architecture. The RARE and ARCADE tools are tightly integrated to simplify an iterative refinement of the software architecture. The method has, as far as we know, only been used by the authors.

6.5. Argus-I

Argus-I [37] is a specification-based evaluation method. Argus-I makes it possible to evaluate a number of aspects of an architecture design. It is able to perform structural analysis, static behavioral analysis, and dynamic behavioral analysis, of components. It is also possible to perform dependence analysis, interface mismatch, model checking, and simulation of an architecture.

Argus-I uses a formal description of a software architecture and its components together with statecharts that describe the behavior of each component. The described architecture can then be evaluated with respect to performance, dependence, and correctness. There is no explicit process defined that the evaluation should follow, but some guidance is provided. The evaluation results in a quantification of the qualities of the architecture. The performance of the architecture is estimated based on the number of times that components are invoked. The simulation can be visualized using logs collected during the simulation. The method has, as far as we know, only been used by the authors.

6.6. LQN — Layered Queuing Networks

Layered queuing network models are very general and can be used to evaluate many types of systems. Several authors have proposed the use of queuing network models for software performance evaluation [14, 15, 22, 30, 33]. Further, there also exist many tools and toolkits for developing and evaluating queuing network models, e.g., [14, 15]. A queuing network model can be solved analytically, but is usually solved using simulation.

The method relies on the transformation of the architecture into a layered queuing network model. The model describes the interactions between components in the architecture and the processing times required for each interaction. The creation of the models require detailed knowledge of the interaction of the components, together with behavioral information, e.g., execution times or resource requirements. The execution times can either be identified by, e.g.,. measurements, or estimated. The more detailed the model is the more accurate the simulation result will be.

The goal when using a queuing network model is often to evaluate the performance of a software architecture or a software system. Important measures are usually response times, throughput, resource utilization, and bottleneck identification. In addition, some tools not only produce measures, but also have the ability to visualize the system behavior.

6.7. SAM

SAM [38] is a formal systematic methodology for software architecture specification and analysis. SAM is mainly targeted for analyzing the correctness and performance of a system.

SAM has two major goals. The first goal is the ability to precisely define software architectures and their properties, and then perform formal analysis of them using formal methods. Further, SAM also supports an executable software architecture specification using time Petri nets and temporal logic. The second goal is to facilitate scalable software architecture specification and analysis, using hier-

archical architectural decomposition. The method has, as far as we know, only been used by the authors.

6.8. EBAE — Empirically-Based Architecture Evaluation

Lindvall et al. describe in [26] a case study of a redesign/reimplementation of a software system developed more or less in-house. The main goal was to evaluate the maintainability of the new system as compared to the previous version of the system. The paper outlines a process for empirically-based software architecture evaluation. The paper defines and uses a number of architectural metrics that are used to evaluate and compare the architectures.

The basic steps in the process are: select a perspective for the evaluation, define/select metrics, collect metrics, and evaluate/compare the architectures. In this study the evaluation perspective was to evaluate the maintainability, and the metrics were structure, size, and coupling. The evaluations were done in a late development stage, i.e., when the systems already were implemented. The software architecture was reverse engineered using source code metrics.

6.9. ABAS — Attribute-Based Architectural Styles

Attribute-Based Architectural Styles (ABASs) [23] build on the concept of architectural styles [9, 35], and extend it by associating a reasoning framework with an architectural style. The method can be used to evaluate various quality attributes, e.g., performance or maintainability, and is thus not targeted at a specific set of quality attribute.

The reasoning framework for an architectural style can be qualitative or quantitative, and are based on models for specific quality attributes. Thus, ABASs enable analysis of different quality aspects of software architectures based on ABASs. The method is general and several quality attributes can be analyzed concurrently, given that quality models are provided for the relevant quality attributes. One strength of ABASs is that they can be used also for architectural design. Further, ABASs have been used as part of evaluations using ATAM [21].

6.10. SPE — Software Performance Engineering

Software performance engineering (SPE) [36, 39] is a general method for building performance into software system. A key concept is that the performance shall be taken into consideration during the whole development process, not only evaluated or optimized when the system already is developed.

SPE relies on two different models of the software system, i.e., a software execution model and a system execution model. The software execution model models the software components, their interaction, and the execution flow. In addition, key resource requirements for each component can also be included, e.g., execution time, memory requirements, and I/O operations. The software execution model predicts the performance without taken contention of hardware resources into account.

The system execution model is a model of the underlaying hardware. Examples of hardware resources that can be modelled are processors, I/O devices, and memory. Further, the waiting time and competition for resources are also modelled. The software execution model generates input parameters to the system execution model. The system execution model can be solved by using either mathematical methods or simulations.

The method can be used to evaluate various performance measures, e.g., response times, throughput, resource utilization, and bottleneck identification. The methods is primarily targeted for performance evaluation. However, the authors argue that their method can be used to evaluate other quality attributes in a qualitative way as well [39]. The method has been used in several studies by the authors, but do not seem to have been used by others.

6.11. Summary of Architecture Evaluation Methods

Table 2 summarizes the most important characteristics (see Table 1) of our survey of software architecture evaluation methods. As we can see, most of the methods address only one quality attribute of those that we consider in this survey, and the most common attribute to address is performance. Surprisingly, no method was found that specifically address portability or testability. Further, we can observe that only one method exists that support trade-off analysis of software architectures. Finally, we also observe that only two methods seem to have been used by others than the method inventor.

7. Discussion

Despite the promising number of primary studies found, i.e., 240, it turned out that only 10 software architecture evaluation methods were possible to identify that addressed one or more of the performance, maintainability, testability, or portability quality attributes. There exist several reasons for this large reduction of the number of articles. First, there were some duplicate entries of the same article since we searched several databases. Second, a large portion of the papers evaluated one or several quality

Table 2. Summary of evaluation method characteristics.

Name	Category	Quality attributes	Method usage
SAAM [19]	Scenario-based	General	creator [18, 19], other [10, 12, 25]
ATAM [21]	Scenario-based	General, trade-off	creator [21], other [32]
ALMA [6, 7]	Scenario-based	Modifiabity	creator [6, 7, 24]
RARE/ARCADE [4]	Simulation-based	Performance	creator [4]
ARGUS-I [37]	Simulation-based	Performance	creator [37]
LQN [33]	Simulation-based	Performance	creator [1, 33]
SAM [38]	Simulation-based	Performance	creator [38]
EBAE [26]	Experience-based, metrics	Maintainability	creator [26]
ABAS [23]	Experience-based	General	creator [23]
LQN [33]	Simulation-based, mathematical modelling	Performance	creator [33, 1], other
SPE [36, 39]	Simulation-based, mathematical modelling	Performance	creator [36, 39]

attributes in a rather ad hoc fashion. As a result, we excluded those papers from our survey since they did not document a repeatable evaluation method or process. Third, several papers addressed both hardware and software evaluations, thus they did not qualify in our survey with its focus on methods for software architecture evaluation.

Continuing with the ten remaining articles, we found that five of the methods addressed only one single quality attribute. Only one (ATAM) of the remaining five methods addressing multiple attributes provide support for trade-off analysis between the quality attributes. No specific methods evaluated testability or portability explicitly. These quality attributes could be addressed by any of the three evaluation methods that are more general in their nature, i.e., that could address more arbitrary selected quality attributes, ATAM [21], SAAM [19], or the method by Lindvall et al. [26].

Many of the methods have been used several times of the authors. Multiple use of the method indicates an increase in validity of the method. However, only two methods have been used by others than the original authors of the method. We believe that external use of a method is an indication of the maturity of the method. These two methods are SAAM and ATAM. However, experience papers that use a method in whole or part are particularly difficult to identify, since the evaluation method that has been used is not always clearly stated.

8. Conclusions

The architecture of a software system has been identified as an important aspect in software development, since the software architecture impacts the quality attributes of a system, e.g., performance and maintainability. A good software architecture increases the probability that the system will fulfil its quality requirements. Therefore, methods for evaluating the quality attributes of software architectures are important.

In this paper, we present a survey of evaluation methods for software architecture quality attribute evaluation. We focus on methods for evaluating one or several of the quality attributes performance, maintainability, testability, and portability. Methods that evaluate several quality attributes and/or trade-off analysis are especially interesting. Based on a broad literature search in major scientific publication databases, e.g., Inspec, and reviewing of 240 articles, we present and compare ten evaluation methods.

We have found that many evaluation methods only address one quality attribute, and very few can evaluate several quality attributes simultaneously in the same framework or method. Specifically, only one of the methods includes trade-off analysis. Further, we have identified that many methods are only used and validated by the method inventors themselves.

In summary, our results suggest

- an increased research focus on software architecture evaluation methods than can address several quality attributes simultaneously,
- an increased research focus on software architecture evaluation methods than can address the possible trade-offs between different quality attributes, and
- an increased focus on validation of software architecture evaluation methods by people other than the method inventors.

Acknowledgment

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" http://www.bth.se/besq.

References

[1] Aquilani, F., Balsamo, S., and Inverardi, P., "Performance Analysis at the Software Architectural Design Level," *Performance Evaluation*, vol. 45, pp. 147-178, 2001.

[2] Avritzer, A. and Weyuker E. J., "Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews," *Empirical Software Engineering*, 4(3):199-215, 1999.

[3] Babar, M. A., Zhu, L., and Jeffery, R., "A framework for classifying and comparing software architecture evaluation methods," *Proc. Australian Software Engineering Conference*, pp. 309-318, 2004.

[4] Barber, K. S., Graser, T., and Holt, J., "Enabling iterative software architecture derivation using early non-functional property evaluation," *Proc. 17th IEEE International Conference on Automated Software Engineering*, pp. 23-27, 2002.

[5] Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice*, ISBN 0-631-21304-X, Addison-Wesley, 2003.

[6] Bengtsson, PO., *Architecture-Level Modifiability Analysis*, ISBN 91-7295-007-2, Blekinge Institute of Technology, Dissertation Series No 2002-2, 2002.

[7] Bengtsson, PO., Lassing, N., and Bosch, J., "Architecture Level Modifiability Analysis (ALMA)," *Journal of Systems and Software*, vol. 69, pp. 129-147, 2004.

[8] Bosch, J., *Design & Use of Software Architectures – Adopting and evolving a product-line approach*, ISBN 0-201-67494-7, Pearson Education, 2000.

[9] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M., *Pattern-Oriented Software Architecture - A System of Patterns*, ISBN 0-471-95869-7, Wiley, 1996.

[10] Castaldi, M., Inverardi, P., and Afsharian, S., "A case study in performance, modifiability and extensibility analysis of a telecommunication system software architecture," *Proc. 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pp. 281-290, 2002.

[11] Dobrica, L. and Niemelä, E., "A Survey On Architecture Analysis Methods," *IEEE Transactions on Software Engineering*, 28(7):638-653, 2002.

[12] Eikelmann, N. S. and Richardson, D. J., "An Evaluation of Software Test Environment Architectures," *Proc. 18th International Conference on Software Engineering*, pp. 353-364, 1996.

