Ralf Reussner Johannes Mayer
Judith A. Stafford Sven Overhage
Steffen Becker Patrick J. Schroeder (Eds.)

Quality of Software Architectures and Software Quality

First International Conference on the
Quality of Software Architectures, QoSA 2005
and Second International Workshop on
Software Quality, SOQUA 2005
Erfurt, Germany, September 20-22, 2005
Proceedings

Springer

Volume Editors

Ralf Reussner
Steffen Becker
University of Oldenburg, Department of Computing Science
Escherweg 2, 26121 Oldenburg, Germany
E-mail: {reussner, becker}@informatik.uni-oldenburg.de

Johannes Mayer
University of Ulm, Department of Applied Information Processing
Helmholtzstr. 18, 89069 Ulm, Germany
E-mail: johannes.mayer@uni-ulm.de

Judith A. Stafford
Tufts University, Department of Computer Science
161 College Avenue, Medford, MA 02155, USA
E-mail: jas@cs.tufts.edu

Sven Overhage
University of Augsburg
Department of Software Engineering and Business Information Systems
Universitätsstr. 16, 86135 Augsburg, Germany
E-mail: sven.overhage@wiwi.uni-augsburg.de

Patrick J. Schroeder
Milwaukee School of Engineering
Department of Electrical Engineering and Computer Science
Milwaukee, WI 53202, USA
E-mail: schroedp@msoe.edu

Library of Congress Control Number: 2005932207

CR Subject Classification (1998): D.2.4, F.3, D.4, C.4, K.4.4, C.2

ISSN	0302-9743
ISBN-10	3-540-29033-8 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29033-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11558569 06/3142 5 4 3 2 1 0

Preface

The goal of software engineering is to achieve high-quality software in a cost-effective, timely, and reproducible manner. Advances in technology offer reductions in cost and schedule, but their effect on software quality often remains unknown. The International Conference on the Quality of Software Architectures (QoSA 2005) focused on software architectures and their relation to software quality, while the International Workshop on Software Quality (SOQUA 2005) mainly focused on quality assurance and more precisely on software testing. These events complement each other in their view on software quality.

One of the main motivations for explicitly modelling software architectures is to enable reasoning on software quality. From a software engineering perspective, a software architecture not only depicts the coarse-grained structure of a program, but also includes additional information such as the program's dynamics (i.e., the flows of control through the system) and the mapping of its components and connections to execution environments (such as hardware processors, virtual machines, network connections, and the like). In this area, QoSA 2005 is concerned with research and experiences that investigate the influence a specific software architecture has on software quality aspects. Additionally, the development of methods to evaluate software architectures with respect to these quality attributes is considered to be an important topic. The quality attributes of interest include external properties, such as reliability and efficiency, as well as internal properties, such as maintainability.

From a business-oriented perspective, software architectures are most often embedded into a greater organizational context (e.g., large enterprises) and cannot be seen in isolation from that context. Requirements that emerge from this context have a major impact on the architecture being developed and have to be dealt with by means of a business-oriented management of software architectures. In this field, QoSA 2005 aims at investigating the impact that activities like the coordination of business architecture and software architecture, business process modelling, assessment and acquisition of (COTS) components, as well as the integration or migration of legacy systems have on the quality of software architectures.

Although it is well-known that software architectures heavily influence software quality, validated research in this area is only recent. Today, even reliable experience reports that go beyond anecdotes from practitioner are rare. For a long time, the software architecture community was mainly concerned with formal specification of architectures. The use of architectures beyond the specification has only lately been taken into consideration by different communities that are producing results on the prediction of various quality attributes, software architecture evaluation, cost estimation, architectural re-use through patterns, etc. By recognizing the intrinsic relationship between the mentioned areas which share an architecture-based approach, the main idea of QoSA 2005 was to bring together researchers and practitioner from these different communities concerned with all areas relating to software architecture quality.

Quality assurance plays an important role in today's world and has gained increased importance. SOQUA 2005, which was organized within the Net.ObjectDays and co-located with QoSA 2005, mainly concentrated on this topic. Object-oriented concepts, component technology, components off the shelf (COTS), and open source software can dramatically reduce development time; however, assuring the quality of systems using these technologies is problematic.

The job of measuring, assuring, and improving the quality of software systems is getting harder with new technologies, not easier. The goal of this workshop was to bring together researchers, engineers, and practitioners to discuss and evaluate the latest challenges and breakthroughs in the field of software quality. The main focus of the workshop was on software quality assurance and more specifically on software testing. The generation of test data is still one of the most prominent problems in this area. Therefore, a number of papers presented and published are dedicated to this important problem.

In line with a broad interest, QoSA 2005 received 32 submissions. From these submissions, 12 were accepted as long papers after a peer-review process. They are published in this volume, together with an extended abstract of the invited talk by Christine Hofmeister. Five additional submissions were considered as original new research, but without having such an elaborated validation as the accepted, more mature long papers. These papers were accepted as posters and were published as short papers in the general Net.ObjectDays 2005 proceedings. Having received this high attraction encourages us to continue with shaping a community that is focused on software architecture quality and establishing QoSA as their primary conference in the future.

SOQUA 2005 attracted 17 submissions from all over the world. In total 6 papers could be accepted as long papers after a peer-review process. These papers are published in this volume, together with an extended abstract on the invited talk by T.Y. Chen. Three additional papers were accepted as short papers, which were published in the Net.ObjectDays proceedings and presented within a special joint session with the Net.ObjectDays Developer Track.

Among the many people who contributed to the success of QoSA 2005 and SOQUA 2005, we would like to thank the members of the Program Committees for their valuable work during the review process, Ch. Hofmeister for her keynote at QoSA 2005, and T.Y. Chen for his invited talk at SOQUA 2005. Additionally, we thank the organizers of the Net.ObjectDays 2005, in particular Mrs. Paradies, for their support in all organizational concerns as well as Mr. Hofmann from Springer for his support in reviewing and publishing the proceedings volume. The QoSA organizers would also like to thank the cooperating partners for their support. The SOQUA organizers are grateful to their cooperating and supporting organizations and in particular to Julia Codrington, Wolfgang Grieskamp, Chani Johnson, and Mario Winter for their support.

July 2005

Ralf Reussner
Johannes Mayer
Judith Stafford
Sven Overhage
Steffen Becker
Patrick J. Schroeder

Organization

QoSA 2005

Organizers and Program Chairs

Ralf Reussner, University of Oldenburg, Germany
Judith Stafford, Tufts University, USA
Sven Overhage, Augsburg University, Germany
Steffen Becker, University of Oldenburg, Germany

Program Committee

Colin Atkinson, University of Mannheim, Germany
Antonia Bertolino, ISTI-CNR, Italy
Alexander Brändle, Microsoft Research, UK
Christian Bunse, Fraunhofer IESE, Germany
Michel Chaudron, Eindhoven University of Technology, Netherlands
Ivica Crnkovic, Mälardalen University, Sweden
Peter Dadam, University of Ulm, Germany
Viktoria Firus, University of Oldenburg, Germany
Ulrich Frank, University of Duisburg-Essen, Germany
Kurt Geihs, University of Kassel, Germany
Ian Gorton, NICTA, Australia
Volker Gruhn, University of Leipzig, Germany
Wilhelm Hasselbring, University of Oldenburg, Germany
Jean-Marc Jézéquel, IRISA (Univ. Rennes & INRIA), France
Stefan Kirn, University of Hohenheim, Germany
Juliana Küster-Filipe, University of Birmingham, UK
Raffaela Mirandola, Università Roma “Tor Vergata”, Italy
Jürgen Münch, Fraunhofer IESE, Germany
Dietmar Pfahl, Fraunhofer IESE, Germany
Frantisek Plasil, Charles University, Czech Republic
Iman Poernomo, King’s College London, UK
Andreas Rausch, University of Technology Kaiserslautern, Germany
Matthias Riebisch, Technical University Ilmenau, Germany
Bernhard Rumpe, University of Technology Braunschweig, Germany
Christian Salzmann, BMW Car IT, Germany
Heinz Schmidt, Monash University, Australia
Jean-Guy Schneider, Swinburne University of Technology, Australia
Johannes Siedersleben, sd&m AG, Germany
Elmar Sinz, University of Bamberg, Germany
Michael Stal, Siemens AG, Germany
Clemens Szyperski, Microsoft Research, USA

Kurt Wallnau, Software Engineering Institute, USA
Wolfgang Weck, Independent Software Architect, Switzerland

Co-reviewers

Guglielmo De Angelis, Università di Roma, Italy
Samir Amiry, Fraunhofer IESE, Germany
Reinder Bril, Eindhoven University of Technology, Netherlands
Yunja Choi, Fraunhofer IESE, Germany
Aleksandar Dimov, Mälardalen University, Sweden
Simon Giesecke, University of Oldenburg, Germany
Vincenzo Grassi, Università di Roma, Italy
Jens Happe, University of Oldenburg, Germany
Rikard Land, Mälardalen University, Sweden
Moreno Marzolla, Università di Venezia, Italy
Johan Muskens, Eindhoven University of Technology, Netherlands
Sasikumar Punnekkat, Mälardalen University, Sweden
Daniel Schneider, Fraunhofer IESE, Germany
Massimo Tivoli, Mälardalen University, Sweden
Johan Fredriksson, Mälardalen University, Sweden
Erik de Vink, Eindhoven University of Technology, Netherlands
Timo Warns, University of Oldenburg, Germany

Cooperating and Supporting Partners

Augsburg University, Germany
Carnegie Mellon University/Software Engineering Institute (SEI), Pittsburgh, USA
Fraunhofer IESE, Kaiserslautern, Germany
German Computer Science Society (GI e.V.), GI AKSoftArch, Germany
Microsoft Research, Cambridge, UK
OFFIS, Oldenburg, Germany
Oversoft Software, Frankfurt, Germany
sd&m, Munich, Germany
Tufts University, Boston, USA
University of Oldenburg, Germany

SOQUA 2005

Organizer

Johannes Mayer, University of Ulm, Germany

Program Chairs

Johannes Mayer, University of Ulm, Germany
Patrick J. Schroeder, Milwaukee School of Engineering, USA

Program Committee

Co-reviewers

Cooperating and Supporting Partners

ACM SIGSOFT, USA
German Computer Society (GI e.V.), SIG TAV, Germany
Microsoft Research, Redmond, USA
Milwaukee School of Engineering, USA
University of Ulm, Germany

Table of Contents

Reexamining the Role of Interactions in Software Architecture

Christine Hofmeister

Computer Science and Engineering Department, Lehigh University,
19 Memorial Drive West, Bethlehem, Pennsylvania 18015, USA
Hofmeister@cse.lehigh.edu

Describing and understanding the interactions among computation units in a software system remains one of the most interesting challenges of software architecture. In the old days, computation units were statements, and interaction among them consisted of the control flow between them. Structured programming and modern languages have made this quite tractable for developers to describe and understand. But object-orientation, concurrency, and component-based systems have pushed us into a realm where interactions between computation units (components) are often quite complex and difficult to understand.

Over the years numerous kinds of models have been used to describe interactions, including state charts, UML sequence diagrams, Petri nets, special-purpose languages, connectors, etc. Another less common, approach is to take advantage of the duality of application state and interaction, either using interaction state as a proxy for application state, or vice versa. With models the typical approach is to describe the allowable interactions of a component via an interaction protocol, then use these to determine whether components may be composed and perhaps to create a new model describing the composed interactions. However, component interactions, like many other component properties, can depend on the context or environment in which the component operates, and on the other components it is composed with. Describing these things in an interaction protocol and developing a corresponding composition semantics remains a problem.

Another issue has arisen with the use of multiple views to describe the software architecture. Interaction behavior is a key part of many types of views, which results in the need to reconcile the relationship between interaction protocols across views. A layered model of interaction protocols may be useful here. For example, the lowest layer could be RPC and HTTP interactions in an execution view, the middle layer be the interactions in a module view, and the top layer be the logical interactions in a conceptual view. Adaptation of interactions in cases of incompatibility, e.g. via packaging, is also important, but typically addresses the relationship between interaction protocols in the same layer rather than across layers.

Some recent work has moved away from a strictly static analysis of interaction protocols to include run-time checking of interactions. This greatly expands the kinds of things that can be described and checked in an interaction protocol, but does not address problems of composing interaction protocols nor of checking their consistency across protocol layers. Solving these problems may require a paradigm shift, from describing interaction protocols as properties of components to treating interactions as entities in their own right, with their own sets of properties.

Are Successful Test Cases Useless or Not?

T.Y. Chen

Faculty of Information and Communication Technologies,
Swinburne University of Technology, Hawthorn 3122, Australia
tchen@ict.swin.edu.au

Test cases are said to be successful if they do not reveal failures. With the exception of fault-based testing, successful test cases are generally regarded as useless. If test cases are selected according to some testing objectives, intuitively speaking, successful test cases should still carry some information which may be useful. Hence, we are interested to see how to make use of this implicit information to help reveal failures. Recently, we have studied this problem from two different perspectives: one based on the properties of the problem to be implemented; and the other based on the notion of failure patterns which are formed by the failure-causing inputs, that is, inputs that reveal failures.

This paper presents a technique which is developed upon the notion of failure patterns. Once a program has been written, its failure-causing inputs are defined, though unknown (otherwise the testing task would be trivial). Furthermore, these failure-causing inputs form different patterns which may be classified as ***block***, ***strip*** and ***point*** failure patterns. An example taken from Chan et al. [1] is shown in Figures 1a-1c.