[13] Etxeberria, L. and Sagardui, G., "Product-Line Architecture: New Issues for Evaluation," *Lecture Notes in Computer Science*, Volume 3714, ISBN 3-540-28936-4, Springer-Verlag GmbH, 2005.

[14] Franks, G., Hubbard, A., Majumdar, S., Petriu, D., Rolia, J., and Woodside C.M., "A Toolset for Performance Engineering and Software Design of Client-Server Systems," *Performance Evaluation*, 24(1-2):117-136, November 1995.

[15] Gunther, N., *The Practical Performance Analyst*, ISBN 0-07-912946-3, McGraw-Hill, 1998.

[16] Holzmann, G.J., "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, 23(5):279-295, May 1997.

[17] IEEE std 610.12-1990 (n.d.). *IEEE Standard Glossary of Software Engineering Terminology*, 1990. Retrieved January 19, 2006. Web site: http://ieeexplore.ieee.org/

[18] Kazman, R., Abowd, G., Bass, L., and Clements, P., "Scenario-based analysis of software architecture," *IEEE Software*, 13(6):47-55, November 1996.

[19] Kazman, R., Bass, L., Abowd, G., and Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. 16th International Conference of Software Engineering*, pp. 81-90, 1994.

[20] Kazman, R., Bass, L., Klein, M., Lattanze, T., and Northrop, L., "A Basis for Analyzing Software Architecture Analysis Methods," *Software Quality Journal*, 13(4):329-355, 2005.

[21] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, S. J., "The Architecture Tradeoff Analysis Method," *Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 68-78, 1998.

[22] King, P., *Computer and Communication Systems Performance Modelling*, ISBN 0-13-163065-2, Prentice Hall, 1990.

[23] Klein, M. and Kazman, R., "Attribute-Based Architectural Styles," CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University, 1999.

[24] Lassing, N., Bengtsson, P., Van Vliet, H., and Bosch, J., "Experiences with ALMA: Architecture-Level Modifiability Analysis," *Journal of Systems and Software*, 61(1):47-57, March 2002.

[25] Lassing, N., Rijsenbrij, D., and van Vliet, H., "Towards a Broader View on Software Architecture Analysis of Flexibility," *Proc. Sixth Asia-Pacific Software Engineering Conference*, pp. 238-245, 1999.

[26] Lindvall, M., Tvedt, R. T., and Costa, P., "An empirically-based process for software architecture evaluation," *Empirical Software Engineering*, 8(1):83-108, 2003.

[27] Liu, A. and Gorton, I., "Accelerating COTS Middleware Acquisition: The i-Mate Process," *IEEE Software*, 20(2): 72-79, March/April 2003.

[28] Luckham, D. C., "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *Proc. DIMACS workshop on Partial order methods in verification*, pp. 329-357, Princeton, 1997.

[29] Luckham, D., John, K., Augustin, L., Vera, J., Bryan, D., and Mann, W., "Specification and Analysis of System Architecture using RAPIDE," *IEEE Transactions on Software Engineering*, 21(4):336-335, 1995.

[30] Menascé, D., Almeida, V., and Dowdy, L., *Capacity Planning and Performance Modelling*, ISBN 0-13-035494-5, Prentice Hall, 1994.

[31] Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G.J., "Implementing Statecharts in PROMELA/SPIN," *Proc. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 90-101, October 1998.

[32] Mukkamalla R., Britton M., and Sundaram P., "Scenario-Based Specification and Evaluation of Architectures for Health Monitoring of Aerospace Structures," *Proc. 21st Digital Avionics Systems Conference*, Vol 2, pp. 12E1-1-12E1-12, October 2002.

[33] Petriu, D., Shousha, C., and Jalnapurkar, A., "Architecture-Based Performance Analysis Applied to a Telecommunication System," *IEEE Transactions on Software Engineering*, 26(11):1049-1065, November 2000.

[34] Reusner, R., Schmidt, H.W., and Poernomo, I. H., "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, 66(3):241-252, 2003.

[35] Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, ISBN 0-13-182957-2, Prentice-Hall, 1996.

[36] Smith, C. and Williams, L., *Performance Solutions*, ISBN 0-201-72229-1, Addison-Wesley, 2002.

[37] Vieira, M. E. R., Dias, M. S., and Richardson, D. J., "Analyzing software architectures with Argus-I," *Proc. 22nd International Conference on Software Engineering*, pp. 758-761, 2000.

[38] Wang, J., He, X., and Deng, Y., "Introducing Software Architecture Specification and Analysis in SAM Through an Example," *Information and Software Technology*, 41(7):451-467, May 1999.

[39] Williams, L. G. and Smith, C. U., "Performance Evaluation of Software Architectures," *Proc. 1st International Workshop on Software and Performance*, pp. 164-177, 1998.

Towards a Framework for Large Scale Quality Architecture

Markus Voss, Andreas Hess, Bernhard Humm

***Abstract*— sd&m Research is actively working on "Quasar Enterprise" – a framework of architectural principles and references for large scale quality architecture. In this contribution we present its basic and advanced concepts plus a concrete set of rules for architectural design on enterprise level. By explicitly working out these concepts and rules we can get a lot more tangible on semantics of the hyped concept of service-oriented architecture (SOA).**

***Index Terms*—Application Landscapes, Quality Architecture, Service-Oriented Architecture (SOA)**

I. Introduction

Over the last years, sd&m Research[1] has defined "Quasar" (Quality Software Architecture) – a well founded framework of terms, architectural principles and references for careful planning and robust construction of information systems [1,2,3]. Quasar has evolved to be a big success story since its principles and references have been utilized in dozens of industrial bespoke application development projects at sd&m and also influenced thinking in the German software engineering community quite a bit.

On the other hand, pure bespoke application development projects tend to decrease in relevance compared to projects where component integration and enterprise level architectural design are of prime importance. An architectural concept much discussed in this context recently is that of Service-Oriented Architecture (SOA). SOA promises to be a good way to structure systems on a large scale in order to achieve agility, flexibility and maintainability - properties of IT solutions needed to address ever faster changing requirements of business and cost pressure. Much has been written about SOA [4,5,6,7,8] and sd&m has published some of its experiences with SOA in industrial practice, too [9,10,11]. Studying these contributions, one finds that on a very general level the understanding of SOA is quite unitary. The concept of loose coupling e.g. is considered central in all contributions. Beyond that however, one can make two observations:

- First, the in depth understanding of SOA is quite different, which already starts with the usage of terms.
- Secondly, if it comes to precise rules for designing components and services we haven't found much in the given literature that goes beyond the general concepts.

Therefore sd&m Research over a year ago has started work on "Quasar Enterprise". As with "Quasar" sd&m Research aims at a well founded framework of terms, architectural principles and references for careful planning and robust construction – but this time for architectural work on a large scale. "Quasar Enterprise" – when completely developed – is meant to be our enterprise architects' handbook like [3] is today for our software engineers.

Putting together "Quasar Enterprise" it is important, not to "reinvent the wheel" and search for a completely new theory of large scale architecture. The idea is to dig out proven best practices from our enterprise level architecture and systems integration projects and to match this with the state of the art in scientific discourse to come up with a real synthesis. Today [12] is the most comprehensive coverage of "Quasar Enterprise" in this respect. But even though there is still some way to go to finish "Quasar Enterprise", we think it is worth to share our view of some of its central elements with the community.

In the following we structure and introduce these in three categories:

- The Terms and Basic Concepts: These are the basics to build the framework on. The contribution of "Quasar Enterprise" here is simply to standardize by clear definition. Within this paper we present some of the central concepts of the ontology.
- The Advanced Concepts: These are the architectural ideas and tools to be used to further specify Quality Architecture compared to what is possible with only the general concepts in terms of the least common denominator of what a SOA is. Here the contribution of "Quasar Enterprise" is to dig these out and make them

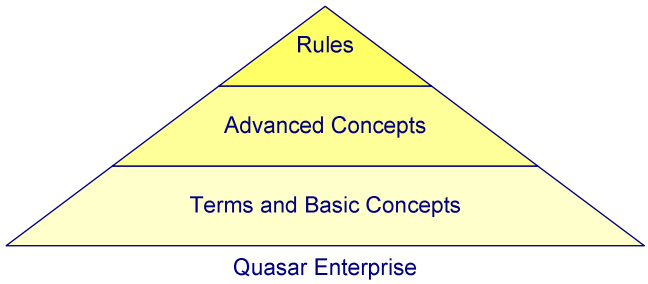


Fig. 1. Elements of the Framework

[1] sd&m Research is the research and technology management unit of the sd&m AG – a German software development, systems integration and IT consulting company of over 1.000 IT professionals. sd&m's focus is on developing and integrating custom solutions.

explicit. Some of these will be outlined in this paper.

- The Rules: These are the actual guidelines on how to design domains, components, services and couplings. Here the contribution of "Quasar Enterprise" will be to put these together in a comprehensive depiction – possibly for the first time. In this paper we present the current draft of the core set of rules and even though we still work on further consolidation, we regard it valuable to present this intermediary result as basis for more in depth discussion with the architectural community.

II. Terms and Basic Concepts

A. Business Architecture and Application Architecture

Different from many approaches to SOA which are technology-driven (e.g. the approaches of many tool vendors, who regard concrete enterprise service bus (ESB) solutions as core), we regard SOA is a business-driven approach to quality architecture. The success factors for reaching the goals of agility and flexibility typically lie within aligning the architecture of the application landscape according to characteristics of the business domains and the business processes. Therefore it is obvious, that a clear distinction between the terms and concepts of business architecture and application architecture is crucial for a sound approach. Doing so, we actually follow the structure that most of the so called Enterprise Architecture Frameworks impose (e.g. Zachman [13], TOGAF [14] or Capgemini's IAF [15]). In [12] we go into more detail on the relationship between these frameworks and "Quasar Enterprise".

For the terms of business architecture we define as follows:

- We define *business architecture* to be the set of all propositions and regularities about business domains, business processes, business services and business objects as well as about their relationships amongst each other.
- A *business domain* is a segment of an enterprise that has influence on processes and organization, e.g. market, customers, suppliers, product or service type, subsidiary structure or sales channel. To shape an enterprise's IT landscape it is important to know, which of the domains are essential and differentiating, since these lead to identify appropriate application domains (see below).
- A *business process* is a sequence of tasks to reach a defined business goal directly or indirectly connected with the enterprise's products or services. It may be organized in sub-processes, the smallest units of which are *elementary tasks* that are to be carried out non-interruptible, by one actor, in one place.
- A *business service* is such an elementary task, which is context free, has a unique actor and a well-defined business goal connected to its execution. It is these business services, that a SOA should be built on in terms of its application services (see below) being deduced from these business services.
- *Business objects* finally are real world objects– material or immaterial. For IT architectural problems we usually work with their model representations which we call *information objects*.

On the other hand the actual subject of our work as enterprise architects or systems integrators is a company's application architecture. For the terms here we define as follows:

- *Application architecture* is the set of all propositions and regularities about application domains, application components, application services and application service-operations as well as about their relationships amongst each other.
- An *application domain* is a set of application components belonging together to be regarded as a unit in conjunction with shaping the application landscape.
- The *application landscape* is the entirety of all application components of an enterprise together with their interconnections in terms of interfaces and data.
- According to [3], a component is an essential element of design, implementation, and planning. It exports and imports interfaces, hides its implementation details, and can be hierarchically structured. There are many other definitions around of what a component is (e.g. [16]), and the concept is general enough to cover the complete spectrum from very large scale components (complete application landscapes) to very small scale components (single software objects or classes). In the context of enterprise level architecture we use the term *application component* to denote a self-contained unit of functionality

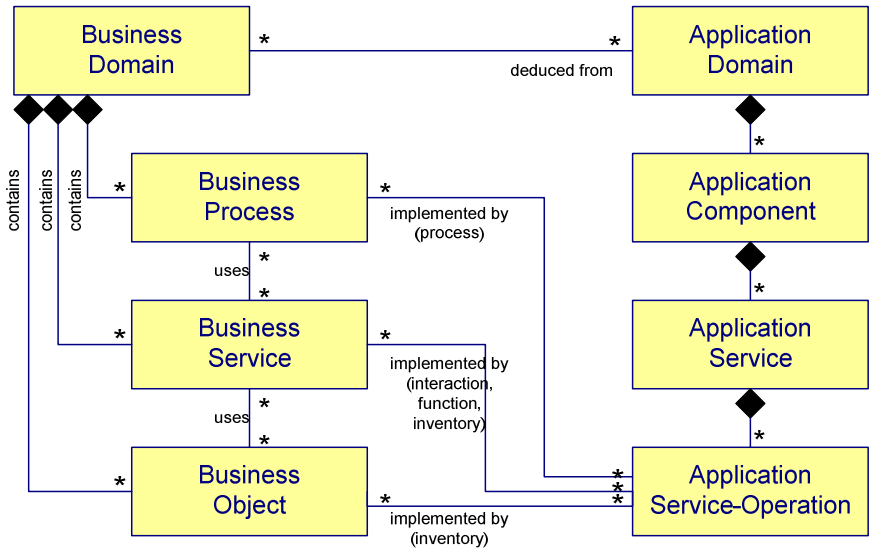


Fig. 2. Terms and Relationships

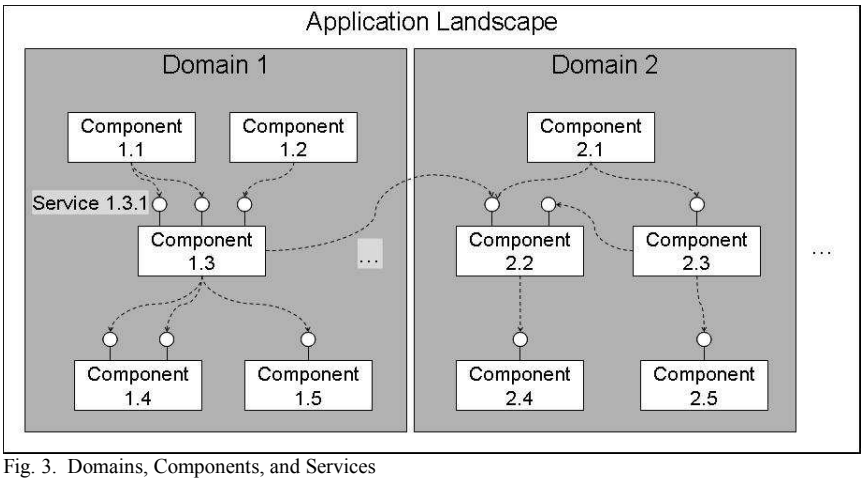


Fig. 3. Domains, Components, and Services

and data, belonging to an application domain and exporting specific parts of its functionality as an application service.

- An *application* is any collection of application components regarded meaningful by system designers.
- An *application service* corresponds with the interface of an application component. It allows access to a defined functionality which can be utilized as part of a comprehensive process. The application service therewith yields an abstraction of the exporting application component hiding its implementation details. It is self-contained and context-independent. The definition of an application service has the character of a contractual agreement between service provider and service user covering syntax, semantics and non functional properties.
- Finally, *application service-operations* are the actual functionalities application services are made of. They have a signature in terms of input and output types and semantics specifiable by pre- and post-conditions.

Fig. 2 shows the most important terms introduced and their relationships.

In the context of enterprise level architecture and when misapprehension is avoided, we only speak of domain, component, service and operation. Fig. 3 shows these elements of application architecture.

B. Service-Oriented Architecture and Quality Architecture

Having the terms defined, one can now formally say, what a Service-Oriented Architecture is. The least common denominator of all the definitions around probably is the following set of statements (defined e.g. in [17]):

- Functionality is encapsulated in services
- Services represent publicly known interfaces
- Services are loosely coupled
- Services are atomic

It is obvious, that these attributes of the architecture are not sufficient to guarantee quality in the sense of the architecture achieving a high ranking in typical quality measures like understandability, adaptability, flexibility, agility, maintainability, manageability, sustainability, cost efficiency, security, etc. Quality Architecture is achieved by constructively searching for an optimum among these measures according to the actual stakeholders' needs. A framework for Quality Architecture like the one we are after with "Quasar Enterprise" therefore needs to assist in this process with more advanced concepts and tools as well as with concrete rules.

III. Advanced Concepts

A. Lay-out Plan

The *lay-out plan* is an architectural tool to analyze and shape application landscapes. It is used to both visualize the as-is structure of the landscape and visualize, how it is planned to be developed in the future. By highlighting two dimensions of the business architecture – often the prime business processes and some other important business domain - it spans an area in which the elements of the application architecture - application domains and components - can be positioned according to how they address or will address the according segments of these domains. Fig. 4 shows an example of a lay-out plan.

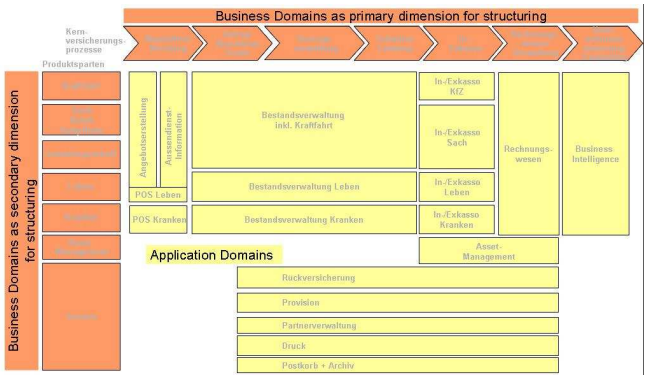


Fig. 4. Lay-Out Plan

Lay-out plans are nothing new and widely used, e.g. as a tool in so called *business-IT alignment* projects. Exactly this is why they are useful in Quality Architecture. They bridge between business and application architecture and support a business-driven approach. In 4.1 we show, how rules with respect to the development plan can help find domains.

B. Service Categories

A second valuable architectural tool is the concept of *service categorization*. Application service-operations are categorized according to their nature in terms of which segment of large scale architecture they primarily support. Related concepts can be found in the literature, e.g. in [8]. From our expertise we found, that the most promising set of categories in terms of quality architecture is the following:

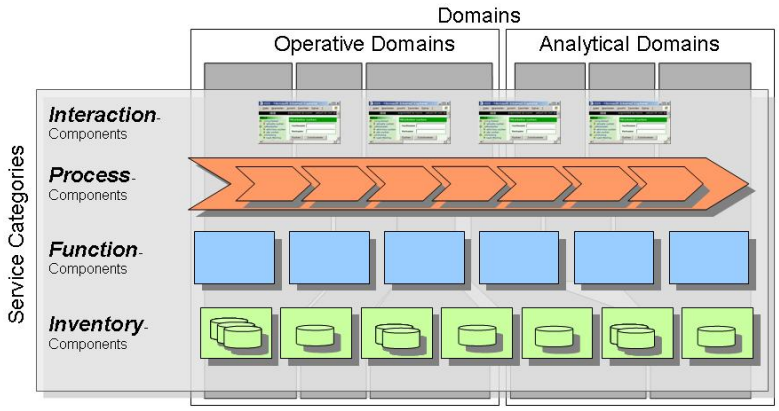


Fig. 5. Reference Architecture according to Service Categories

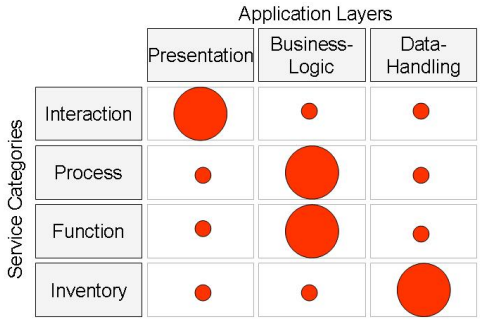


Fig. 6. Service Categories vs. Application Layers

- *Interaction*: Service-operations for user interaction with the application landscape (e.g. in enterprise portals)
- *Process*: Service-operations supplying implemented business processes (e.g. in order management)
- *Function*: Service-operations supplying implemented core business functions (e.g. in billing)
- *Inventory*: Service-operations for managing and accessing business data (e.g. in customer management)

The categorization of service-operations leads to an ordering concept for application services and components. For the later, the ideal of every component being of exactly one category leads to a technical reference architecture on enterprise level. This is shown in Fig. 5.

Note that the concept of categorized application components is not to be confused with the layers or tiers of a single application, often called presentation, business-logic and data-handling. Large scale application components – even though they me be of only one service category – usually consist of sub-components on all three layers. One may only state, that components of a special category are typically "heavier" in the according layers (meaning that they have "more" or "bigger" sub-components within these layers). Fig. 6 illustrates this.

C. Integration Levels

The third architectural tool of importance is the concept of *integration levels*. The basic idea is that application components can be integrated in three forms resembling the layers or tiers of an application as mentioned above:

- *Presentation integration*: Components are coupled in terms of their presentation layer components interacting. This sort of coupling is typically technically implemented by portal server or screen scraping technologies.
- *Service integration*: Components are coupled in terms of their business-logic layer components interacting. This sort of coupling is typically technically implemented by enterprise application integration (EAI) technologies.
- *Data integration*: Components are coupled in terms of their data-handling layer components interacting. This sort of coupling is typically technically implemented by data integration and extract-transform-load (ETL) technologies.

Fig. 7 illustrates the concept.

The concept of integration levels is also not new. One can find e.g. the same levels in SAPs Netweaver architecture [18] (alternatively named people integration, application integration and data integration).

As we see, integration is nothing more than the enterprise level term for *coupling* of application components or applications. Note that the concept of integration levels is not to be confused with the service categories since these are strictly orthogonal. As above, one may only state, that components of a special category are typically "heavier" coupled in the according levels in that they have "more" or "closer" interactions within these levels. Fig. 8 illustrates this.

Talking of Service-Oriented Architecture, the focus usually is on the service integration level. EAI / ESB solutions are often regarded as the typical means for implementing a SOA technically.