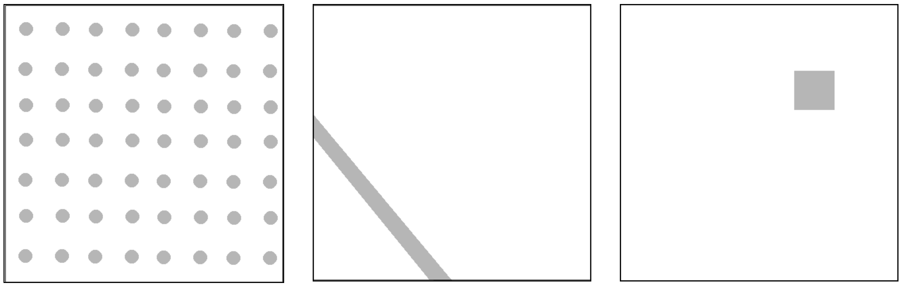


Figure 1a. Point Pattern   Figure 1b. Strip Pattern   Figure 1c. Block Pattern

Fig. 1. Types of Failure Patterns

It has been observed that ***block*** and ***strip*** patterns occur more frequently than the ***point*** patterns. For Random Testing, intuitively speaking, an even spread of test cases should have a better chance of detecting failure when the failure patterns are non-***point*** type. Hence, the approach of Adaptive Random Testing (***ART***) has been proposed to enhance the fault-detection capability of Random Testing. ART is basically a Random Testing method with an enforced even spread of random test cases. Simulation and experimental results have shown that ***ART*** may use less than 50% of test cases to detect the first failure as compared with Random Testing with replacement.

Obviously, there are different approaches of evenly spreading random test cases. As a consequence, there are various methods to implement ***ART***. It is interesting to investigate the strengths and limitations of these even spreading approaches, as well as to identify the conditions appropriate or favourable for their applications.

Acknowledgement

The author is most grateful to Dave Towey, Robert Merkel, Diana Kuo, Kwok Ping Chan, Aaron Huang, Zhiquan Zhou, Huai Liu and Changhai Sun who have worked with him on Adaptive Random Testing.

Reference

1. K. P. Chan, T. Y. Chen and D. Towey, ‘Normalized Restricted Random Testing’, 8th Ada-Europe International Conference on Reliable Software Technologies, Toulouse, France, June 16-20, 2003, Proceedings. LNCS 2655, pp. 368-381, Springer-Verlag Berlin Heidelberg 2003.

DoSAM – Domain-Specific Software Architecture Comparison Model*

Klaus Bergner[1], Andreas Rausch[2], Marc Sihling[1], and Thomas Ternité[2]

[1] 4Soft GmbH, Mittererstraße 3,
D-80336 Munich, Germany
{bergner, sihling}@4soft.de
[2] Technische Universität Kaiserslautern, Fachbereich Informatik,
Arbeitsgruppe Softwarearchitektur, Gottlieb-Daimler-Straße,
D-67653 Kaiserslautern, Germany
{rausch, ternite}@informatik.uni-kl.de

Abstract. The architecture of an IT system is of crucial importance for its success. In order to assess architecture's fitness, a number of standardized architecture evaluation methods have been proposed. Most of them are intended for the evaluation of a single architecture at a certain point in time. Furthermore, the results are often highly dependent on the person performing the evaluation. Thus, such methods cannot be used to compare and rate different architectures. The DoSAM method instead provides an evaluation framework for comparing different software architectures in a certain domain. After adapting this framework to the application domain at hand once, it can then be used repeatedly for all future evaluations in a methodical and reproducible way.

1 Introduction

The architecture of an IT system is a key success factor throughout its whole life cycle. During development, the architecture serves as a blueprint for all further design and implementation activities. Decisions met by the software architect determine quality attributes of the final system like performance, scalability and availability [6]. If these decisions must be revised later on, substantial costs and delays may arise. Even more serious are the consequences for the operation and maintenance phase, during which the majority of costs accumulate [2, 3]. Only future-proof systems based on a clear, extensible structure can be adapted to new requirements with reasonable effort [5].

A method for assessing the strengths and shortcomings of system architectures is, therefore, very valuable in a multitude of situations. First of all, it allows the software architect to evaluate his decisions early in the software life cycle in order to avoid costly wrong decisions. Another application is the assessment of an existing legacy

* This work originates from a framework of research studies related to Modularity of Naval Ships and funded by the German Procurement Agency Bundesamt für Wehrtechnik und Beschaffung in Koblenz.

system in order to guide its redesign. Finally, potential customers may use such a method to rate the architectures of alternative systems during the tender process.

Today, a number of architecture evaluation methods exist, like SAAM and ATAM (both described in [1]). Most of them concentrate on the assessment of a single system's architecture at a certain point of time, yielding a list of strengths and weaknesses (like SAAM) or of critical design tradeoffs (like ATAM). This is helpful for performing an architecture review for a single system or for selecting a certain design variant for a special aspect of a system's architecture.

However, the majority of current architecture evaluation methods do not support the comparison and rating of different architectures against each other. Note that the question here is not whether a certain detail of architecture A's solution is better compared to architecture B's solution – often, questions like that have an immediate, obvious answer. However, sacrificing an optimal detail solution in one area may pay off well in other areas, and may indeed be a deliberate and wise decision. To take aspects like this into consideration, and to shift the view from detail problems to the big picture, assessment methods need to take a more general and more specific approach at the same time: More general, as they have to provide an assessment framework that is applicable for different software architectures, taking into account their strengths and weaknesses in order to ultimately get a single number on a scale. More specific, this weighting process must be based on clearly stated, transparent criteria derived from the given context and goals of the application domain.

The DoSAM method presented in this paper can bring together these contrasts. Its central idea is to provide a domain-specific software architecture comparison framework, which serves as a reference for different architectures in a certain domain. The comparison framework is derived from domain-specific scenarios, and comprehends as its two dimensions the necessary architecture services and the quality attributes for the domain under consideration.

The comparison framework is developed once and can then be used repeatedly to assess different architectures. This makes it valuable especially for application in the tender procedure, but also for comparing and selecting different overall approaches during system architecture design.

In principal, the DoSAM method may also be used for a single architecture review. Compared to other methods, the additional investment in the reference framework leads to additional costs, however. This weakness turns into strength when the architecture is evaluated more than once, e.g. to support an iterative design process. Furthermore, the creation of the reference comparison framework usually brings additional clarity, as it forces the assessor to think about the services and quality attributes necessary for the application domain.

The rest of the paper is structured as follows: First, in Section 2 we shortly outline the state of the art with respect to software architecture evaluation methods. Then, in Section 3 we give an overview over the proposed approach for a domain-specific software architecture comparison model. In Section 4 we present a sample application of DoSAM. Section 5 shows a part of the comparison framework for a specific quality attribute. A conclusion is given at the end of the paper in Section 6.

2 Related Work

Although commonly accepted in its importance for cost and quality of software systems, the methodic assessment of software architectures is a rather young discipline. Most approaches rely on academic work presented in [1]:

SAAM - Software Architecture Analysis Method - Assessment of software architectures is done via scenarios, quality attributes and quality objectives. Scenarios provide an understanding of typical situations the architecture must cope with. Quality attributes and quality objectives define the aspects of interest. During the process of assessment, the architectural approaches at hand are identified and evaluated with respect to the identified objectives. As a result, each architectural decision is associated with one or more risks to indicate problem areas.

ATAM – Architecture Tradeoff Analysis Method - In comparison to SAAM, this approach concentrates on finding crucial tradeoff-decisions and tries to detect the risks involved. Therefore, ATAM results in a good understanding of the pros and cons of each tradeoff and the implications of alternative solutions in the context of a given overall architecture.

CBAM – Cost-Benefit Analysis Method - The CBAM method [10] enhances ATAM by quantifications of the economic benefit and cost of architectural decisions.

ALMA – Architecture-Level Modifiability Analysis - Compared to the other approaches, the ALMA method [11] has its focus on adaptation and modification scenarios for a given architecture and hence takes a special interest in the corresponding subset of quality attributes.

ARID – Active Reviews for Intermediate Designs - With a focus on the architecture of subsystems, this lightweight evaluation approach provides a procedure for the early phases of architectural design. The proposed technique is that of design reviews – a simple and effective, but rather subjective method.

SACAM – Software Architecture Comparison Analysis Method - This recently proposed method allows for comparison of a set of software architectures possibly of quite different application domains (cf. [8]). From a company's business goals, the relevant quality criteria are derived and described by means of scenarios (similar to ATAM). Based on predefined metrics and architecture views, the evaluator looks for indicators to score each architecture with respect to the criteria.

The DoSAM method presented in this paper is based on similar base concepts, like scenarios and quality attributes. However, in addition, it brings in some new aspects:

- Apart from SACAM, none of the presented approaches allows for the comparison of several architectures of the same application domain. The evaluation procedure is highly tailored to the specifics of the architecture under consideration and depends on the person performing the evaluation. DoSAM strives for a more objective evaluation which can be repeated as requirements change over time.
- Like SACAM, the preparation of the comparison framework only needs to be done once for a certain application domain. Once the comparison framework has been adapted, it can be easily reused for additional architectures or for re-evaluation after modifications of existing ones.

- In the process of a SACAM evaluation, the evaluator tries to find a common level of abstraction for comparison by identifying a number of common views. This may be quite difficult, especially when comparing completely different architectures. DoSAM instead restricts itself to the comparison of architectures in the same application domain based on a common architecture blueprint. This blueprint serves as a conceptual framework, providing guidance for the assessor and simplifying the comparison process.
- Furthermore, DoSAM introduces architectural services as a means of describing the basic functionality and purpose of the application domain in an abstract way. Compared to application scenarios, which are often very specific and may change over the lifetime of the architecture, architectural services provide a more high-level and stable view. Conceptually, services are roughly similar to SACAM view types, as they determine which architectural aspects are assessed. However, in contrast to SACAM view types, services are motivated by the application domain and not by technical issues found in the architectural documentation (e.g. component dependency and complexity metrics, as mentioned as examples by SACAM).

To sum up, nowadays most architecture evaluations focus on the evaluation of a single architecture, not on the repeatable and standardized comparison of different architectural solutions. The only exception known to us is the SACAM method. Compared to it, DoSAM provides additional guidance, structure and stability due to the explicit consideration of a given application domain.

3 Domain-Specific Software Architecture Comparison Model

The DoSAM approach supports the evaluation and comparison of different architectural solutions within a given domain. Once a *domain architecture comparison framework* (DACF) has been created by an expert for a certain application domain, the *concrete architecture evaluation* (CAE) of the architecture in this domain can be performed following a standardized and repeatable process.

The structure of the DACF is shown in Fig. 1. Each of the inner boxes represents an intermediate result to be elaborated during the elaboration of the DACF.

As can be seen, the framework consists of three main parts:

1. *What is evaluated?* - The *domain architecture blueprint* and the *architectural services* together form an abstract description schema for all architectures in the application domain. The intention here is to find constituents of the architectures that are inherent to the application domain at hand, and therefore inevitably represented in all imaginable solutions. Thereby, it is crucial to find the right level of abstraction: it must not be too detailed to be able to form a representative blueprint for the domain but it must not be to abstract to be able to reason about the relevant architectural properties. Hence a domain specialist is needed to determine the proper architecture blueprint. Thereby, prevalent architectural styles and patterns in a given domain have to be taken into account (cf. [12]). An example for an abstract architecture service would be the *data transfer service* of a network-centric system, made up by a certain number of hardware and software components in a specific architecture.

2. *How is it evaluated?* – The *relevant quality attributes* and the corresponding *quality attribute* metrics (one for each quality attribute) state in which respect the architectures are assessed and how the assessment of these criteria is performed. An example for a quality attribute would be the *availability*, which could be measured by quality attribute metrics like the mean time to failure (MTTF) in a certain scenario.
3. *What is important?* – Finally, the *quality computation weights* in the middle of the figure must be provided to state the relative importance of each application of a quality attribute metrics on an architectural service. For example, the assessor might decide that the *availability* of the *data transfer service* is very important in a certain application domain, at least in relation to the availability of other architecture services.

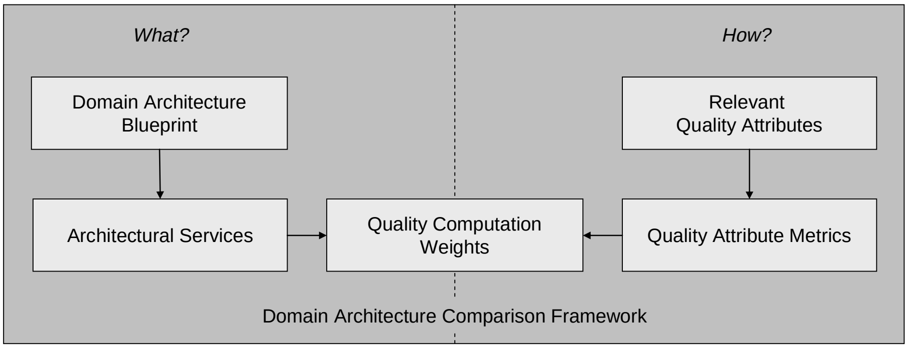

What?
How?
Domain Architecture Blueprint
Relevant Quality Attributes
Architectural Services
Quality Computation Weights
Quality Attribute Metrics
Domain Architecture Comparison Framework

Fig. 1. Domain Architecture Comparison Framework (DACF)

The possibility of adjusting the architectural services and the quality attributes, as well as the metrics and weights, makes the creation and fine-tuning of a DACF a challenging task. However, when this task has been performed well, the resulting DACF can be intuitive and understandable even for persons with no deeper architectural knowledge. At a high level, the DACF might capture statements like that military systems rely heavily on the availability, safety and security of communication services, whereas business systems might emphasize the performance of their transaction and storage services, for example.