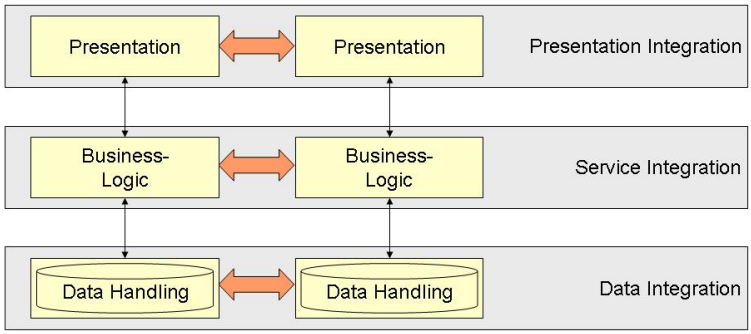


Fig. 7. Integration Levels

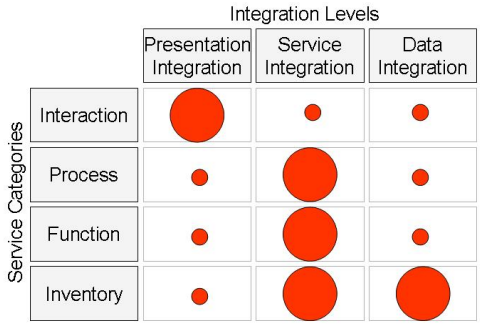


Fig. 8 Service Categories vs. Integration Levels

Talking of Quality Architecture however, choosing the appropriate levels of integration is indeed a measure.

D. Business Process Management and Orchestration

One specific form of service integration as introduced above is by following a *business process management* (BPM) approach. BPM recently is often cited in the same breath with SOA (e.g. [19,20]). Even though – as with SOA – BPM can be regarded as (only) an IT management approach, here we regard it as a specialization of the architectural style of SOA. The BPM approach aims at a SOA, where all elementary steps of a business process are supported by services and all the comprehensive process logic and the tasks of coordination between these elementary services is located in another dedicated service.

Another term of interest in this context is *orchestration*. According to [11] e.g. orchestration is the definition of a business process as a composition of elementary services according to the BPM approach as defined above. In the contrary case where process execution logic is not centralized but implicitly contained in the messages exchanged between the elementary services themselves the composition is called *choreography*.

The BPM structure allows for separating the more stable elements in terms of the elementary functions from the more agile ones dealing with potentially fast changing business processes. According to the concept of service categorization the former correspond to function or inventory components/services. The later correspond to process components/services and their implementation can be based on either dedicated systems like e.g. order managements solutions allowing for the flexible configuration of workflows or can be based on dedicated BPM products. How to actually implement a BPM-aligned SOA therefore is an explicit design decision.

IV. RULES

A. Designing Domains

Business domains are defined by business requirements. Therefore business domains in the first place are found along the core processes of the value chain. Other domains may result from core business objects like partner or product that are of importance cross the business process. Thirdly, business domains of importance may be found looking at the single branches, e.g. product type. However, for business domains there is no rule of how to find the right ones for your architectural approach other then to try to understand the business of the enterprise and its strategy. Business domains result from questioning the stake holders.

This is different with application domains. Well cut application domains follow the business domains identified as important. For optimizing the domains design the following rule applies:

- ***Optimal Covering****: Design and arrange your application components within domains in a way, that according to the most important business domains the covering in the lay-out plan is optimal. The covering is optimal, if there are no avoidable multiply covered areas and if horizontal and vertical partitioning is as homogeneous as possible*

Unavoidableness and possibility in this regard result from given limitations like e.g. organizational constraints (accountability). Obviously, since shaping an application landscape on enterprise level always starts from some a given as-is status, applying the rule in reality is about planning a migration on a large scale.

B. Designing Components

Finding components is probably the oldest design challenge in software engineering. Nobody will be surprised, that the "golden rules of good design" like

- Information Hiding (as well as the related concepts of encapsulation and minimal coupling / maximal cohesion),
- Separation of Concerns
- Decoupling

also apply on the enterprise level finding the right application components. Taking these plus the advanced concepts presented in chapter 3 plus our findings from industrial practice we propose 7 concrete rules for designing application components in large scale quality architectures:

- ***Professional Categorization****: Define your application components according to appropriate professional categories*

On the highest level this demand for a separation of concerns corresponds with the demand for a unique mapping of application components to domains (see 4.1). But on lower levels it also calls for separating components inside one application domain according to different stakeholders involved, different speeds of change (see 3.4), different dependency on process etc.

- ***Service Category Orientation****: Define your application components to be of exactly one service category*

After what was said in Chapter 3, this rule is obvious. We really vote for taking service categories serious, since mixtures here have often proven to be time and cost intensive to maintain and later replace.

- ***Dependencies according to Service Categories****: Allow only for dependencies from components to components on equal or lower level of service category*

Dependencies amongst components can be of type "knows", "calls", or "gets data from". For all these we demand: components of type interaction only depend on components of type interaction, process, function or inventory, process only on process, function or inventory, function only on function or inventory and inventory only on inventory

- ***No Cyclic Dependencies****: Avoid cyclic dependencies amongst application components. Use (if appropriate) merging, separation or call-back*

Cyclic dependencies cause problems in development, testing, maintenance and replacement.

- ***Minimal Coupling / Maximal Cohesion****: Design your components for minimal coupling and maximal cohesion*

This is well known from software design and rightfully demanded on the large scale, too.

- ***Data Sovereignty****: Access to all business objects data is allowed through components of type inventory only. The areas of business objects data to be handled by different components are to be disjoint*

The same argument counts as with cyclic dependencies.

- ***Manageable Size****: Domains should have a medium-size number of components (not too many, not too little) and all components of a domains should have approx. equal size (e.g. in terms of lines of code, use case points etc.)*

This is a well known principle supporting manageability of the architecture.

C. Designing Services

Designing the components is one side of the coin – designing the services is the other. But before we can define the rules for the later, we need to add a little enhancement to the definitions from Chapter 2 concerning types of application service-operations. Operations can be classified as follows:

- *Elementary*: These operations yield simple basic functionality.
- *Compound*: These operations are implemented by using multiple elementary operations and provide added value services.
- *Orchestrateable*: These operations are designed to be a basis for an orchestration as defined in chapter 3. Usually these are compound.

Therewith we define the following 7 rules for the design of service-operations:

- ***Service Category Orientation***: *Every service-operation should be of exactly one service category*

Same argument as for components.

- ***Implementation Neutrality****: Service-operations should not export any implementation details*

This is the obvious demand for encapsulation, e.g. technical keys should never be part of any operation signature.

- ***Normality*** *(being complete and free of redundancy): Elementary service-operations should be normal*

For a detailed discourse see [21].

- ***Coarse-Granularity****: Orchestrateable service-operations should be coarse-granular*

Few operations doing much are better then many operations doing little in the context of loose coupling.

- ***Idempotency****: Orchestrateable service-operations should be idempotent*

Idempotent operations can be called multiple times if necessary not changing the result. This is necessary in the context of decoupling.

- ***Context-Insensitivity****: Service-operations of type function and inventory should not have any knowledge about the context in which they are called and therefore should not make any assumptions about it*

This is another necessary condition for decoupling.

- ***Constraint on Transactionality****: Transactional behavior (guaranteeing atomicity, consistency, isolation, and durability) is only allowed for operations of category inventory or function. All other operations are to be designed non-transactional with according compensation operations*

And this is yet another necessary condition for decoupling.

All presented information about rules for designing services are summarized in Fig. 9.

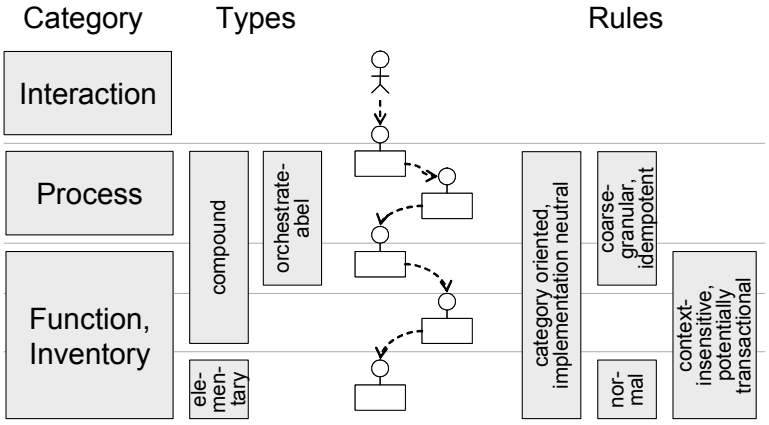


Fig. 9 Rules for Designing Services

D. Designing Couplings

Finally, the rules for designing coupling will be presented. We order these in two groups. The first group contains the general rules of coupling.

- ***Loose Coupling***: *Couple your application components as loose as possible and as tight as necessary*

This is the way to achieve maximum flexibility.

- ***Constraint on Tight Coupling***: *The tightest form of coupling in terms of synchronous intra-process communication is only allowed for operations of category inventory or function*

The second group finally contains rules with respect to the integration levels. These are self-explanatory.

- ***Presentation Integration***: *Choose presentation integration, if user interfaces to be integrated already exist and integration logic is restricted to dialogue control and value passing*

- ***Service Integration***: *Choose service integration, if applications to be integrated with substantial business-logic already exist and quick implementation of new processes and products is strategic*

- ***Data Integration***: *Choose data integration, if data redundancy already exists or is planned for performance reasons and differences between logical and physical data model are small*

V. Prospect

In this contribution we have outlined some of the core elements of what we aim at with "Quasar Enterprise" – a framework for Quality Architecture on the enterprise level. It was shown that SOA is indeed a promising concept, but also that there is more to Quality Architecture than SOA. Most prominently we have proposed a draft set of concrete rules to design a SOA to meet usual quality requirements.

Future research regarding the presented elements of "Quasar Enterprise" will be on further consolidating the set of rules by extending the set of concrete sd&m projects and experiences to validate the rules against. Also we will elaborate more on the "traceability" between rules or design decisions and the actual quality attributed listed e.g. in II B.

Many other elements that "Quasar Enterprise" will be about were not mentioned at all like the idea of *reference architectures* for specific parts of the application architecture (many of them already existing at sd&m, see e.g. [22]) or the concept of *embedding* as a mechanism for integrating legacy systems and COTS application components into a SOA.

Nonetheless we hope that this contribution will lead to the sought-after scientific discourse within the community.