As already stated, the creation of the DACF has to be performed only once for each application domain. Note that the modular structure of the DACF makes it possible to add new quality attributes and/or architecture services to the framework without breaking the existing framework.

After the creation of the DACF (shown in the upper part of Fig. 2) it can be applied to evaluate and compare the concrete architectures. This is a three-step process, which closely corresponds to the three main parts of the DACF shown in Fig. 1. It consists of the three following activities, which are performed for each concrete architecture to be evaluated:

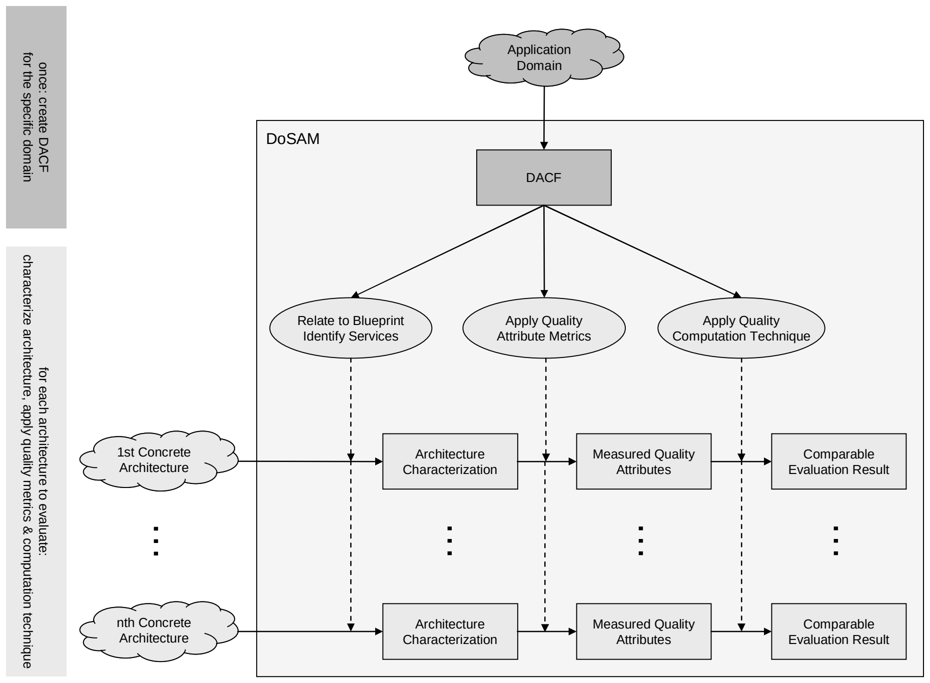

once: create DACF for the specific domain
for each architecture to evaluate: characterize architecture, apply quality metrics & computation technique
Application Domain
DoSAM
DACF
Relate to Blueprint Identify Services
Apply Quality Attribute Metrics
Apply Quality Computation Technique
1st Concrete Architecture
nth Concrete Architecture
Architecture Characterization
Measured Quality Attributes
Comparable Evaluation Result

Fig. 2. DoSAM Application: DACF Creation and Concrete Architecture Evaluation (CAE)

1. *Relate to Blueprint & Identify Services:* The concrete architecture is related to the architectural blueprint of the application domain, yielding the set of all hardware and software components and their connections that are relevant for the subsequent steps. This may result in one or more high-level architecture overview diagrams of the whole system. Based on that, the components involved in the realization of the identified architectural services are identified. For the data transfer service of a business transaction system, this may, for example, result in hardware and software components like routers, firewalls, protocol stacks and caching software.
2. *Apply Quality Attribute Metrics:* The concrete service implementation of the evaluated architecture is examined and assessed with respect to the identified quality attributes of the application domain. This is done by employing the quality attribute metrics. Each of these metrics yields a single evaluation result on a normalized scale from 0 to 100. The data transfer service could, for example, be evaluated with respect to performance by assessing the combination of average throughput and average response time. A detailed example for the *modifiability* quality attribute metrics is given in Section 5.3.
3. *Apply Quality Computation Technique:* When all architecture services have been assessed with respect to all quality attributes, the evaluation results may be entered

into the weighted quality computation matrix. Again, this leads to a single, normalized evaluation result, characterizing the fitness of the architecture compared to other architectures and providing a hint for its absolute value with respect to the criteria captured within the DACF.

Provided that a well-designed DACF has been elaborated, the degree of freedom in performing the concrete architecture evaluation (CAE) will be rather low, ensuring a fair, repeatable evaluation of all architectures. This also means that a CAE can also be performed by people who are not highly-skilled application domain or architecture experts. Furthermore, if the application of the quality attribute metrics is supported by tools, the effort and time needed to perform a CAE can be kept very low.

4 Sample Application Domain

The following sections will briefly demonstrate the presented assessment process for the domain of IT architectures for modern naval crafts. We have chosen this application domain because DoSAM has been designed for a navy related study in the first place [9], but also because of the interesting architectural challenges involved:

- Compared to the clear, historic separation of duties for the different kinds of military ships, modern times demand highly flexible crafts that can easily be modified to fit requirements of various mission scenarios (e.g. clearance of mines, evacuation, etc.)
- Cooperation with units of different, international armies steadily grows in importance and so does the ability of interaction and communication.

Consequently, quality attributes like modularity, flexibility, and extensibility are of growing importance for the system architecture of modern military crafts. The chosen sample application domain is that of “modular ships” with the following understanding of modularity:

- All ships use a common, basic architecture which can be enhanced and specialized for all different kinds of crafts. This idea of a shared infrastructure platform results in a rather high degree of reuse and hence reduces the costs for development, production, maintenance and education.
- A single ship can be equipped for a couple of different deployment scenarios, which is done either in the shipyard or in the field. That is, separate modules may be installed, each adding new features and services.
- Single modules can be standardized and standard COTS (commercial off-the-shelf) components can be used within the ship to further reduce development and maintenance cost.

To cope with the complexity of this modular approach, the demands on the quality of the underlying software and hardware infrastructure platform are very high. Its architecture is subject to assessment in the following chapters.

5 DoSAM Applied

This section offers a concrete example of the application of DoSAM as described in Section 3 in the sample application domain presented in Section 4. The presentation follows the three main areas of the DACF as shown in Fig. 1 as well as their application during the CAE as shown in Fig. 2. First, in Sections 5.1 and 5.2 the domain architecture blueprint and the corresponding architecture services are described, respectively. In the following, the relevant quality attributes are described in Section 5.2, and an example for a detailed quality attribute metrics is given in Section 5.3. Finally, the quality computation technique and its application are shown in Section 5.4.

5.1 Domain Architecture Blueprint

In order to evaluate architectures using the DACF evaluation model, it is necessary to create an architecture blueprint that shows a high level of universality. As already mentioned in Section 3, such a blueprint must not be too detailed to represent all concrete domain architecture implementations and to support the mapping to real system architectures without constraining them. But the blueprint must not be to abstract, otherwise it will not be powerful enough to reason about the relevant architectural properties. It is not an easy task to find such a blueprint, but extensive usage of the knowledge of domain specialists will lead to good results. During development, universality of the blueprint must be kept in mind.

In our sample application domain described in the previous section, a component based style seems to be a reasonable choice. Obviously, in other domains, other styles may be more appropriate. For example, one would expect pipes and filters in a signal processing application or, if applicable, the architecture might be oriented on event triggers [12].

Fig. 3 shows a domain architecture blueprint for the sample application domain described in the previous section. As stated above, this blueprint follows an architectural style based on components. Obviously, this central platform consists of both hardware and software. Software can be split up into application software and base software. Since we want to assess only the fitness of the base platform, the application-specific software is not evaluated, visualized by the grey and white boxes in Fig. 3. Here, grey boxes represent components of the central platform, whereas white boxes show application-specific software and hardware that relies on it.

Base software and hardware are each subdivided into four categories:

- Adapters serve as interfaces for sensors and effectors and their software. Examples in the hardware domain are network adapters and hardware interfaces for sensors and effectors. In the software domain, generic device drivers fall into this category.
- Workstation hardware (consisting of CPU, main memory, data storage subsystem etc.) and software (e.g. operating system, communication software, mail client etc.)
- Server hardware and software, similar to their workstation counterparts.
- Network infrastructure consisting of switches, routers, firewalls (both hardware and software) and the network cables.

The categories depicted by grey boxes define the scope of the subsequent evaluation: During the assessment of the architectural services within the CAE, only components falling into one of these categories may be considered.

To relate the concrete architecture to the blueprint architecture, one or more architecture overview diagrams may be created. A simple representation could look as

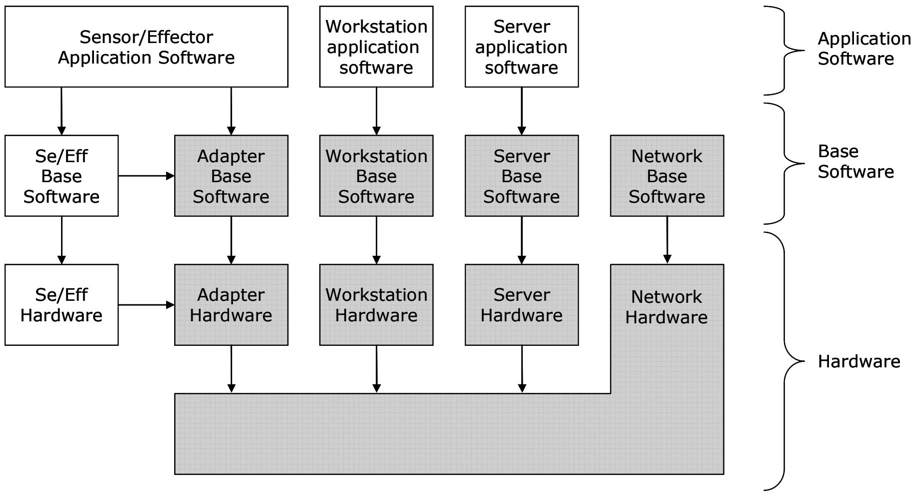


Fig. 3. Domain Architecture Blueprint for the Base Platform as part of the DACF

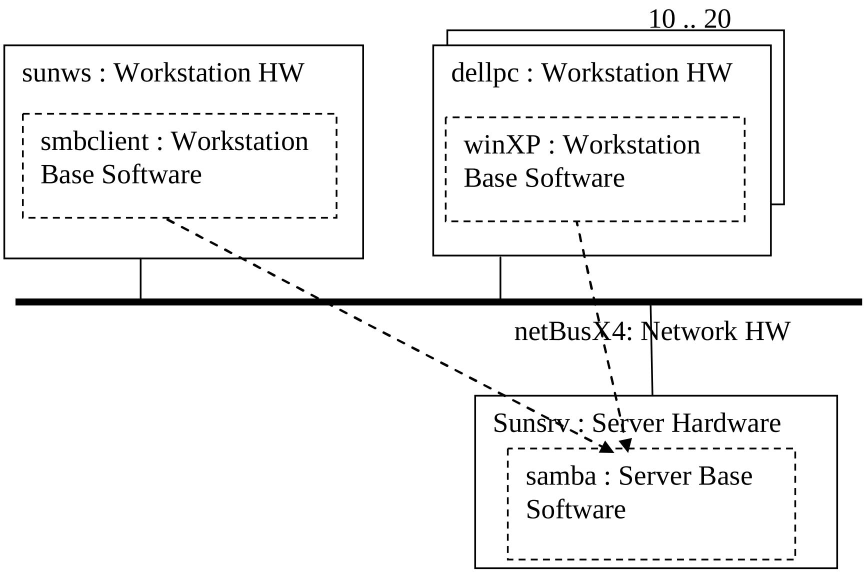

10 .. 20
sunws : Workstation HW
smbclient : Workstation Base Software
dellpc : Workstation HW
winXP : Workstation Base Software
netBusX4: Network HW
Sunsrv : Server Hardware
samba : Server Base Software

Fig. 4. Sample Part of an Architecture Overview Diagram created during CAE

shown in Fig. 4, with software components shown as boxes with a dashed border andhardware components as boxes with a solid border. Alternatively, UML description techniques like deployment or component diagrams could also be used, as far as they can visualize the component categories of the blueprint architecture.

5.2 Architectural Services

The central platform is providing various services for the applications running on the ship. The following architecture services have been identified as crucial for the evaluation of different architectural solutions:

- **Data transfer service:** Achieves the transfer of heterogeneous data between different applications and hardware modules with naval, military and civil application. The data transfer service must provide an infrastructure for multimedia messages like video/speech, real-time data read in by sensors, and application messages between different application services.
- **Data storage service:** Achieves the persistent storage for real-time, multimedia, and application data as well as for arbitrary documents.
- **Processing service:** CPU power must be supplied for computationally expensive tasks. This applies to both interactive and batch programs.
- **System management service:** Monitoring, administration and maintenance of the platform and its components must be achieved.
- **Authentication/authorization service:** Carries out the identification, authentication and authorization of users, applications and components.
- **Presentation service:** Achieves the presentation of GUIs and output on end devices like screens, printers etc.

These application-specific architecture services decompose the functionality of the underlying system into separate areas that can be evaluated independently.

Note that in order to assess each service for a given architecture during the CAE, it must be possible to identify the components that constitute its implementation. This may be done by marking the components on the architecture overview diagram, thus building a series of cut-out views from the overall architecture. Compared to the full architecture, the complexity of such an architecture service view will usually be greatly reduced, thus simplifying the evaluation of the services.

Very simple examples for architectural service cut-out views taken from Fig. 4 could range from all components (for the remote data storage service, in which all components might participate) to just the `dellpc` workstation hardware and software (for the presentation service, which could employ these workstations as terminals for other computers without screens).

5.3 Relevant Quality Attributes

Quality attributes like those found in the IEEE standard for quality attributes [7] define criteria for a certain capability area of the architecture. In a military environment, especially on a ship, the following quality attributes play an important role in the fulfillment of its purpose.

- **Availability** evaluates the capability of the architecture to provide continuous functionality. It is defined as the relative amount of time the system functionality is accessible. System break downs and malfunctions have a negative effect on availability.
- **Performance** is influenced by parameters like throughput and response time. It measures how fast the architecture is and how many users can use it simultaneously.
- **Usability** characterizes the operability of the system. The user interfaces must provide fast support of routine processes on the ship. The architecture defines the framework to be used by applications in order to provide a user-friendly interface.
- **Safety and Security** as quality attribute evaluates the possibility of input errors and unwanted effects (safety) and the degree of protection against intended and unintended attacks (security).
- **Modifiability** defines the degree to which the architecture is configurable and extendible. This quality attribute includes aspects like developer friendliness, understandability and modularity. It has a crucial influence on the efforts and costs for necessary changes.
- **Functionality** valuates the completeness of the functionality and features provided for applications depending on the platform.

The description above stays at a very abstract level, of course. In order to be useful for a concrete evaluation, there must be unambiguous operational procedures that can be applied in a repeatable way. During the creation of the DACF, we have devised a number of such procedures. As all of them yield as result a single number normalized by a scale from 0 to 100, we have called them *quality attribute metrics*. The following section gives an example.

5.4 Sample Quality Attribute Metrics – Modifiability

As an example for the evaluation procedure of a particular quality attribute, our quality attribute metrics for the modifiability quality attribute is described in the following.

In the scenario described in Section 4, modifiability is a central aspect of the intended modularity of the ship. The following requirements have to be regarded in the evaluation of the architecture:

- The quality of the modular ship must not be degraded due to the change.
- Preparation and realization of the modification must be done in as little time as possible.
- The costs for preparation and realization of a change must be as low as possible.

In summary, the aspects quality, time, and costs enable an evaluation of modifiability of a modular ship. Time and costs can be determined and compared for different scenarios, but the resulting quality is not quantifiable as easily. In the following context, we assume that methods guaranteeing quality assurance like regression tests are used. This induces the assumption that the quality is not worsened by potential changes in the following context.

Concerning modifiability, the following types of change can be identified:

- **CS: Change of solution**: A change resulting from an alteration of the technical setup without any change in functionality that could be perceived by the user. The technical solution only defines *how* the system works.
- **CF: Change of functionality**: A change of functionality implies a change of any abilities of the system determining *what* the system does. This can include a change of solution, too.

The different change types (CS, CF) can be realized by the following change realization types:

- **CC: Change by configuration**: The configuration parameters are the only parts of a component that underlie a change. This usually needs the least realization time.
- **CI: Change by installation and deinstallation**: New components are integrated into the modular ship or removed, respectively. In order to be subject to a CI, the particular component and its modules must already be known and well specified. Otherwise, this could lead to a decreased quality.
- **CD: Change by development**: The implementation of a component is changed or totally renewed. As mentioned before, quality assurance methods have to be applied in order to prevent a decreasing quality due to newly implemented components.

Usually, a change by development directly implies a change by installation and deinstallation, whereas a CI usually contains a change by configuration.

In order to enable an evaluation of modifiability, different changing scenarios have to be identified first. These scenarios are written down in an evaluation table (see **Table 1**). Identifying these scenarios is part of adjusting the DACF and must therefore only be done once.

Table 1. Exemplary Evaluation Table for Modifiability

Changing scenario	**Frequency of change**	**Services involved**	**Change realization type**	**Amount of architecture components to be changed**	**Average changing effort in MD**	**Dependent architecture components**	**Average integration and test effort in MD**	**Total change effort in MD**	**Persons affected**	**Additional costs**	**Total costs**
CS: Change of solution:											
Scenario 1 Change of DBMS	0,25 / Year	Data storage	CD	1	80	0	40	120	0	0 €	144,000 €
		Reason/details:	A change is restricted to the data storage service. It can only be achieved by programming. For a change of the DBMS, e.g. from Oracle to MySQL, the component Mission-App has to be revisited.								
CF: Change of functionality:											
Scenario 2 Integration of a different Mission Database	1 / Year	Data storage Processing System Mgmt. Presentation Authorization	CD	1	60	2	40	180	0	0 €	864,000 €
		Reason/details:	A change involves all services except the data transfer service. It can only be achieved by programming. The integration of already existing Oracle databases originating from other applications needs an enhancement of the database schema and the original database has to be adapted. The component Oracle 10i is directly involved.								

During architecture evaluation and comparison, for each architecture the change realization type of all scenarios has to be determined. Additionally, the following aspects are written down: the amount of architecture components that are touched by the change, the amount of dependent architecture components, the average effort for integration and testing, and for the change itself. Furthermore, the number of affected persons is noted and the additional are determined. When evaluating a concrete architecture, these values are used to calculate the architecture specific costs to realize scenario s. The calculation rule is explained below.

Every architecture is evaluated using the same scenarios. For each scenario s, the *total change effort* $TChEff_s$ in man days (MD) is determined using the following equation:

$$TChEff_s = ACCh_s \cdot \overline{ChEff_s} + \left(ACCh_s + ACDep_s\right) \cdot AITEff_s, \tag{1}$$

where $ACCh_s$ are the *average components to be changed*, $\overline{ChEff}_s$ is the *average change effort* for a component change, $ACDep_s$ are the *dependent architecture components*, and $AITEff_s$ is the *average integration and testing effort*. All these values are taken from the evaluation table exemplarily shown in **Table 1**.

The *total costs* TC_s are determined using the following equation, where a MD is exemplarily set to a standard value like 1200 Euro. Being noted in the last column of **Table 1**, TC_s represents the architecture specific costs to realize scenario s:

$$TC_s = Frq_s \cdot \left(CRT_s \cdot TChEff_s \cdot \left(PA_s + 1\right) \cdot 1200 + C_s\right), \tag{2}$$

where Frq_s is the assumed *frequency of a change*, CRT_s is the *change realization type*, PA_s are the *persons affected*, and C_s are the *additional costs* in this scenario s. The different change realization types each induce different factor values:

- Change realization type CC: $CRT_s = 1$
- Change realization type CI: $CRT_s = 2$
- Change realization type CD: $CRT_s = 3$

During architecture evaluation, not only potential changes and their costs are of interest. Understandability and applicability of the architecture itself have an influence on the quality of its modifiability, too. This holds true especially in a domain like a modular ship, where it is an important requirement that application components are exchangeable with low effort. In order to respect this, the architecture has to be evaluated regarding this aspect, too. This happens by determination of an *architecture understandability and applicability factor* $AUAF_d$ for each architecture service d:

$$AUAF_d = IQF_d \cdot UF_d, \tag{3}$$

where IQF_d is the *interface quality factor* and UF_d is the *understandability factor* defined by the following table.

Table 2. Definitions of interface quality factor and understandability factor

	IQF_d	UF_d
Standard interface	1	1
Fully specified interface	2	UFF
Informally specified interface	4	UFF

Table 2 shows the factor value for the three possible interface types of an architecture service: standard interface, fully specified interface, and informally specified interface. It is assumed that there exist no unspecified interfaces. The *understandability factor formula* UFF is defined by:

$$UFF = \frac{\sum_{m \in M} PAF_m \cdot PTF_m}{|M|}, \tag{4}$$

where M is the set of parameters involved and $|M|$ is its cardinality. The *parameter amount factor* PAF_m has the value 1, if the amount of parameters in the method m has less than 10 parameters, otherwise it has the value 2. PTF_m is the *parameter type factor* being determined by the following rule:

- Parameter type is a primitive data type: $PTF_m = 1$
- Parameter type is a complex data type: $PTF_m = 2$
- Parameter type is a complex data type with pointers: $PTF_m = 4$
- Parameter type is not defined: $PTF_m = 16$

An $AUAF$ value of 1 indicates the best value that can be achieved. After having determined the costs for each change scenario and the $AUAF_d$ for all architecture services d, the intended metric value of the modifiability Mod_d of an architecture service can be determined by:

$$Mod_d = 101 - AUAF_d \cdot \frac{TC + \sum_{s \in S} TC_s}{TC}, \tag{5}$$

where TC are the total project costs. The modifiability of a particular architecture service therefore is determined by the application understandability and applicability factor and the costs resulting from all changing scenarios that have been thought of during the development of the domain architecture comparison framework. This value is set in relation to the total project costs and then normalized to a maximum value of 100. A negative value is interpreted as 0.

5.5 Quality Computation Weights and Quality Computation Technique

The quality computation technique is based on a two-dimensional matrix with the architecture services in its rows and the quality attributes in its columns.

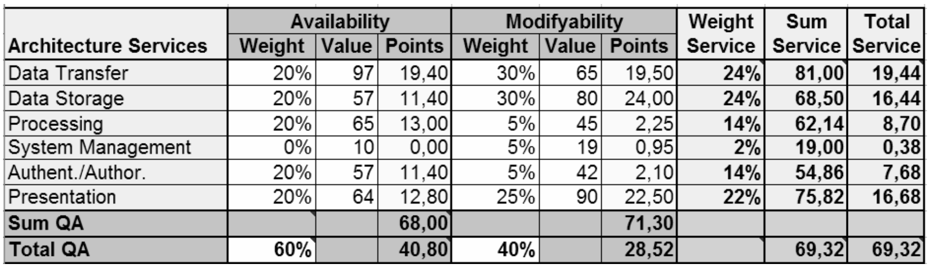
For every combination of quality attribute and architecture service, the DACF provides a quality attribute metrics yielding a metric percentage value normalized to the range from 0 to 100 (usually, the same quality attribute metrics will be used for all architecture service alike, although this is not strictly necessary). In Table 3, these values can be found in the Value columns. As can be seen in the example, the availability of the data transfer service has reached a high evaluation result of 97%, indicating the high fitness of the architecture with respect to this aspect.

In order to adapt the evaluation process to the requirements of the application domain, the DACF creator has to weight the relative importance of the quality attributes as well as the contribution of each architecture service to each quality attribute (QA). These weights are part of the DACF and must therefore be used for all CAEs alike without changes.

In Table 3, the quality attribute weights can be found in the white cells in the Total QA row, and the architecture service weights can be found in the Weight columns. As an example, the availability has been rated as more important compared to the modifiability (at a rate of 60 to 40). Furthermore, all services have been deemed equally important for the evaluation of the availability – with the single exception of the system management service, which in the example has no relevance for the availability altogether.

Note that only the cells with a white background can be set by the user – all other contents are provided automatically by the DoSAM quality computation technique. This pertains to the weighted points in the Points columns as well as to all Sums and Totals. The sums characterize the relative fitness of the QAs and services. In the example, the architecture reaches a score of 68% with respect to the availability QA, and of 71,30% with respect to the modifiability QA, while its total weighted score is 69,32%. This end result number can now be compared with the respective end results for other architectures, allowing to rate the architectures against each other.

Table 3. Exemplary Evaluation Matrix for Two Quality Attributes



	Availability			Modifyability			Weight	Sum	Total
Architecture Services	Weight	Value	Points	Weight	Value	Points	Service	Service	Service
Data Transfer	20%	97	19,40	30%	65	19,50	24%	81,00	19,44
Data Storage	20%	57	11,40	30%	80	24,00	24%	68,50	16,44
Processing	20%	65	13,00	5%	45	2,25	14%	62,14	8,70
System Management	0%	10	0,00	5%	19	0,95	2%	19,00	0,38
Authent./Author.	20%	57	11,40	5%	42	2,10	14%	54,86	7,68
Presentation	20%	64	12,80	25%	90	22,50	22%	75,82	16,68
Sum QA			**68,00**			**71,30**			
Total QA	**60%**		**40,80**	**40%**		**28,52**		**69,32**	**69,32**

Another benefit of the two-dimensional evaluation matrix is that an architecture may not only be assessed with respect to its QAs, but also with respect to its architecture services. To allow for this, the matrix first computes architecture service weights (column Weight Service) and weighted service sums indicating the total fitness of the architecture services (column Sum Service). In the example, the relative importance of the data transfer service evaluation is 24%, whereas the system management service reaches only 2%, due to the low weights of 0% and 5% in the corresponding Weight columns. Furthermore, the score of the system management of 19% is coincidentally also much lower than the score of the data transfer service of 81%.

The Sum QA row and the Sum Service column together form a powerful analysis tool for the architect. Not only does he see how good the architecture comes out with respect to the quality attributes, but he can also trace this back to his design decisions for single architecture services. In the example, it could have been a wise decision to put no emphasis on the system management service, as it doesn't contribute much to the overall fitness of the system.

6 Conclusion

In the paper, we have presented the DoSAM method for evaluating and comparing different architectures in a certain application domain. We have motivated the necessity for a new approach by comparing it to other architecture evaluation methods.