References

[1] Denert, E., Siedersleben, J.: Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum 4/2000, pp 247-257

[2] Siedersleben, J. (Hrsg.): Quasar: Die sd&m Standardarchitektur. Teile 1 und 2, 2. Auflage. sd&m Research, 2003

[3] Siedersleben, J.: Moderne Software-Architektur – umsichtig planen, robust bauen mit Quasar. dpunkt Verlag. 2004

[4] Bieberstein, N., Bose, S., Fiammante, M., Jones, K., Shah, R.: Service-Oriented Architecture (SOA) Compass: Business Value, Planning, and Enterprise Roadmap. IBM Press 2006

[5] Erl, T.: Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services. Prentice Hall PTR. April 2004

[6] Keller, W.: Enterprise Application Integration. Erfahrungen aus der Praxis. dpunkt-Verlag 2002

[7] Krafzig, D., Banke, K., Slama, D.: Enterprise SOA : Service-Oriented Architecture Best Practices. The Coad Series. Prentice Hall PTR. November 2004

[8] Woods, D.: Enterprise Services Architecture. Galileo Press. Bonn, 2004

[9] Richter, J.-P.: Wann liefert eine serviceorientierte Architektur echten Nutzen? Proceedings Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005, Essen, Page 231-242

[10] Richter, J.-P., Haller, H., Schrey, P.: Aktuelles Schlagwort Serviceorientierte Architektur. Informatik-Spektrum Band 28, Heft 5, S. 413 – 416. Springer-Verlag, 2005

[11] Richter, J.-P., George, T., Gugel, T., Heimann, T., Lange, H. Möllers, T.: Technology Guide: Serviceorientierte Architekturen. sd&m, 2005

[12] Hess, A., Humm, B., Voss, M.: Quasar Enterprise – Serviceorientierte Architektur konkret. White Paper, sd&m Research, 2006

[13] Zachman, J.A.: The Zachman Framework for Enterprise Architecture – A Primer for Enterprise Engineering and Manufacturing. Zachman International, 2003

[14] The Open Group: TOGAF – The Open Group Architecture Framework. www.opengroup.org/togaf/

[15] Capgemeini: IAF – Integrated Architecture Framework. www.capgemini.com

[16] Szyperski, C.: Component Software. Addison Wesley 2002

[17] Reussner, R., Hasselbring, W.: Handbuch der Software-Architektur. dpunkt Verlag. 2006

[18] SAP: Netweaver. www.sap.com/solutions/netweaver

[19] Noel, J.: BPM and SOA: Better together. White Paper, IBM, 2005

[20] Rother, T.: Der optimale Weg zur Geschäftsagilität: SOA oder BPM. White Paper, Software AG, 2005

[21] Humm, B., Juwig, O.: Eine Normalform für Services. Proceedings of Software Engineering 2006. GI Edition Lecture Notes in Informatics (LNI) P-79. Gesellschaft für Informatik, 2006

[22] Haft, M., Humm, B., Siedersleben, J.: The architect's dilemma – will reference architectures help? In: R. Reussner et al. (Eds.): Quality of Software Architectures and Software Quality (QoSA-SOQUA 2005), Lecture Notes in Computer Science 3712, pp. 106 – 122, 2005. Springer-Verlag, 2005

An Approach to Resolving Contradictions in Software Architecture Design

Daniel Kluender
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
Email: kluender@informatik.rwth-aachen.de

***Abstract*— A key element to designing architectures of good quality is the systematic handling of contradicting quality requirements and the structuring principles that support them. The *theory of inventive problem solving* (TRIZ) by Altshuller offers tools that can be used to define such a systematic way. This paper describes the idea and preliminary results of using *inventive principles* and the *contradiction matrix* for the resolution of contradictions in the design of software architectures. By rearchitecting a flight simulation system these tools are analyzed and their further development is proposed.**

I. Introduction

It has long been recognized that a system's software architecture has a major impact on the nonfunctional properties of that system like dependability, performance or modifiability [20]. Designing software architectures of good quality is therefore central to software engineering as is the evaluation of architecture quality.

Figure 1 shows the process of requirements analysis according to [2]. The non-functional requirements represent the business goals associated with the system-to-be, the functional requirements are technically oriented. During requirements analysis the functional specification is written down and the driving qualities are identified. Driving qualities represent the hard to implement but yet most important stakeholder interests in a product. Because of their important impact on the architecture the driving qualities are also called architectural drivers. Architecture design can be seen as an optimization problem with the driving qualities being the optimization criteria and the functional specification being the optimization constraints.

Structuring principles which support certain qualities help the architect in finding the optimal architecture. These structuring principles can be architectural tactics or styles [2] like information hiding or architectural patterns [6] like a client-server architecture. Most structuring principles affect several qualities, either enabling or inhibiting them, e. g. information hiding supports the maintainability of a system but is impairing its performance. While the architect can choose from a set of well documented principles (see e. g. the work of Booch on a handbook of software architecture [3]) the merging and consolidation of different principles is by and large still an ad hoc and largely unsystematic process to date.

Conflicting quality requirements like performance and maintainability or conflicting structuring principles are compounding the design of a system's software architecture. The resolution of these conflicts relies heavily on the architect's experience and knowledge of the structuring principles. A software architecture represents the tradeoffs between the conflicting qualities that are acceptable for all stakeholders.

The author believes that the key to designing architectures of good quality is the systematic handling of contradictions between quality requirements or their according structuring principles ideally resulting in the elimination or resolution of the conflict. The *theory of inventive problem solving* (TRIZ) by Altshuller [1] offers tools that can be used to define such a systematic way. This paper describes the idea and preliminary results of using the TRIZ tools *inventive principles* and *contradiction matrix* for the resolution of contradictions in the design of software architectures. The rest of the paper is structured as follows: section II gives a short introduction into the origin of TRIZ and its possible use in software engineering. In sections III and IV we analyze the applicability of two tools offered by TRIZ to architecture design. These tools are used to rearchitect a flight simulation system as an example in section V. Section VI concludes the paper and gives an outlook to future work.

II. TRIZ

TRIZ is the Russian acronym for a term that is commonly translated as *theory of inventive problem solving* and has been developed by Altshuller et al. since 1946. By analyzing over 200,000 patents they found that [1]:

- Innovations emerge from the application of a relatively small set of strategies, so called *inventive principles*.
- The strongest solutions actively seek out and destroy the conflicts or *contradictions* most design practices assume to be fundamental.
- They also transform unwanted or harmful elements of a system into useful resources.
- Technology evolution trends are predictable.

Based on these foundations TRIZ offers a set of tools that can be used to solve problems by applying the strategies employed for successful inventions. These tools include e. g. *Substance-Field Analysis*, *Subversion Analysis*, *Trends of Evolution* or the *Contradiction Matrix*. The basic idea is to use generalized solutions from one domain to provide inspiration for other domains. This makes TRIZ a very general theory that can be

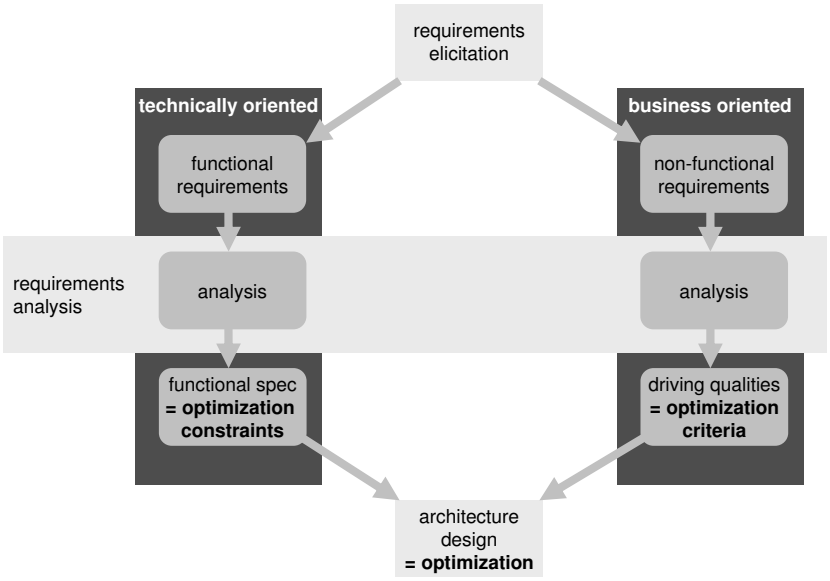


Fig. 1. Two paths of requirements analysis according to [2]

applied in many areas. This paper examines the application of the TRIZ tool *Contradiciton Matrix* for software architecture design.

The application of TRIZ to software engineering is a relatively new field, hence publications are only few. Rea discusses analogies to the inventive principles in software [16], [17] and uses them to obtain several patents [19]. These analogies are extended by Fulbright [7] and Tillaart [21]. Most of them are not directly applicable to software architecture and will be further discussed in section III. Nakagawa is reviewing topics in software engineering such as structured programming to reason about them using TRIZ [13], Rea reviews concurrency [15]. Rawlinson discusses the application of contradictions between speed, reliability, energy and complexity [14] but does not go into the implications for software architecture. A more general review of the application of TRIZ to software can be found in [18]. Hartmann et al. emphasize the practicability of TRIZ for software architecture [8], Muller classifies TRIZ as a possible architecting method [12]. Figure 2 shows his classification of architecting methods with the more general methods placed to the right.

Mann's summary [10] gives a short insight into his upcoming book [11]. He analyzed 40,000 patents in software and developed a newly tailored contradiction matrix, slightly modified inventive principles, trends of evolution and other TRIZ tools. Since the paper just gives a short overview, the book that is not available at the time of this writing needs to be awaited for a more detailed discussion.

III. Inventive Principles

The aforementioned observation that innovations emerge from the application of a relatively small set of strategies lead TRIZ researchers to the formulation of 40 innovative principles.These are the generalized descriptions of 40 solution strategies like e. g. *nesting*, *counterweight* or *changing the colour* that were identified by analyzing patents. Despite their generality not all of the 40 innovative principles are directly applicable to software architecture design, some are apparent mismatches. Nevertheless these principles subsume solutions to conflicts and contradictions that are successfully applied in other domains, hence a mapping into software architecture terms seems promising.

Since TRIZ was developed in hardware-based technology fields this mapping is not a straight forward task. Others have tried to find analogies [7], [16], [17], [21] but these analogies are concerned with multiple phases of software engineering. Most of them are close to implementation issues of specialized domains and as such not usable for application to software architecture. Mann has analyzed software patents [10], [11] but as said before his results have not yet been published. Some ideas can also be found in his paper about buildings' architectures [9] .

Formulating a new set of innovative principles for software architecture in the same way it was done during the development of the TRIZ theory is hard because these principles originate from the analysis of patents but there are few patents on software architectures. In contrast to a derivation of software architecting principles from patents they can be formulated using patterns. Architectural patterns are a good source for principle mining because they are generalized solutions for frequently occurring problems. As such they contain the heuristics of successfull solutions. The analysis of correspondences between inventive principles and patterns shows the following:

1) Inventive principles are more general than patterns and as such often comprise several patterns. For example the principle *segmentation* is a generalized description of patterns like the *layered architecture pattern* or the principle *copying* comprises several redundancy pat-

software architecting methods:	multidisciplinary systems architecting methods:	methods also addressing process and organization:	generic methods:
• SAAM (SEI) • ATAM • 4+1 (Kruchten) • 4 views (Soni) • ZIFA (Zachman) • 9126 (ISO) • VAP (Bredemeyer)	• SE practices (INCOSE) • 1471 (IEEE) • CAFCR (Muller)	• Systems Engineering (Martin) •Systems Architecting (Maier, Rechtin)	• TRIZ (Altshuller) • GST (Hitchins, Heylighen)

Fig. 2. Classification of architecting methods according to [12]

terns. Hence inventive principles are no replacement for patterns but combined with the contradiction matrix can serve as a navigation aid for selecting patterns or finding new ones.