Our overview of DoSAM could of course not capture all details of the method. Therefore, we have restricted ourselves to a survey of the method's base principles and a short presentation of some key results from an example application, namely, the evaluation of IT application platforms for modular naval crafts. The actual, complete application example is much bigger – among others, it contains elaborated quality attribute metrics for all identified QAs, not only for the modifiability QA.

Our present experience with DoSAM up to now is encouraging: It seems to be a generally applicable and very flexible method suitable for all kinds of architectures. For the future, we plan to further elaborate and refine the DoSAM method based on additional evaluation experience. To do this, we will devise new or refined quality attribute metrics for the existing and for new QAs. In the following, better tool support for these QA metrics may then be developed in order to lower the time and effort necessary for concrete architecture evaluations. A long-term objective would be the design and implementation of a toolkit for rapidly building domain-specific architecture comparison frameworks based on existing quality attribute metrics.

References

1. P. Clements, R. Kazman, M. Klein: Evaluating Software Architectures, Methods and Case Studies, SEI Series in Software Engineering, 2002.
2. R. Harrison, 'Maintenance Giant Sleeps Undisturbed in Federal Data Centers', Computerworld, March 9, 1987.
3. B.P. Lientz and E.B. Swanson. Software Maintenance Management, Reading, MA: Addison-Wesley. 1980.
4. N. Lassing, P. Bengtsson, H. van Vliet and J. Bosch. Experiences with SAA of Modifiability, May 2000.

5. P. Bengtsson, J. Bosch, ‘Architecture Level Prediction of Software Maintenance’, The 3rd European Conference on Software Maintenance and Reengineering (CSMR'99), pp. 139-147, 1999.
6. L. Bass, P. Clements, R. Kazman, K. Bass. Software Architecture in Practice (Sei Series in Software Engineering). Addison-Wesley Publishing. 1998.
7. IEEE Std. 1061-1998, IEEE Standard for a Software Quality Metrics Methodology. 1998.
8. C. Stoermer, F. Bachmann, C. Verhoef, SACAM: The Software Architecture Comparison Analysis Method, Technical Report, CMU/SEI-2003-TR-006, 2003
9. K. Bergner, A. Rausch, M. Sihling, Verfahren zur Bewertung von DV-Architekturen für das modulare Schiff, 4Soft Customer Study, 2004.
10. J. Asundi, R. Kazman, M. Klein, Using Economic Considerations to Choose Among Architecture Design Alternatives, Technical Report, CMU/SEI-2001-TR-035, 2001.
11. P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, Architecture-Level Modifiability Analysis (ALMA), Journal of Systems and Software, Volume 69, Issue 1-2, 2004.
12. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, A System of Patterns, John Wiley&Sons, 1996.

An Architecture-Centric Approach for Producing Quality Systems

Antonia Bertolino[1], Antonio Bucchiarone[1,2], Stefania Gnesi[1], and Henry Muccini[3]

[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI-CNR), Area della Ricerca CNR di Pisa, 56100 Pisa, Italy
{antonia.bertolino, stefania.gnesi}@isti.cnr.it

[2] IMT Graduate School, Lucca
antonio.bucchiarone@imtlucca.it

[3] Dipartimento di Informatica, Universitá dell'Aquila, Via Vetoio 1, 67100 L'Aquila, Italy
muccini@di.univaq.it

Abstract. Software architecture has been advocated as an effective means to produce quality systems. In this paper, we argue that integration between analysis conducted at different stages of development is still lacking. Hence we propose an architecture-centric approach which, by combining different technologies and tools for analysis and testing, supports a seamless tool-supported approach to validate required properties. The idea is to start from the requirements, produce a validated model of the SA, and then use the SA to derive a set of conformance test cases. In this paper, we outline the process, and discuss how some existing tools, namely QuARS, ModTest, CowTest and UIT, could be exploited to support the approach. The integrated framework is under development.

1 Introduction

Based now on more than a decade of research in this new discipline, the Software Architecture (SA) [18] has today become an important part of software development processes (e.g. [22,9]). From SA application in practice, software developers have in fact learned how SA specifications permit to obtain a better quality software system, while reducing realization time and cost. In current trends SA description is used for multiple purposes: while some companies use the SA description just as a documentation artifact, others make also use of SA specifications for enhanced analysis purposes, and finally many others use the architectural artifacts to (formally or informally) guide the design and coding process [28,12].

We focus here on the first two purposes, i.e., using SA for documentation and analysis. Indeed, notwithstanding the enormous progresses in software technologies promoted by the affirmation of UML, open standards specifications and new development paradigms, the problem of assuring as early as possible the adequacy and dependability of a software system still takes a preponderant portion of the development cycle cost and time budgets. Considering the SA, analysis techniques have been introduced to understand if the SA satisfies certain expected properties, and tools and architectural languages have been proposed in order to make specification and analysis techniques rigorous and to help software architects in their work [33,11]. Even though much work has been done on this direction, many limitations still need to be handled: *i*) analysis results from one phase of the software process are not capitalized for use in the following phases, *ii*) tool support remains limited, fragmented, and in demand of high investment for introduction, *iii*) use in industrial projects generally requires specific training needs and extra requirements. In analyzing the difficulties behind wider industrial take-up of SA methodology, one practical obstacle is that the existence of a *formal* modeling of the software system cannot be assumed in general. What we may reasonably presuppose for routine industrial development, rather, is a semi-formal, easy to learn, specification language. We thus employ and extend the standard UML as our notation for modeling SAs. Moreover, the applied methods should be *timely* (even incomplete models should allow to start analysis), and *tool supported* (automated tool support is fundamental for strongly reducing analysis costs).

We here interpret the term "Quality of Software Architecture" as implying that the SA specification becomes the main artifact used to analyze the entire system. We hence propose an *SA-centric approach for producing quality systems*: such an approach starts from the Natural Language (NL) definition of requirements, analyzes them with an automated tool-supported method, continues up to the specification and validation of an architectural model, and finally selects architectural-level test cases that are then used to test the implementation conformance with respect to the SA specification. Such process is meant to address the requirements imposed by items *i*), *ii*) and *iii*) above, and is conceived to be tool supported, model-based, and applicable even on incomplete specifications.

We do not claim that our proposal is a consistent and complete process for SA-centric software development. On the contrary, we emphasize that our proposed approach addresses specifically the analysis aspects of development, in order to eventually produce a quality system with validated properties. Our aim is to develop a seamless analysis process centered on the SA, which proceeds side by side along with development. We do not deal here with the activities entitles to the production of the artifacts, such as requirements or SA design, which we analyze.

Note that to realize the proposed analysis process we have taken a sort of "component-based" approach, in that instead of producing the supporting tools from scratch, we exploit tools and technologies already available, by adapting and suitably assembling them. This in the long term may not be the most effective solution, in the sense that perhaps other more suitable tools could instead be adopted. However, our interest is rather in analyzing the feasibility and advantages of a seamless analysis centered on SA, rather than in implementing a highly efficient approach. Therefore, we leverage on results produced in previous research projects aimed at addressing specific stages of development. The emphasis of this paper is in the attempt of combining specific analysis techniques into an integrated SA-centric analysis process.

The paper is organized as follows: after introducing in Section 2.1 the most significant SA-centric development processes proposed so far, we outline our proposed analysis approach in Section 2.2. We introduce the exploited approaches and their tool support in Section 3, then we show some initial results and issues in integrating these approaches and tools (Section 4). Conclusions and future work are briefly drawn in Section 5.

2 SA-Based Analysis Process: State of the Art and Our Proposal Outline

The dependability of a software system strongly depends on the software development process followed to produce it, thus, on the selected activities and on how the system evolves from one activity to another. Several papers advocate the use of architectural specifications to identify deficiencies soon in the process, but only few systematic processes have been proposed to implement such requirement. Currently, automating the analysis stages that accompany the process of transiting from requirements to architectures and from architectures to code still remains subject of research.

2.1 Related Work

SA-based development processes, SA integration with the other development phases, and SA-based analysis are the main topics related to our work. We here survey some relevant works on each topic.

Software Architecture-Based Development Process: The Unified Software Development Process (UP) [24], is an iterative and incremental, use case driven, and architecture-centric software process. In the UP, an architecture specification is not provided in a single phase in the development process, but it embodies a collection of views created in different phases. In the Hofmeister, Nord and Soni' book on applied software architecture [22], an architectural specification is composed by four different views: *conceptual*, *module*, *execution*, and *code*. In the "architecture-based development" proposed in [10], the authors present a description of an architecture-centric system development where a set of architecture requirements is developed in addition to functional requirements. The process comprises six steps: *i*) elicit the architectural requirements (expressing quality-based architectural requirements through quality-specific scenarios), *ii*) design the architecture (using views), *iii*) document the architecture (considered as a crucial factor in the SA success), *iv*) analyze the architecture (with respect to quality requirements), *v*) realize and finally, *vi*) maintain the architecture. The requirements are informally validated, the architecture is specified through different views, the architecture is analyzed through the Architecture Tradeoff Analysis Method (ATAM). This process differs from traditional development in which SA is used for design perspectives since it focuses on capturing system qualities for analysis. KobrA (Component-based Application Development) [5] is a software process- and product-centric engineering approach for software architecture and product line development. KobrA uses a component-based approach in each phase of the development process, it uses UML as the modelling language, it exploits a product line approach during components development and deployment, and it makes a clear distinction between the Framework Engineering stage and the Application Engineering stage (typical of product line developments).

For completeness, we also cite TOGAF [2], a detailed method and a set of guidelines to design, evaluate and build enterprise architectures and Catalysis [34], a methodology that makes special emphasis on component based development.

Software Architecture Integration with other Phases: To increase the overall system quality, two main topics need to be carefully addressed: *i*) how different phases in the software process should be mutually related, *ii*) and how results from one phase should be propagated down to the other phases.

Even if such processes take into consideration the different stages in the software process, they do not specify how requirements, architectures and implementation have to be mutually related. This is still an open problem advocated and investigated by many researchers [3,29,21]. In [3,29], ways to bridge the gap between requirements and software architectures have been proposed. Egyed proposes ways to trace requirements to software architecture models [21] and software architecture to the implementation [26].

Software Architecture-Based Analysis: In the research area of Software Architecture much work is oriented on architectural specification and analysis [11,33]. Many analysis methods and tools, based upon formal Architecture Description Languages (ADLs), have been proposed in order to perform analysis and model checking [25], behavioral analysis of concurrent systems [14], deadlock detection [4,15], architectural slicing [35] and testing [27]. Today, only few ADLs with their analysis capabilities are still supported. Many UML-based approaches have been proposed in order to make the adoption of SA specification and analysis feasible in industrial contexts [16].

2.2 Our Proposal Outline

Previous SA-based processes have provided guidelines on how to move from one stage to another and on how to analyze a software architecture. However, techniques and tools which allow to formalize and implement such techniques are not illustrated.

What we envision, instead, is a tool supported, SA-centric analysis process which allows to specify and analyze requirement specifications, specify the architecture and validate its conformance to requirements. Once the SA has been identified, both system tests and the system implementation are derived from it. Test specifications are identified from the architectural model (rather than directly from requirements) thus unifying some design steps, guaranteeing consistency of the whole project and, virtually, eliminating the need of review processes. During test execution, if faults are identified, the only artifact to be revised is the implementation (since the SA has been proved to be correct). Figure 1 summarizes this process (the labels A,B,C and D have been introduced in order to track correspondence with Figure 4). Both requirements, architectures and implementation are revised whenever specification or conformance errors are detected. The entire analysis process is driven by model-based specifications. The validation steps are tool supported.

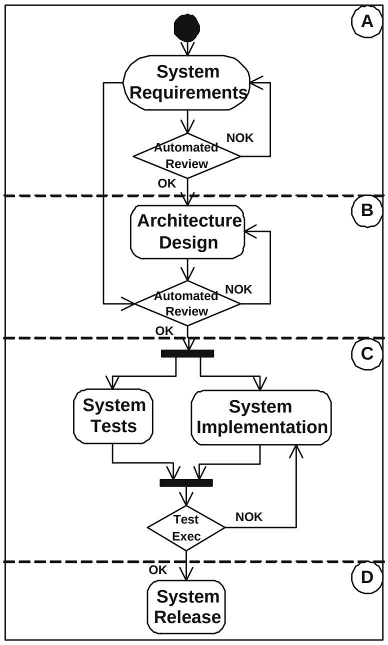


Fig. 1. Software Architecture-Based Analysis Process

In the following of this paper we present three different approaches which are the results of previous research projects carried on in independent way. By an appropriate integration approach, these three tools can support the entire analysis process we propose. The three tools are: QuARS (Quality Analyzer for Requirements Specifications), which supports the analysis of natural language requirements in a systematic and automatic way; ModTest, a framework for SA-based model-checking driven testing, which allows to validate the SA model with respect to requirements and to select test procedures; and CowSuite which supports the generation of executable test procedures from UML diagrams, and their management via weights coefficients. The execution of such test procedures allows us to analyze the system conformance with respect to the SA.

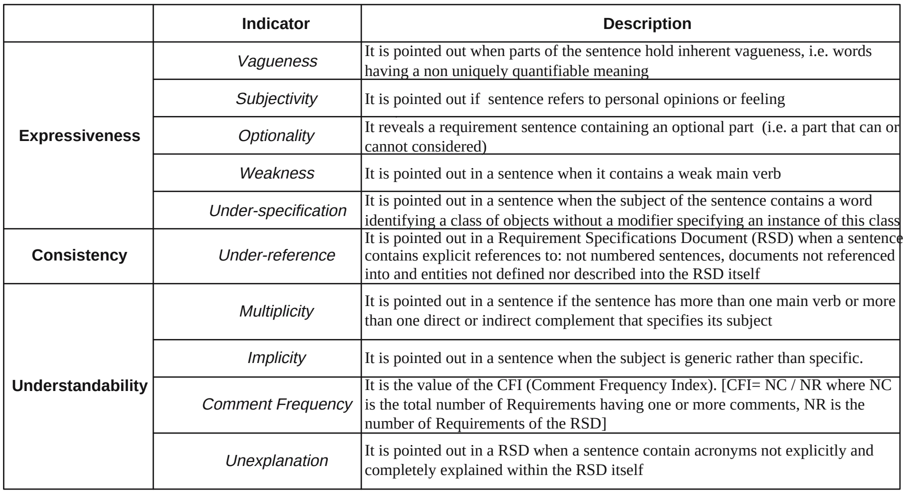
Indeed, other approaches and analysis tools can be selected for integration. However, in this initial attempt to create an architecture-centric approach for producing quality systems we selected those we are more familiar with.

3 The Three Approaches in Isolation

3.1 QuARS

The tool QuARS (Quality Analyzer for Requirements Specifications) provides a systematic and disciplined analysis process for Natural Language (NL) requirements. The application of linguistic techniques to NL requirements allows their analysis from a lexical, syntactical or semantic point of view. For this reason it is proper to talk about, e.g., lexical non-ambiguity or semantic non-ambiguity rather than non-ambiguity in general. For instance, a NL sentence may be syntactically non-ambiguous (in the sense that only one derivation tree exists according to the applicable syntactic rules) but it may be lexically ambiguous because it contains wordings, which do not have a unique meaning.A Quality Model is the formalization of the definition of the term "quality" to be associated to a type of work product. The QuARS methodology performs the NL analysis by means of a lexical and syntactic analysis of the input file in order to identify those sentences containing defects according to the quality model in Figure 2 [19]. When the Expressiveness analysis is performed, the list of defective sentences is displayed by QuARS and a log file is created. The defective sentences can be tracked in the input requirements document and corrected, if necessary. Metrics measuring the defect rate and the readability of the requirements document under analysis are calculated and stored. The tool QuARS(Quality Analyzer for Requirement Specifications) provides a systematic and disciplined analysis process for NL requirements. The development of QuARS has been driven by the objective to realize a tool modular, extensible and easily usable. QuARS is based on the quality model, shown in Figure 2 [19]. It is composed of a set of high-level quality properties for NL requirements to be evaluated by means of syntactic and structural indicators directly detectable and measurable looking at the sentences in requirement document. These properties can be grouped in three categories:

- **Expressiveness:** it includes those characteristics dealing with incorrect understanding of the meaning of the requirements. In particular, the presence of ambiguities in and the inadequate readability of the requirements documents are frequent causes of expressiveness problems.
- **Consistency:** it includes those characteristics dealing with the presence of semantic contradictions in the NL requirements documents.
- **Understandability:** it includes those characteristics dealing with the lack of necessary information within the requirements document.



	Indicator	Description
Expressiveness	*Vagueness*	It is pointed out when parts of the sentence hold inherent vagueness, i.e. words having a non uniquely quantifiable meaning
	Subjectivity	It is pointed out if sentence refers to personal opinions or feeling
	Optionality	It reveals a requirement sentence containing an optional part (i.e. a part that can or cannot considered)
	Weakness	It is pointed out in a sentence when it contains a weak main verb
	Under-specification	It is pointed out in a sentence when the subject of the sentence contains a word identifying a class of objects without a modifier specifying an instance of this class
Consistency	*Under-reference*	It is pointed out in a Requirement Specifications Document (RSD) when a sentence contains explicit references to: not numbered sentences, documents not referenced into and entities not defined nor described into the RSD itself
Understandability	*Multiplicity*	It is pointed out in a sentence if the sentence has more than one main verb or more than one direct or indirect complement that specifies its subject
	Implicity	It is pointed out in a sentence when the subject is generic rather than specific.
	Comment Frequency	It is the value of the CFI (Comment Frequency Index). [CFI= NC / NR where NC is the total number of Requirements having one or more comments, NR is the number of Requirements of the RSD]
	Unexplanation	It is pointed out in a RSD when a sentence contain acronyms not explicitly and completely explained within the RSD itself

Fig. 2. Quality Model

The Functionalities provided by QuARS are:

1. Defect identification: QuARS performs a linguistic analysis of a requirement document in plain text format and points out the sentences that are defective according to the expressiveness quality model. The defective sentences can be automatically tracked on the requirement document to allow their correction.
2. Requirements clustering: The capability to handle collections of requirements, i.e. the capability to highlight clusters of requirements holding specific properties, can facilitate the work of the requirements engineers.
3. Metrics derivation: QuARS calculates metrics (The Coleman-Liau Formula and the defect rate) during the analysis of a requirements document.

QuARS satisfies the requirements identified in Section 1. The Quality Analysis by QuARS allows the automatic generation of validated requirements documents that will be used for the definition of Functional Properties *(goal i)*. The quality analysis is supported by QuARS*(goal ii)*. QuARS can be applied even on partial specifications, it does not require formal modeling and it is tool supported *(goal iii)*.

3.2 ModTest

ModTest is an architecture-centric approach for model-checking based testing [13]. It is based on the idea that SA-based testing and exhaustive analysis through model-checking can be successfully integrated in order to produce highly-dependable systems. In particular, ModTest works in a specific context, where the SA specification of the system is available, some properties the SA has to respect are identified, and the system implementation is available.

Model-checking techniques are used to validate the SA model conformance with respect to selected properties, while testing techniques are used to provide confidence on the implementation fulfillment to its architectural specification. (The architecture-centric approach is graphically summarized in Figure 3).

The *advantages* we obtain are manifold: we apply the model-checking technique to higher-lever (architectural) specifications, thus governing the state-explosion problem, while applying testing to the final implementation. We check and test the architecture and the system implementation with respect to properties elicited from requirements. Moreover, the test case selection phase is driven by both the requirements and the architectural model.

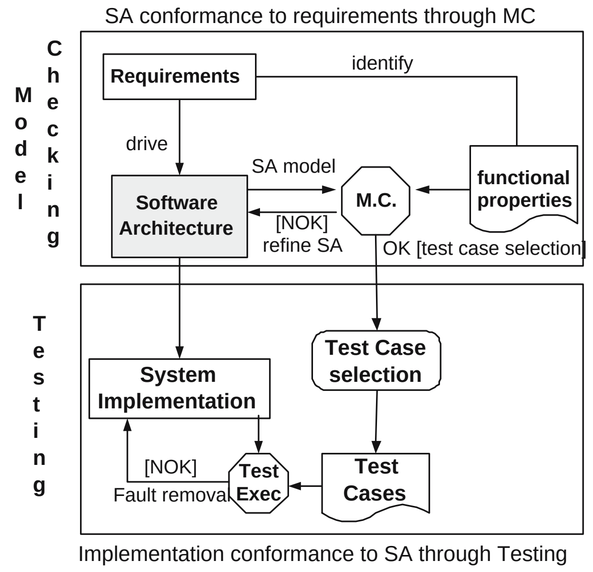


Fig. 3. ModTest

ModTest has been implemented by using two different technologies: Charmy [1], a framework which allows to model check architectural specifications with respect to selected properties, and TeStor [32] an algorithm which permits to extract test specifications from Charmy outputs.

Following the Charmy approach, the software architect specifies the system architecture, in an UML-based diagrammatic way. Both a structural and a behavioral viewpoints are taken into account. The SA topology is specified in terms of components, connectors and relationships among them, where components represent abstract computational subsystems and connectors formalize the interactions among components. The internal behavior of each component is specified in terms of state machines. Once the SA specification is available, a translation feature is used to obtain from the model-based SA specification, a formal executable prototype in Promela (the specification language of SPIN)[23].

On the generated Promela code, we can use the SPIN standard features to find, for example, deadlocks or parts of states machines that are unreachable. Behavioral properties modeled using the familiar formalism of sequence diagrams, are automatically translated into Büchi automata (the automata representation for LTL formulae). Each sequence diagram represents a desired behavioral property we want to check in the Promela (architectural) prototype. (More details may be found on [31]).

Whenever the SA specification has been validated with respect to certain requirements, the SA itself is used to identify test procedures to be used to provide confidence on the implementation fulfillment to the architectural specification. TeStor (our test sequence generator algorithm) gets in input the behavioral model of the software under test (i.e., the SA behavioral model) and a set of test generation directives (inSD), and outputs a set of sequence diagrams (outSD) representing the test specification. An inSD represents the property previously checked with Charmy (used as a test directive) while an outSD contains the sequence of messages expressed by the inSD, enhanced/completed with information gathered by the components' state machines. (More details may be found on [32]).

The Charmy approach is tool supported: via a *topology editor*, the SA topology is specified in terms of components, connectors and links. The *thread editor* allows to specify the internal behavior of each component. The *sequence editor* allows to draw sequence diagram representing desired behavioral properties we want to check. TeStor has been implemented as a plugin component. The user selects the inSD from a list of scenarios, then she runs the TeStor algorithm by activating the plugin module and the corresponding outSDs are automatically generated. The tool is available on [1].

ModTest satisfies the requirements identified in Section 1. Model-checking and testing are integrated in a unique process, thus allowing to reuse model-checking results for generating test specifications (goal *i*). The approach is tool supported, from SA specification to test procedures generation (goal *ii*). ModTest can be applied even on partial specifications, it does not require formal modeling, and it is tool supported (goal *iii*).

3.3 Cow_Suite Test Strategy

Cow_Suite [8] is a methodology originally conceived for test planning and generation, since the early stages of system analysis and modeling, based on a UML design. The name Cow_Suite stands for *COW*test plu*S* *UIT* Environment, and as the name implies it combines two original components:

1. CowTest (Cost Weighted Test Strategy) is a strategy for test prioritization and selection;
2. UIT (*U*se *I*nteraction *T*est) is a method to derive the test cases from the UML diagrams.

These two components work in combination, as CowTest helps decide which and how many test cases should be planned from within the universe of testcases that UIT could derive for the system under consideration. In the remainder of this section we very shortly present the main characteristics of these two components. Further information can be retrieved from [17].

CowTest: Essentially, the CowTest strategy considers the diagrams composing the design and, by using their mutual relationships, organizes them into a hierarchical structure. More precisely, considering in particular the Actors, the Use Cases (UCs) and the Sequence Diagrams (SDs), they are first organized in an oriented graph called the Main Graph, which is then explored by using a modified version of the Depth-First Search algorithm for producing a forest of Trees, which constitute the basic hierarchical structures of the CowSuite approach. Each level in each tree evidences a different degree of detail of the system functionalities and represents for testing purposes a specific integration stage.

The central feature of CowTest then is that the nodes of the derived trees are annotated by the test manager with a value, called the weight, belonging to the [0,1] interval and representing its relative "importance" with respect to the other nodes at the same level: the more critical a node the higher its weight (the tool by default provides initially uniform weights which can be easily modified). Different criteria can be adopted to define what importance means for test purposes, e.g., the component complexity, or the usage frequencies, such as in reliability testing. Oftentimes, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers.

The basic idea behind CowTest is requiring the test manager to make explicit these criteria and supporting them with a systematic tool strategy to use such information for test planning.

In the context of this proposed architecture-centric approach, it is our intention to enrich the phase of requirement analysis (normally focused on modeling concerns), also with the managerial concern of introducing an evaluation of the importance of the analysed requirements. Clearly, in this case we will consider weights which reflect architectural concerns. The CowTest approach fits quite well into the proposed process, as an help to manage the apportioning of testing effort onto the architectural specification.

For each node (which will correspond to a SD, i.e. to the description of a system architectural behaviour) the final weight is then computed as the product of all the nodes weights on the complete path from the root to this node. These final weights can be used for choosing amongst the tests to execute, in two different manners:

1. by fixing the number of test cases: then CowTest selects the most suitable distribution of the test cases among the functionalities on the basis of the leaves weights.
2. by fixing a functional test coverage as an exit criterion for testing. In this case CowTest can drive test case selection, by highlighting the most critical system functionalities and properly distributing the test cases.

UIT: Largely inspired to the Category Partition Method [30], UIT systematically constructs and defines a set of test cases for the Integration Testing phase by using the UML diagrams as its exclusive reference model and without requiring the introduction of any additional formalisms.

UIT was originally conceived [7] for UML-based integration testing of the interactions among the objects, or objects groups, involved in a SD. For each given SD, UIT automatically constructs the Test Procedures. A Test Procedure instantiates a test case, and consists of a sequence of messages, and of the associated parameters. UIT is an incremental test methodology; it can be used at diverse levels of design refinement, with a direct correspondence between the level of detail of the scenarios descriptions and the expressiveness of the Test Procedures derived. For each selected SD, the algorithm for Test Procedures generation is the following:

Define Messages_Sequences: Observing the temporal order of the messages along the vertical dimension of the SDs, a Messages_Sequence is defined considering each message with no predecessor association, plus, if any, all the messages belonging to its nested activation bounded from the *focus of control* [20] region.

Analyze possible subcases: the messages involved in a derived Messages_Sequence may contain some feasibility conditions (e.g., if/else conditions). In this case a Messages_Sequence is divided in subcases, corresponding to the possible choices.

Identify Settings Categories: the Settings Categories are the values or data structures that can influence the execution of a Messages_Sequence.