2) It's common in TRIZ to cluster the inventive principles into four classes: contradiction resolution in space, time, structure and material. Correspondences between architectural patterns and principles can mainly be found in the first three.
3) For some principles we can not find any correspondences in architectural patterns. For example *accelerated oxidation* describes the principle to replace common air with oxygen-enriched air. Most of these principles are part of the material class.
4) Correspondences between the 40 inventive principles and architectural patterns can be found for 29 principles (e. g. *garbage collection* as an example of the principle *rejecting and regenerating parts*) and no correspondences for 11 principles (e. g. *changing the state of aggregation*). The complete list of correspondences can be retrieved from the author's website[1].

IV. Contradiction Matrix

The inventive principles can be used stand-alone to search for solutions or in form of the contradiction matrix for a more directed search in solution space. This matrix is another TRIZ tool that allows users to detect side-effects the 40 innovative principles can have 39 various technical parameters like e. g. repairability, reliability or temperature. The 39 * 39 matrix is a table that contains the three or four most often used inventive principles for solving contradictions between the 39 technical parameters. From an architect's point of view these parameters can be seen as the quality attributes of the system to be designed. The viewpoint of TRIZ is that there are ways of eliminating or resolving contradictions between the technical parameters and that designers should actively look for them. Though a software architect might rather think of a trade-off between contradicting quality attributes the elimination of these contradictions are considered to be a valuable ideal.

[1]`http://www-i11.informatik.rwth-aachen.de/kluender.html`

Table I shows an extract from the contradiction matrix since the complete matrix would use too much space. Each cell lists the identification numbers of the inventive principles that are most often used to resolve a conflict or contradiction between the technical parameters of the cell's row and column. It shows e. g. that inventive principle 2 (Extraction: Separate or extract an interfering part or property from a technical system or single out the only necessary part or property.) is the most often used principle to resolve the contradiction between manufacturability and convenience of use.

For constructing a contradiction the positive and negative effects of a function or action are described in terms of two of the 39 technical parameters of the contradiction matrix. For resolving this contradiction the inventive principles noted in the corresponding cell of the matrix are applied. Therefore the matrix's rows describe the positively affected parameters and the columns the negatively affected ones. Of course each contradiction has a corresponding inverse contradiction that is constructed by describing the effects of the inverse function or action. Since the matrix is not completely equal to its transpose the creation of inverse contradictions can sometimes reveal more inventive principles. Contradictions including more than two parameters can always be segmented into several contradictions including only two parameters also described as binary contradictions [1].

As a very simple example regard a static physical object that is to be extended without increasing its weight. The technical parameter to be improved is the *length of a stationary object* but that would also lead to impairing the technical parameter weight of a stationary object. In the according cell of the contradiction matrix the following innovative principles are suggested: composite materials, replacement of a mechanical system, transforming of physical or chemical states, use pneumatic or hydraulic systems. It should be mentioned here that some cells of the contradiction matrix are empty, simply because no solution for this conflict can be found in the patents that were analyzed to construct this matrix.

As said before quality attributes can be seen as translations of the technical parameters when applying the contradiction matrix to software architecture. Some of these translations are straight forward like reliability, availability (durability of an object), adaptability or maintainability (repair friendliness).

TABLE I
AN EXTRACT FROM THE CONTRADICTION MATRIX

	manufacturability	**convenience of use**	**repairability**	**adaptability**
manufacturability		2, 5, 13, 16	35, 1, 11, 9	2, 13, 15
convenience of use	2, 5, 12		12, 26, 1, 32	15, 34, 1, 16
repairability	1, 35, 11, 10	1, 12, 26, 15		7, 1, 4, 16
adaptability	1, 13, 31	15, 34, 1, 16	1, 16, 7, 4	

Others can not be easily translated into software architecture terms like mass, length, area or volume of a moving or immobile object. Although not all inventive principles and all technical parameters of the contradiction matrix can yet be translated into software architecture terms the remaining extract can be useful for software architecting because it offers a navigation help when searching for an architectural solution to contradictions in contrast to common trial-and-error methods that are solely based on the architect's experience.

In the software design process the contradiction matrix can be of help if two driving qualities or their supporting architectural principles contradict each other. The architect looks up the corresponding technical parameters in the matrix and tries to apply the listed innovative principles and architectural patterns that belong to them. Just as well the matrix can help to choose an architectural pattern to support a driving quality that has no contradicting quality requirement by supporting information on the effect of a pattern on other qualities. That way the matrix allows a systematic approach to choosing patterns and resolving contradictions between quality requirements which can easily be integrated into mature architecture design methods like e. g. attribute driven design [5].

V. EXAMPLE: FLIGHT SIMULATOR

To analyze the applicability of the two TRIZ tools and give an example of their usage the well documented requirements and architecture of the flight simulation system introduced in chapter eight of [2] is used. Rearchitecting the existing system allows to examine whether the inventive principles and contradiction matrix can help designing the system's architecture. The question of interest is whether the general principles for resolving contradictions found in other areas can also be applied to software architecture.

During requirements elicitation and analysis the driving qualities are identified. Afterwards their corresponding technical parameters from the contradiction matrix are denoted:

- the system's performance corresponds to technical parameter 9: speed
- modifiability to accommodate changes in requirements and scalability of function correspond to technical parameter 35: adaptability
- integrability corresponds to technical parameter 32: manufacturability
- testability corresponds to technical parameter 37: complexity of control and measuring

In general some of these architectural drivers contradict each other. For the flight simulator improving modifiability could impair the system's performance. To resolve this contradiction the contradiction matrix suggests using the inventive principles *dynamicity*, *prior action* or *copying*. In fact the architecture design suggested in [2] uses a partitioning that maintains a close correspondence between the aircraft partitions and the simulator virtually copying parts of the aircraft.

Other inventive principles that can be found in the suggested architecture include *segmentation*, *extraction*, *mediator* and *nesting*. The example shows that the inventive principles are no replacement for architectural tactics or patterns but rather an extension that helps selecting merging and balancing them. As said before some parts of TRIZ seem to make no sense for software architecture, e. g. the suggested usage of the inventive principle *changing the state of aggregation*.

VI. CONCLUSION AND FUTURE WORK

Contradicting quality requirements and the merging of their supporting architectural strategies are core problems in software architecture design and make it a task that is heavily dependent on the architect's experience and knowledge. Using the TRIZ tools inventive principles and contradiction matrix can help directing the search in the solution space into a heuristically promising direction. Hence these tools can be seen as an extension to architectural tactics and patterns. This paper displays the author's approach of finding correspondences between inventive principles and architecture patterns on the one hand and technical parameters and quality attributes on the other. Although not all 40 principles and 39 parameters have a corresponding pattern or attribute the remaining can be useful in architecture design. In fact some of the general principles for resolving contradictions found in other areas can also be applied to software architecture.

Future work will concentrate on deepening the understanding of inventive principles and the contradiction matrix and their application to software architecture. Questions to be answered are whether the matrix should be modified into a software architecture specific matrix or whether all the experience and knowledge of other fields can be used for designing software architectures. The analyzed TRIZ tools could also be used for other fields than architecture design as e. g. scenario based architecture evaluation [4]. It might also be promising to try to apply other TRIZ tools like *Substance-Field Analysis*, *Subversion Analysis* or *Trends of Evolution* to software architecture. Finally the insight of using TRIZ for software might be valuable for TRIZ theory itself.

REFERENCES

[1] G. Altshuller, H. Altov, and L. Shulyak. *And Suddenly the Inventor Appeared: Triz, the Theory of Inventive Problem Solving*. Technical Innovation Ctr., 1996.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Professional, 2. edition, 2003.

[3] Grady Booch. On architecture. *IEEE Software*, 23(2), March / April 2006.

[4] Liliana Dobrica and Eila Niemel. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, Juli 2002.

[5] L. Bass, F. Bachmann, and M. Klein. Quality attribute design primitives and the attribute driven design method. *In Proceedings of the 4th International Conference on Product Family Engineering*, Springer Verlag, Berlin, Germany, 2002.

[6] Bruce Powel Douglass. *Real-Time Design Patterns*. Addison-Wesley, 2003.

[7] Ron Fulbright. TRIZ and software fini. *TRIZ Journal*, August 2004.

[8] Herman Hartmann, Ad Vermeulen, and Martine van Beers. Application of TRIZ in software development. *TRIZ Journal*, September 2004.

[9] Darrell Mann. 40 inventive (architecture) principles with examples. *TRIZ Journal*, July 2001.

[10] Darrell Mann. TRIZ for software? *TRIZ Journal*, October 2004.

[11] Darrell Mann. *TRIZ for Software Engineers*. IFR Press, to appear soon.

[12] Gerrit Muller. *CAFCR: A Multi-view Method for Embedded Systems Architecting*. PhD thesis, Technische Universiteit Delft, 2004.

[13] Toru Nakagawa. Software engineering and TRIZ - structured programming reviewed with TRIZ. In *Proceedings of TRIZCON*. Altshuller Institute, April 2005.

[14] Graham Rawlinson. TRIZ and software. In *TRIZCON*. Altshuller Institute, March 2001.

[15] Kevin C. Rea. Using TRIZ in computer science - concurrency. *TRIZ Journal*, August 1999.

[16] Kevin C. Rea. TRIZ and software - 40 principle analogies part1. *TRIZ Journal*, September 2001.

[17] Kevin C. Rea. TRIZ and software - 40 principle analogies part 2. *TRIZ Journal*, November 2001.

[18] Kevin C. Rea. Applying TRIZ to software problems. In *Proceedings of TRIZCON*. The Altshuller Institute, May 2002.

[19] Kevin C. Rea. TRIZ for software: Using the inventive principles. *TRIZ Journal*, January 2005.

[20] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[21] Rob van den Tillaart. TRIZ and software - 40 principle analogies, a sequel. *TRIZ Journal*, January 2006.

Quality Attribute Variability within a Software Product Family Architecture

Varvana Myllärniemi, and Tomi Männistö and Mikko Raatikainen
Helsinki University of Technology
Software Business and Engineering Institute
P.O. Box 9210, 02015 TKK, Finland
Email: {varvana.myllarniemi, tomi.mannisto, mikko.raatikainen}@tkk.fi

***Abstract*— Software variability has received research attention. However, only a few studies address the issue of quality attribute variability within software product families. This paper describes the ongoing research on quality attribute variability within a software product family architecture. We motivate the research topic by identifying three different situations in which quality attributes may vary. These situations can be characterised by the trade-offs they represent. We briefly discuss a number of existing methods that can be used for realising quality attribute variability within the scope of one architecture, i.e., as a software product family. The ongoing research aims to deepen this understanding.**

I. Introduction

Due to the growing diversity of customer needs, software often has to support variability. *Variability* is the ability of software to be efficiently extended, changed, customised or configured for use in a particular context [1]. Techniques such as *software product families* (also known as software product lines) have emerged to aid efficient variability management and realisation [2], [3]. Variability in software product families has been studied to a considerable extent. However, most studies address only functional variability, or study variability only at a very general level. Some studies identify that also quality attributes can vary [4]–[6], but to the best of our knowledge, an extensive treatment on quality attribute variability from product family point of view is missing.

Quality attributes, such as performance, security or maintainability, are often architectural in nature [7]. That is, the architecture is critical to the realisation of these quality attributes, and they should be designed in and evaluated at the architectural level [7]. This is in contrast to functionality, which is largely nonarchitectural in nature: any number of possible structures can be conceived to implement any given functionality [7]. Therefore, quality attribute variability may in some cases have architectural ramifications that differ from those of functional variability. For example, often the realisation of a quality attribute cannot be localised into one part of the architecture. In the worst case, a change in such a quality attribute may require architecture-wide changes. In contrast, functionality is often more easily decomposed into a certain subsystem, and thus it is easier to be varied.

As said, the topic of quality attribute variability in software product families has not received much research attention. However, there are several existing software products that offer varying levels of quality attributes. Since a common theory as well as generic methods to address the issue are missing, the solutions in the existing cases tend to be *ad hoc* or very domain-specific. Therefore, quality attribute variability should be integrated to be a part of the systematic variability management of software product families, enabling to take an advantage of the phenomenon to its fullest extent.

This paper describes the ongoing research on quality attribute variability. We motivate the research topic by identifying three different trade-off situations that can be solved with varying quality attributes. For each situation, an example from the Finnish industry is given. We also briefly discuss how quality attribute variability can be realised within the scope of a varying architecture, i.e. as a software product family. For this purpose, we discuss a number of existing methods in terms of their feasibility in supporting varying quality attributes. However, the ongoing research aims to deepen this understanding.

The scope of this research is limited to software variability. That is, we are interested in software techniques, in particular architectural ones, that aim to achieve varying quality attributes. Although hardware often plays a role in achieving quality attributes, we focus on software architecture. Further, we limit our scope to situations in which quality attribute variability can be implemented through a varying architecture, i.e., as a software product family.

The rest of the paper is organised as follows. Section II characterises the research topic by identifying three situations in which quality attributes may vary. Section III shortly addresses existing methods for realisation of quality attribute variability. Section IV discusses the research findings. Finally, Section V draws conclusions and identifies future work.

II. Quality attribute variability characterisation

A. Motivation for quality attribute variability

Quality attribute requirements are often specified through minimum acceptable bounds. For a certain quality attribute, an acceptable realised value can be better than the desired value specified in the requirements. As an example, if user U requires that availability is at least 0.99, she can accept a software with availability figure of 0.995.

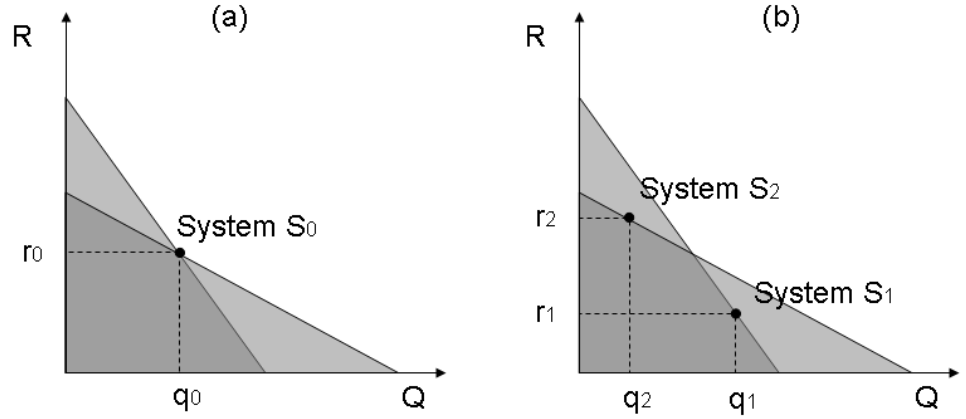


Fig. 1. Figure exemplifies how two conflicting quality attributes, Q and R, can be resolved into two variants. Instead of developing system S_0 as a compromise between Q and R, develop systems S_1 and S_2 to optimise Q and R, respectively. (Figure (a) has been modified from [8].)

Given this frame of reference, the need for varying quality attributes might not be self-evident. As an example, consider the situation in which user U_1 requires availability to be at least 0.99, and user U_2 requires at least 0.995. What is the motivation for constructing two systems S_1 and S_2 with availability figures 0.99 and 0.995 respectively, since system S_2 satisfies both users' needs?

This characteristic of quality requirements is different from functional requirements. Although feature f_2 is more extensive than feature f_1, in general f_2 cannot be said to be better. If the user regards feature f_1 to match her needs, she may be annoyed to have a system with feature f_2 instead. As an example, an advanced image editor may be more trouble than worth to a user who only wants to view her holiday photos.

In order to see the motivation for varying quality attributes, it is worth remembering that quality attributes interact with each other, often affecting each other negatively [9], p. *ix*. Typically, these trade-off situations are resolved by finding a compromise between conflicting quality attributes (see Fig. 1a and [8]). However, quality attribute variants are an alternative way of solving the dilemma. Instead of developing one system as a compromise of conflicting quality attributes (Fig. 1a), develop a set of systems that optimise one quality on behalf of the another (Fig. 1b).

In many cases, these variants can be realised effectively as a software product family. However, in some cases the variants can be so apart from each other that they must be built as completely separate products. Hence, the architecture does not need to support quality attribute variability.

B. Quality attribute variability as trade-off solutions

In order to elaborate the situations in which quality attribute variability may provide a solution, we propose a classification of three different situations. These situations can be characterised by the trade-off they represent. For each type, an example case is given; the cases are drawn from the Finnish industry. Table I summarises the types and gives definitions for each trade-off situation.

a) Trade-off between quality attributes: Many quality attributes are in conflict with each other: security is often in conflict with usability and performance, performance is often in conflict with availability and maintainability. Instead of developing a system as a compromise between security and performance, develop some variants to favor security over performance, and vice versa. Depending on the user needs, the choice between these variants can be made.

Case: F-Secure is a company developing various products related to software security. One discontinued product, SSH client, provides a remote shell client with encrypted communication. During the installation, the user can choose the length of the encryption key. Selecting a long encryption key provides better security but hinders performance, and vice versa. Thus the user can select a combination of security and performance to suit her needs.

b) Trade-off between quality attribute and business quality: Quality attributes are often in conflict with business qualities, such as cost, time-to-market and projected lifetime [7]. Varying levels of quality can be differentiated with varying price: a product with higher security costs more. Further, projected lifetime can be varied with maintainability: a long-lived variant is designed to be easier to maintain than a short-lived variant.

Case: Kone Elevators produces elevators for private and public use. A large part of revenue comes from service contracts. Kone Elevators provides three service levels with varying availability rates: the most expensive service contract guarantees availability figure of 0.995, while the second level contract guarantees availability of 0.99. Although in this case availability can be provided by means that are not directly related to software (such as maintenance rate), higher availability can be backed with fault-resistant software also.

c) Trade-off between quality attribute and varying external constraint: Finally, there may be an external varying constraint that affects quality attributes. The constraint can be e.g. hardware-related or market-related. Some countries prohibit the use of too strong encryption mechanisms. In such

TABLE I

TYPES OF TRADE-OFF SITUATIONS THAT CAN BE SOLVED BY QUALITY ATTRIBUTE VARIABILITY. THE ABOVE DEFINITIONS ASSUME THAT THE VALUES OF Q, R AND B CAN BE REPRESENTED WITH A COMPARABLE METRIC, I.E., COMPARISONS ARE MEANINGFUL. IF A NUMERICAL METRIC IS UNAVAILABLE, THE METRIC CAN BE A DISCRETE SET OF TYPE $\{$*low*, *medium*, *high*$\}$. THE ABOVE DEFINITIONS CAN ALSO BE EXTENDED, SEE SECTION II-C.

Trade-off type	**Quality attribute variability solution**
Trade-off between quality attributes Q and R	Instead of system S_0 as a trade-off with $Q = q_0, R = r_0$, provide system S_1 with $Q = q_1, R = r_1$ and system S_2 with $Q = q_2, R = r_2$ such that $q_2 < q_0 < q_1$ and $r_1 < r_0 < r_2$.
Trade-off between quality attribute Q and business quality B	Instead of system S_0 as a trade-off with $Q = q_0$ and $B = b_0$, provide system S_1 with $Q = q_1, B = b_1$ and system S_2 with $Q = q_2, B = b_2$ such that $q_2 < q_0 < q_1$ and $b_1 < b_0 < b_2$.
Trade-off between quality attribute Q and varying constraint C	If variants $c_1, \ldots, c_n$ of constraint C limit quality Q such that $\forall c_i : Q \leq q_i^{max}, i = 1 \ldots n$, provide systems $S_1, \ldots, S_n$ such that $\forall S_i : Q = q_i \leq q_i^{max}, i = 1 \ldots n$.

a case, different security levels can be targeted for different legislation. Further, available hardware and network resources may affect performance. As an example, a mobile device can increase its graphics processing throughput when a higher bandwidth network is available.

Case: Fathammer is a small company that produces 3D games for various mobile devices. Due to the enormous differences in the target mobile devices, Fathammer must compensate varying hardware by varying game performance and memory consumption. Since graphics form a major factor in the consumption of computing and memory resources, it is easy to tune the performance and memory consumption level by, e.g., changing the number of drawn polygons, the drawing algorithms, materials and applied textures. For further information, see [10].

C. Extending the trade-off dimensions

Sections II-A and II-B treated different trade-off types independently from each other. In practice, these trade-off types are seldom identified as presented in the definitions. Instead, there may be several simultaneously conflicting factors, each of which may be a quality attribute, a business quality, or a constraint. As an example, Fig. 1b assumes that two conflicting qualities can be optimised independently from cost. In practice, the solutions are not searched from the boundary of technically possible solutions (crossing lines in Fig. 1b), since these solutions tend to be very costly. In other words, the conflict shown in Fig. 1 was two-dimensional, whereas in general there are $N \geq 2$ conflicting factors, which constraint an N-dimensional solution space.

Also the number of variants can be extended. Most of the definitions in Table I resolved the conflict situation with two variants. In general terms, the number of variants can be any $M \geq 2$. As an example, F-Secure SSH client provides several key lengths to choose from. If parameterisation is used, the number of variants can be very large, or even practically infinite. As an example, if there was some parameter to tune quality Q, the variants could cover the range $q \in [q_2, q_1]$ (see Fig. 1b). An example of such a setting could be an MP3 recorder with the possibility to select the bit rate as a compromise of sound quality and file size.

Finally, besides the number of variants, also the binding time of variants as well as the possibility to rebind needs to be considered. If the balance between conflicting factors changes dynamically, runtime reconfiguration of quality attributes may a viable option. An example situation could involve a user whose preferences over conflicting quality attributes change dynamically (first trade-off type in Section II-B). In a similar fashion, there may be a dynamically varying external constraint, such as network bandwith, to which the software can adapt (third trade-off type in Section II-B).

III. QUALITY ATTRIBUTE VARIABILITY REALISATION

Section II characterised the situations in which quality attribute variability may be needed. After the need for variability has been identified, the variants must be realised within the scope of a software product family architecture. In this section, we briefly address some existing techniques in terms of their suitability for realising quality attribute variability.

A. Specifying varying quality attribute requirements

Before varying quality attributes can be realised, they need to be specified. In the most simplistic form, a varying quality attribute can be captured with a range of discrete values, such as $\{$*low*, *medium*, *high*$\}$. However, this alone does not suffice, since the meaning of such values is ambiguous. Quality requirements, like any good requirements, should be verifiable. Without verifiability, it is impossible to know whether the system actually meets its requirements.

In the other extreme, the quality requirement can be captured with a quantitative metric. If such a metric is available, the varying requirement can contain a set or range of values for the metric. However, many quality attributes do not have a commonly accepted metric, and are thus difficult to be specified quantitatively.

In addition to metric-based approach, there are other mechanisms for specifying quality requirements in a verifiable manner. Examples include scenarios [11] and misuse cases [12]. However, to the best of our knowledge, it has not been studied how these constructs can capture varying quality requirements.

Feature models [13], [14] have been proposed as a means of specifying varying requirements. In principle, feature models can capture quality attributes; the definitions for the term feature [3], [14] cover also non-functional properties. However, feature modelling does not give any guidance for specifying the quality attribute in a verifiable manner. Further, variability in feature models is specified through discrete features. It is unclear how feature models can capture qualities with a very large number of variants (see Section II-C).

Finally, the means of capturing varying quality requirements should take the dependencies between variants into account. For example, the situation depicted in Fig. 1b creates two dependencies between Q and R (the first dependency being $Q = q_1 \Leftrightarrow R = r_1$ and the second $Q = q_2 \Leftrightarrow R = r_2$).

B. Architecture design strategies for varying quality attributes

In general, there are several more or less codified strategies for achieving quality attributes in the architecture design. These include styles, patterns and various architectural mechanisms [7], [15]. There are some studies on varying design patterns [4], [6]. However, to a larger extent, there is not much research on how these strategies can be varied within the scope of one architecture.

The cases presented in Section II-B exemplify how quality attributes can vary within an architecture. It is interesting to note that varying quality requirements are achieved through varying functionality. In case of F-Secure, security is varied with encryption key length. For Fathammer, performance is varied by tuning the graphics. The possibility of transforming quality requirements to functional requirements has been identified for non-varying requirements (see e.g. [3]). It seems that this kind of approach is relatively convenient, since functional requirements are often easier to implement and verify. However, there is the risk of focusing too tightly on the functionality—using a long key does not necessarily guarantee that the system is secure.

C. Achieving quality attribute variants through evaluation

In addition to utilising traditional software architecture design strategies for achieving quality attributes, it is possible to construct a design method that utilises software architecture evaluation. In such a case, the architecture that satisfies a given quality attribute value is found, not through applying architectural strategies, but through evaluating the quality attributes of possible product variants. A prerequisite is that the evaluation method should be effective enough; in practice this requires automated evaluation.

An example of an approach that searches for suitable variants through subsystem evaluation is presented in [16]. The evaluation in [16] is done through evaluating property predictor functions attached to components. However, the authors of [16] do not explicitly address how to construct prediction functions, i.e., how to predict quality attributes from component properties. This is typically a non-trivial task.

However, there exists a wealth of proposed evaluation methods for this purpose. The problem of *predictable assembly* [17] is as follows: given a set of components, and component assembly, how can system-wide quality attribute value be evaluated? In most cases, predictable assembly models require that the quality of interest can be captured with a quantitative metric. Several approaches have been presented, e.g., for performance [18], resource consumption [19], and reliability [20]. However, some quality attributes are more difficult in the sense that their prediction requires more information [18]. Not surprisingly, existing prediction methods tend to concentrate on the easiest categories.

IV. Discussion

Section II characterised quality attribute variability by identifying three different trade-off situations in which quality attributes can vary. However, this characterisation as such is not sufficient for scoping the software product family. Many of the conflicts presented in the characterisation stem from the constraints of the architectural design decisions. Therefore, trade-off situations and consequently the suitable variants cannot be identified until enough design decisions have been made. Hence, software product family scoping must be done in parallel with the architecture design. We plan to study this topic further.

As can be noted from Section III-A, there are approaches for specifying quality attribute requirements that do not address variability explicitly. In a similar fashion, there are approaches for specifying varying requirements that do not address quality attributes. Although there are some studies that address both aspects (e.g. [21]), a thorough understanding of the applicability of existing methods is missing.

Section III-B shortly mentioned how quality attribute variability can be achieved with some well-known architectural strategies. Although some studies on the topic exist, we aim to deepen this understanding by identifying some archetypical ways in which quality attribute variability is reflected in the software architecture. The complexity of this relation affects the difficulty of achieving quality attribute variability: implementing a simple parameter or a functional component is relatively easy, while large-scale variability may be too costly to implement within a single software architecture.

Section III-C addressed the possibility of finding quality attribute variants through evaluating possible architectures. However, discussed example approach [16] evaluates architectures in a *brute force* fashion, i.e., one by one. The scalability risks involved in [16] are unfortunate, especially since the approach is used for dynamic reconfiguration. A more sophisticated and effective solution would be to use a logic-based inference engine to find the satisfying solutions. We are currently investigating on how to model varying security levels such that *smodels* [22] inference engine can be utilised for

finding satisfying solutions. With such a solution, the runtime evaluation of all possible solutions can be made effectively.

Besides the more family-centric approaches described in this paper, there are also a few research areas that address quality attribute variability mainly from the dynamic point of view.

The are plenty of studies on resource-constrained, QoS (Quality of Service) adaptive applications, such as multimedia applications [23], or distributed applications with real-time requirements [24]. Since the availability of resources fluctuates at runtime, the applications need to adapt their QoS in order to ensure satisfactory user experience. In the most simple case, this adaptation could mean sending less data; in a complex case it could require reconfiguration of the application architecture [24]. From the viewpoint of the characterisation presented in Section II-B, this kind of resource-adaptive QoS is a trade-off between quality attribute and a varying constraint.

There are also studies on context-aware and context-adaptive systems [25]. In some cases, context switches can act as the source of quality attribute variability. Depending on the situation, a context switch might change the balance of user preferences over conflicting quality attributes (first trade-off type in Section II-B), or alter some external constraint, such as available resources (third trade-off type in Section II-B).

V. Conclusions

This paper described the ongoing work on quality attribute variability. To motivate the research topic, we identified three different situations in which quality attributes may vary. These situations were characterised by trade-offs that can be solved with varying quality attributes. For each of these situations, an example case was given.

We also briefly discussed how quality attribute variability can be realised within a varying architecture, i.e. as a software product family. We discussed some existing methods and approaches in terms of their applicability for realising quality attribute variability.

For the rest of our research, we plan to study the following.

Firstly, we plan to study more closely the feasibility of quality attribute variability and its relation to software architecture design. We plan to further develop the characterisation presented in Section II by constructing an approach for selecting and designing a cost-effective set of quality variants. We plan to identify some archetypical ways in which quality attribute variability is reflected in the software architecture; this has an effect on the costs of producing quality attribute variants.

Secondly, we plan to develop an approach for modelling software product family architectures with varying security requirements. With such an approach, the product variants that satisfy desired security levels are found by evaluating possible product architectures. In order to make evaluation efficient, we are planning to use *smodels* [22] inference engine for this purpose.

Acknowledgment

The financial support of National Technology Agency of Finland (Tekes) is acknowledged.

References

[1] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxononomy of variability realization techniques," *Software—Practice and Experience*, vol. 35, no. 8, 2005.

[2] P. Clements and L. Northrop, *Software Product Lines—Practices and Patterns*. Addison-Wesley, 2001.

[3] J. Bosch, *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[4] S. Hallsteinsen, T. E. Fægri, and M. Syrstad, "Patterns in product family architecture design," in *Proc. of Workshop on Product Family Engineering*, 2003.

[5] G. Halmans and K. Pohl, "Communicating the variability of a software-product family to customers," *Software and Systems Modeling*, vol. 2, no. 1, 2003.

[6] M. Matinlassi, "Quality-driven software architecture model transformation," in *Proc. of Working IEEE/IFIP Conference on Software Architecture*, 2005.

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 1998.

[8] M. Barbacci, T. Longstaff, M. Klein, and C. Weinstock, "Quality attributes," Software Engineering Institute, Tech. Rep. CMU/SEI-95-TR-021, 1995.

[9] B. Boehm, J. Brown, H. Kasper, M. Lipow, G. Macleod, and M. Merrit, *Characteristics of Software Quality*. North-Holland Publishing Company, 1978.

[10] V. Myllärniemi, M. Raatikainen, and T. Männistö, "Inter-organisational approach in rapid software product family development—a case study," in *Proc. of International Conference on Software Reuse*, 2006.

[11] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures—Methods and Case Studies*. Addison-Wesley, 2002.

[12] I. Alexander, "Misuse cases: use cases with hostile intent," *IEEE Software*, vol. 20, no. 1, 2003.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, ADA 235785, 1990.

[14] K. Czarnecki and U. Eisenecker, *Generative Programming*. Addison-Wesley, 2000.

[15] L. Bass, M. Klein, and F. Bachmann, "Quality attribute design primitives," Software Engineering Institute, Tech. Rep. CMU/SEI-2000-TN-017, 2000.

[16] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, 2006.

[17] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, "Anatomy of a reseach project in predictable assembly," in *Proc. of 5th Workshop on Component-Based Software Engineering*, 2002.

[18] M. Larsson, "Predicting quality attributes in component-based software systems," Ph.D. dissertation, Mälardalen University, 2004.

[19] M. de Jonge, J. Muskens, and M. Chaudron, "Scenario-based prediction of run-time resource consumption in component-based software systems," in *Proc. of 6th Workshop on Component-Based Software Engineering*, 2003.

[20] R. Reussner, H. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, 2003.

[21] B. Gonzales-Baixauli, J. Prado Leite, and J. Mylopoulos, "Visual variability analysis for goal models," in *Proc. of International Requirements Engineering Conference*, 2004.

[22] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1–2, 2002.

[23] J. Jin and K. Nahrstedt, "QoS specification languages for distributed multimedia applications: A survey and taxonomy," *IEEE Multimedia*, vol. 11, no. 3, 2004.

[24] J. Ye, J. Loyall, R. Shapiro, S. Neema, S. Abdelwahed, N. Mahadevan, M. Koets, and D. Varner, "A model-based approach to desiging QoS adaptive applications," in *Proc. of Real-Time Systems Symposium (RTSS)*, 2004.

[25] R. M. Mayrhofer, "An architecture for context prediction," Ph.D. dissertation, Johannes Kepler Universität, 2004.