Determine Choices: for each Message choices represent the list of specific situations or relevant cases in which the messages can occur; for the Settings Categories, they are the set or range of input data that parameters or data structures can assume.

Determine Constraints among choices: to avoid meaningless or even contradictory values of choices inside a Messages_Sequence, constraints among choices are introduced.

Derive Test Cases: for every possible combination of choices, for each category and message involved in a Messages_Sequence a test case is automatically generated.

Combined Approach: The combination of CowTest and UIT seems to well satisfy requirements posed in the Introduction. The approach is *(goal ii)* tool-supported with the CowSuite tool (some adaptation is of course needed); its two features of trading-off between extensive testing and limited effort, by means of CowTest weights, and of using standard UML diagrams, without requiring any extra annotation for derivation of test procedures, make it *(goal iii)* purposely conceived for industrial usage. Finally, with regard to req *(goal i)*, we intend (see Figure 4) to combine the usage of CowTest with the requirement analysis stage to find appropriate Architectural relevant weights, and to use as an input the SD produced by ModTest.

4 Integration

4.1 Methodologies Integration

In this section we instantiate the analysis process in Figure 1 via the integration of the three methodologies previously introduced into a smooth architecture-centric process. The result is the process sketched in Figure 4:

a1) Requirements are specified with commercial tools, e.g., RequisitePro, Doors, AnalystPro. Natural Language (NL) requirement documents are automatically produced (by commercial tools such as SoDA, DOORSRequireIT) in the form of .txt or .doc files;

a2) QuARS takes in input a NL requirement document, makes a quality analysis and gives in output a log file which lists the indications of requirements containing defects or not. If defects are detected by QuARS (NOK arrow), it points to some defects lowering the quality of the requirements document, a refinement activity (*Requirements Revision*) is initiated, followed by another quality analysis step. At this stage, use-cases can be identified and a first distribution of CowTest weights is performed. If defects are not detected (OK arrows), the architectural specification can start on the basis of the approved requirement document.

a3) NL Requirements are used to define some high-level functional properties that the system must satisfy.

a4) Requirements are used to drive the Software Architecture (SA) definition, that will be designed into Charmy;

a5) In parallel the NL Functional Properties are formalized in the appropriate form for Charmy (for example PSC [6]).

a6) By using Charmy we have an easy to use, pratical approach to model, and check architectural specifications;

a7) Whenever the SA specification does not properly implement selected requirements/properties (NOK arrows), the SA itself needs to be revised (*SA Revision*). Thanks to the model-checker outputs, and following the Charmy iterative process, we identify specific portions of the SA which need to be better specified. Thus, starting from a first system SA high level abstraction, which identifies few main subsystems and a set of high level properties, we obtain a model that can be validated with respect to significant high level behaviors. This first validation step allows us to gain confidence on the global design architectural choices and to identify portions of the SA that might need further and deeper investigation. The subsequent steps are to zoom into the potentially problematic subsystems by identifying implied sub-properties;

a8) The sequence diagrams representing the properties validated through Charmy is the input for TeStor. Through the TeStor algorithm we identify traces of interest for testing the whole system or just a relevant subsystem. The TeStor output represents test sequence specifications (outSD);

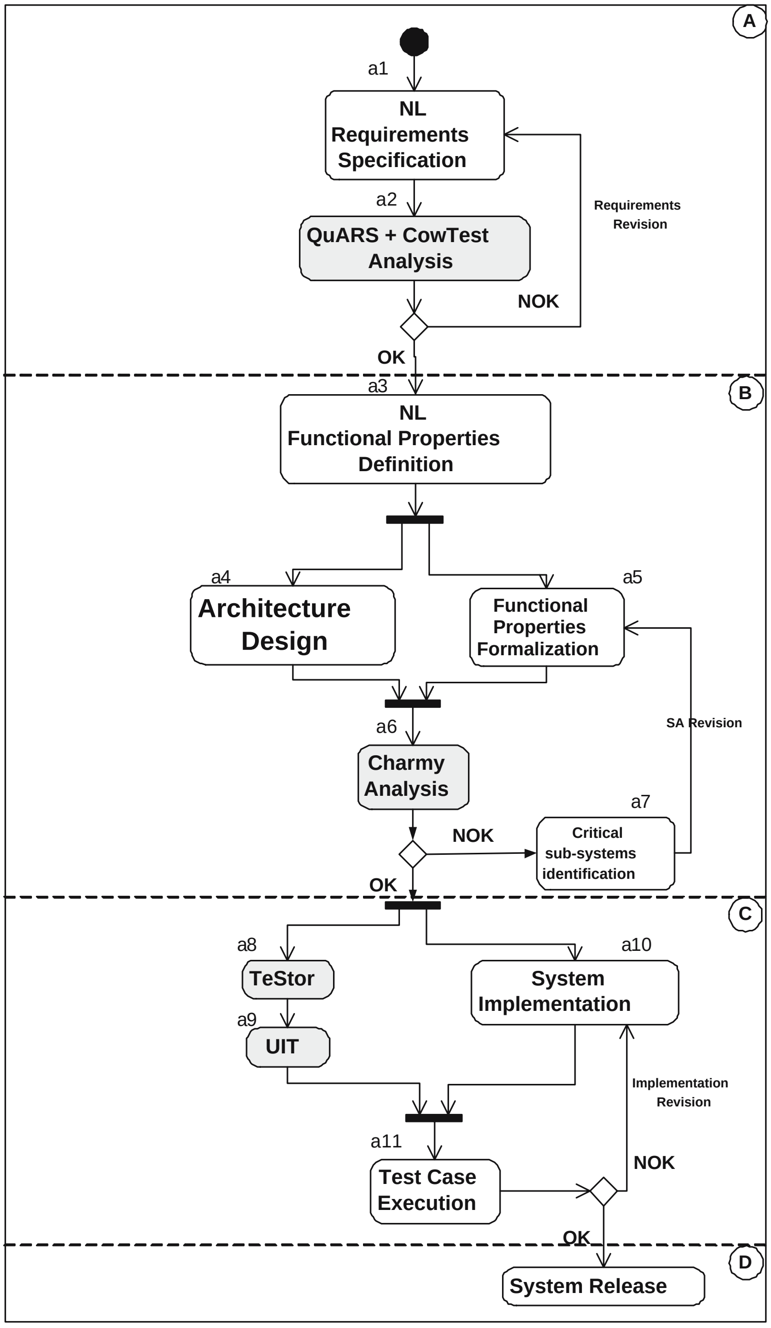


Fig. 4. Instantiating the SA-based Analysis Process

a9) The TeStor output (outSD) represents the input for the UIT methodology. UIT constructs the Test Procedures (TP) using solely the information retrieved from the outSD. A Test Procedure is a set of detailed instructions for setting-up, executing, and evaluating the results of a given test case. The number of Test Procedures associated with each outSD is assigned on the basis of the test effort distribution made according to the CowTest procedures.

a10) In parallel with the testing process, the SA is implemented;

a11) In the last step, the Test Procedures are executed on the system code. If the system testing doesn't introduce errors the system is released (OK arrows) otherwise, the system implementation has to be revised (NOK arrows).

4.2 Tools Integration

In order to perform a complete analysis of a software architecture we need an easy to use and automatic framework that starting from requirements specification, is able to validate the SA from qualitative points of view (Requirements, SA and Testing validation). For this aim the Charmy tool architecture, that is shown in Figure 5, can be of help. Taking a look to the Charmy `Core` macro-component,

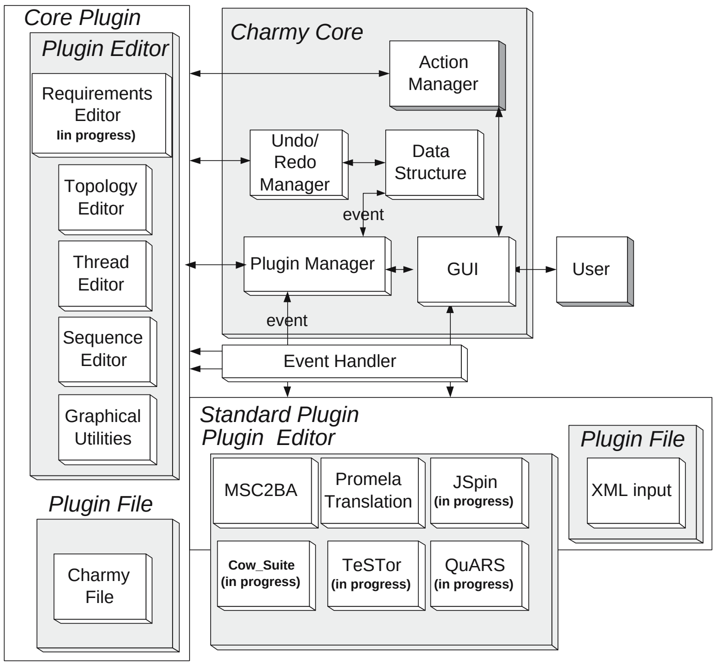


Fig. 5. The Charmy Plugin Architecture

it is composed by the `Data Structure` component, the `Plugin Manager` which allows to handle the plug of a new component in the core system, the `GUI` which receives stimuli by the users, and activates the `Action Manager` and the `Event Handler`.

The `Core Plugin` meta-component contains a set of core plugs, which allow to edit the NL system requirements, software architecture topology, the state machines and the scenarios (functional properties).

The `Standard Plugin` contains a set of standard plugs, which allow to implement the translation from sequence diagrams to Büchi automata and from state machines to Promela code. Moreover, this component contains the new plug `XML Input` and will contain the others.

The plugin SA of Charmy will allow the introduction of new features. We are currently adapting the existing tools, CowSuite, TeStor and QuARS, as plugins to insert them in Charmy, so as to have a complete framework able to specify and analyze a software architecture along its life-cycle process. Indeed, the tools integration requires a certain effort, since existent tools (QuARS written in Tcl/Tk and CowSuite in Visual Basic) need to be ported to Java.

5 Conclusions and Future Work

As rightly claimed in the QoSA 2005 CFP, progress in software quality research is to be expected by joining research efforts of several groups and from disciplines which have proceeded separately so far. Our modest proposal is meant exactly as a little step towards this ambitious vision, by proposing the combination of a tool for requirements analysis, a tool for SA model-checking and model-based testing, and a tool for test planning and derivation based on a UML design.

In future work we wish to apply the entire analysis process to a selected case study, in order to improve the integration analysis and empirically evaluate main advantages and limitations. Regarding tool support, we plan to port existent tools inside the Charmy plugin architecture.

References

1. CHARMY Project. Charmy Web Site. http://www.di.univaq.it/charmy, 2004.
2. TOGAF 8: The Open Group Architecture Framework. `http://www.opengroup.org/architecture/togaf/`, 2005.
3. STRAW '03: Second Int. Workshop From Software Requirements to Architectures, May 09, 2003, Portland, Oregon, USA.
4. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, July 1997.
5. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-Based Product-Line Engineering with UML*. Addison-Wesley, 2001.
6. M. Autili, P. Inverardi, and P. Pelliccione. A graphical scenario-based notation for automatically specifying temporal properties. Technical report, Department of Computer Science, University of L'Aquila, 2005.

7. F. Basanieri and A. Bertolino. A Practical Approach to UML-based Derivation of Integration Tests. In *Proceeding of QWE2000, Bruxelles*, November 20-24.
8. F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. In *Proceeding of UML 02, LNCS 2460, Dresden, Germany*, pages p. 383–397.
9. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, second edition.* SEI Series in Software Engineering. Addison-Wesley Professional, 2003.
10. L. Bass and R. Kazman. Architecture-Based Development. Technical report, Carnegie Mellon, Software Engineering Institute, CMU/SEI-99-TR-007, 1999.
11. M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods.* SFM-03:SA Lectures, LNCS 2804, 2003.
12. R. J. Bril, R. L. Krikhaar, and A. Postma. Architectural Support in Industry: a reflection using C-POSH. *Journal of Software Maintenance and Evolution*, 2005.
13. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In *Proc. International Testing Methodology workshop*, Lecture Notes in Computer Science, LNCS, vol. 3236, pp. 351 - 365 (2004), October 2004.
14. R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable dynamic plugin systems. In *FASE*, pages 129–143, 2004.
15. D. Compare, P. Inverardi, and A. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. In *Science of Computer Programming*, pages 33(2):101–131, February 1999.
16. V. Cortellessa, A. D. Marco, P. Inverardi, H. Muccini, and P. Pelliccione. Using UML for SA-based Modeling and Analysis. In *Int. Workshop on Software Architecture Description & UML. Hosted at the Seventh International Conference on UML Modeling Languages and Applications*, Lisbon, Portugal, October, 11-15 2004.
17. CowSuite. http://www.isti.cnr.it/researchunits/labs/se-lab/software-tools.html.
18. D. Garlan. Software Architecture. In *Encyclopedia of Software Engineering, John Wiley & Sons*, 2001.
19. S. Gnesi, G. Lami, G. Trentanni, F. Fabbrini, and M. Fusani. An Automatic Tool for the Analysis of Natural Language Requirements. In *Computer Systems Science & Engineering Special issues on Automated Tools for Requirements Engineering*, volume Vol 20 No 1, 2005.
20. O. M. Group. OMG/Unified Modelling Language(UML) V2.0, 2004.
21. P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures with Intermediate Models. *Journal for Software and Systems Modeling (SoSyM)*, accepted for publication, 2004.
22. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture.* Addison-Wesley, 1998.
23. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, September 2003.
24. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, Object Technology Series, 1999.
25. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *First Working IFIP Conference on Software Architecture, WICSA1*, 1999.
26. N. Medvidovic, P. Grünbacher, A. Egyed, and B. Boehm. Bridging Models across the Software Life-Cycle. *Journal for Software Systems (JSS)*, 68(3):199–215, December 2003.

27. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.
28. G. Mustapic, A. Wall, C. Norstrom, I. Crnkovic, K. Sandstrom, and J. Andersson. Real world influences on software architecture - interviews with industrial system experts. In *Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004*, pages 101–111, June 2004.
29. B. Nuseibeh. Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115–117, March 2001.
30. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. In *Communications of the ACM*, pages pp. 676–686.
31. P. Pelliccione, P. Inverardi, and H. Muccini. Charmy: A framework for Designing and Validating Architectural Specifications. Technical report, Department of Computer Science, University of L'Aquila, May 2005.
32. P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. Deriving Test Sequences from Model-based Specifications. In *Proc. Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, Lecture Notes in Computer Science, LNCS 3489, pages 267–282, St. Louis, Missouri (USA), 15-21 May, 2005 2005.
33. D. J. Richardson and P. Inverardi. ROSATEA: International Workshop on the Role of Software Architecture in Analysis E(and) Testing. In *ACM SIGSOFT Software Engineering Notes, vol. 24, no. 4,*, July 1999.
34. D. D. Souza and A. C. Wills. Objects, components, and frameworks with UML. The Catalysis approach. Addison-Wesley, 1998.
35. J. Zhao. Software Architecture Slicing. In *Proceedings of the 14th Annual Conference of Japan Society for Software Science and Technology*, 1997.

A Model-Oriented Framework for Runtime Monitoring of Nonfunctional Properties

Kenneth Chan[1], Iman Poernomo[1], Heinz Schmidt[2], and Jane Jayaputera[2]

[1] Department of Computer Science,
King's College London, Strand, London, WC2R2LS, United Kingdom
{chankenn, iman}@dcs.kcl.ac.uk
[2] School of Computer Science and Software Engineering,
Monash University, Caulfield East, Victoria, Australia 3145
{janej, hws}@csse.monash.edu.au

Abstract. It is now recognized that nonfunctional properties are important to practical software development and maintenance. Many of these properties involve involving time and probabilities – for example, reliability and availability. One approach to ensuring conformance to nonfunctional requirements is the use of runtime monitoring. Currently, such monitoring is done in one of two ways: 1) monitoring through use of a generic tool or 2) by adding instrumentation code within system software and writing a tool to manage resulting datasets. The first approach is often not flexible while the second approach can lead to a higher development cost. In this paper, we present a flexible framework for runtime verification of timed and probabilistic nonfunctional properties of component-based architectures. We describe a Microsoft .NET-based implementation of our framework built upon the Windows Management Instrumentation (WMI) infrastructure and the Distributed Management Task Force's Common Information Model standard. We use a language for contracts based on Probabilistic Computational Tree Logic (PCTL). We provide a formal semantics for this language based on possible application execution traces. The semantics is generic over the aspects of an application that are represented by states and transitions of traces. This enables us to use the language to define a wide range of nonfunctional properties.

1 Introduction

Non-functional properties should be included as part of a software system specification. Many of these properties are statistical in nature, so their runtime verification requires some form of continuous monitoring over time. Currently, such monitoring is done in one of two ways: 1) monitoring through use of a generic tool, such as Microsoft's performance monitoring console, which logs a set of system properties according using a specific format, or 2) by adding instrumentation code within system software and writing a tool to manage resulting datasets. The problem with the first approach is that it often not flexible enough to provide data required for the verification of domain-specific properties. The problem with the second approach is that the onus is on the developer to add instrumentation and to provide a monitoring infrastructure, with the potential often ad-hoc approaches, a higher level of cost and a lack of reusability. This paper presents a framework for monitoring non-functional properties efficiently and practically. This is achieved by employing a tiered archiecture for non-functional property monitoring.

Our approach is based upon the probes and gauges approach of [4]. There, Garlan proposed an architecture-oriented approach for the runtime monitoring and dynamic adaptation of software systems. We modify this approach through utilization of monitoring features provided by the .NET framework. In our framework, the task of developing a means of monitoring a non-functional constraint is delegated to the implementation of a gauge component. This has the benefit that gauges can be reusable over similar contexts. Our approach to runtime verification can be understood as an extension of Meyer's design-by-contract approach. Meyer's general definition of a contract is a formal agreement between a system component and its clients, expressing each party's rights and obligations [7]. Through specifying contracts and monitoring contract fulfillment, trust between a component clients is modelled and achieved. Design-by-contract has been effectively employed at a fine-grained level, expressing required functional properties (as boolean valued pre- and post-conditions) on individual methods of objects.

Classical design-by-contract uses Boolean functions to express constraints. Boolean functions are not sufficient to express the wide range of nonfunctional constraints that involve probability and timing requirements. For example, the following availability constraint is difficult to write as a Boolean function: "after a request for a service, there is at least 98% probability that the service will be carried out within 2 seconds". We use a variant of Probabilistic Computational Tree Logic (PCTL) to express our contracts. This language is widely used in model checking of system designs with timed and probabilistic properties and is suitable to express constraints such as the example above.

An important aspect of our approach is its genericity over target domains. We provide a formal semantics for our contractual language, based on possible system execution traces. A domain metamodel determines what aspects of a system are to be modelled by states and state transitions within a system trace. This enables us to use PCTL to define a wide range of nonfunctional properties.

We have implemented our approach within the Microsoft .NET framework, using the Windows Management Instrumentation (WMI) framework to architect a probes-and-gauges approach. The implementation of our method for contract checking involves the Windows Management Instrumentation (WMI) API for .NET. Using WMI, a system is monitored for method calls (corresponding to time steps in PCTL assertions) and for changes in state (over which atomic statements can be verified). Over repeated executions of a system, the probability of a certain assertion holding can be determined with increasing accuracy. When the probability falls below the average set by the assertion, the contract is taken to be violated.

Our approach for contractual specification and runtime verification complements previous work by the authors [8]. That work developed a compositional approach to reliability models where probabilistic finite state machines are associated to hierarchical component definitions and to connectors. Markov chain semantics permits hierarchical composition of these reliability models. In [6], the contracts of this paper were understood with respect to that semantics for verification purposes. However, while that previous work was developed for verification of designs, this work enables constraints from a design to be carried over to an implementation for further runtime verification. This means that our method can be integrated with a model verification approach to achieve a greater level of trust.

To the best of our knowledge, no existing approach has adapted PCTL to runtime verification, and little similar work exists in probabilistic assertion checking.

This paper is organized as follows:

– Section 2 describes our language for probabilistic timed constraints and provides a formal semantics with respect to possible execution traces of a system.
– Section 3 outlines our framework for specification and monitoring of systems for conformance to constraints.
– Section 4 defines our framework implementation, explaining how our constraints are used as assertions and verified using .NET and the WMI.
– Conclusions and related work are discussed in section 5.

2 PCTL

Probabilistic Computational Tree Logic (PCTL) was devised by Hansson and Jonsson [5] as a specification language for probabilistic model checking. It is used to specify properties hold within a given time and given likelihood. The language takes time to be discrete step units. For instance, PCTL can specify a property such as: "there is a 99% probability within 20 time steps that a requested service will be carried out". It is decidable for systems whose behaviour can be modelled as probabilistic finite state machines with a truth valuation function associated with Boolean values.

We do not use PCTL for model checking, but instead as a language for making probabilistic assertions about a system. These assertions are to be verified via a monitoring program, with probabilistic constraints compared against average behaviour. In this section, we define our variant of PCTL and provide an execution trace semantics that will be used by the monitoring process.

2.1 Syntax

PCTL formulae are built from atomic propositions, the usual logical connectives (implication, conjunction and negation) and special operators for expressing time and probabilities.

We use the grammar of Fig. 1, where *atom* ranges over a set of atomic propositions, *int* is the integers and $float$ is any floating point number between

0 and 1. Formulae of the form F are called nonprobabilistic formulae and Q are called probabilistic formulae.

The informal meaning of formulae as system constraints is best understood with respect to the idea of a system that can execute a multiple number of runs. Each run is invoked by a call to the system from a client. Each execution run is considered as a series of discrete states. The transition from one state to another is determined by some system activity. The transition from one state to another should be considered as a discrete time step. The truth of a formula is determined according to the state of the system, and, in the case of probabilistic formulae, the number of runs a system has had.

Nonprobabilistic formulae consist of two kinds of formulae:

– Ordinary propositional formulae built from a set of atomic propositions. We assume that atomic propositions correspond to statements about the state of a system, and can be verified to be true or false at any point in an execution. By Boolean semantics, it is then easy to verify if compound propositional formulae are true for a state. Consequently, all propositional formulae can be viewed as assertions about a system's state.
– Formulae with timing. These formulae make statements about the way a run may evolve, given certain assumptions hold at a state.
  The informal meaning of the until statement is as follows. The statement A until B steps: s is *false* at a state when B becomes true within s time steps, but where A became false at a step before s. This is equivalent to the statement A until B steps: s is *true* when, for some $q \leq s$, assuming B is true at q, then A is true at each time step $k < q$.
  The informal meaning of the leadsto statement is as follows. The statement A leadsto B steps: s holds when, assuming A is true at a state, B will become true within s steps.

Probabilistic formulae are understood in terms of corresponding nonprobabilistic formulae with truth averaged over a number of runs. For instance, A until B steps: s prob: p is true for a number of runs, if the corresponding nonprobabilistic formula A until B steps: s is true over these runs with a probability of p. The case is similar for A leadsto B steps: s prob: p.

We use PCTL formulae to specify required properties of a system, *immediately after a client call to the system.* That is, we make our specifications with respect to the initial state of system execution runs. However, the ability to make timing constraints enables us to specify requirements about later states in a system. For example, if T holds for all states, then T leadsto OK steps: 10 is a constraint that the system will be functioning normally, OK, after 10 steps from initialization.

2.2 Domain-Specific Interpretations

We shall use PCTL to specify nonfunctional constraints within system models, which will, in turn, be used as assertions to be checked at runtime over model implementations. Models will be defined with respect to metamodels. A metamodel identifies the architectural roles and relationships necessary to specify systems and system constraints for a particular domain. It is at this metamodelling level that meaning a time step for a domain is set.

PCTL statements are understood with respect to abstract notions of a time step and application state. A time step designates a transition that distinguishes one state of the application from another state. Different domains will involve different abstract machine model that provide demarcation of application states by time step transitions. Consequently, the metamodel defines the choice of what kind of appplication states and time steps are to be used by statements.

A metamodel designer chooses a system activity that PCTL specifications are to involve, and the consequent demarcation of states and time step. The expert should also provide the set *atom* of atomic propositions, to stand a vocabulary of important boolean data views that can be verified at each state. In this way, the scope and application of PCTL as a specification language can be fixed to suit a particular domain.

Example. To see how different kinds of temporal constraints result in different notions of time step, consider the following two examples.

– It is often important to monitor a component's conformance to a protocol. The dynamic behaviour constraint is the sequence of calls required to be emitted by the component. Monitoring the component involves waiting for a method call event, and checking that the call is admissible according to the requirement. The distinguished time step here is the occurrence of an outbound method call.
– Consider the following load balancing constraint: if the number of clients that request access to objects of a class on a server rises above 50 for each instance (written as atomic proposition *AllCientsofAExceeded*), then the number of client distributed evenly and be below 50 for each instance (atomic proposition *EvenDistribution*). The balancing should be rectified within 10 client calls to the server, with a 99.9% probability. It is not imperative that the balancing occurs quickly within real time. This is because it is the rate of client calls that makes load balancing important: a high number of clients for class instances but with a low rate of calls means that balancing need not be achieved quickly. Thus, the notion of a time step for this constraint is given by client calls to the server. In PCTL, this constraint can be written

$$\begin{array}{l} AllClientsofAExceeded \\ \ \ \mathsf{leadsto}\ EvenDistribution\ \mathsf{steps}\text{: } 10\ \mathsf{prob}\text{: } .999 \end{array}$$

A balancing algorithm that would satisfy this constraint might wait for 5 client calls, before determining if a new instance of the class is required, or if the number of requesting clients has fallen to an satisfactory level.
– Consider a system in which PCTL time steps are taken to correspond to method calls to components within the system. The important system properties are "the system is in a failed state" and "the system is in a healthy

$$\begin{aligned} F := {} & atom \mid \ \textsf{not}\ F \mid F\ \textsf{and}\ F \mid F\ \textsf{or}\ F \mid \\ & F\ \textsf{until}\ F\ \textsf{steps:}\ int \mid \\ & F\ \textsf{leadsto}\ F\ \textsf{steps:}\ int \mid \\ Q := {} & F\ \textsf{until}\ F\ \textsf{steps:}\ int\ \textsf{prob:}\ float \mid \\ & F\ \textsf{leadsto}\ F\ \textsf{steps:}\ int\ \textsf{prob:}\ float \end{aligned}$$

Fig. 1. Our contractual specification language. The syntax is parametric over a set of atomic propositions *atom*. *int* ranges over integers and *float* ranges over reals between 0 and 1.

state". We denote these two properties by the propositions *Failed* and *Healthy*, respectively. Then a fault-tolerance constraint can be specified as

$$Failed\ \textsf{leadsto}\ Healthy\ \textsf{steps:}\ 2\ \textsf{prob:}\ .999$$

This states that, with a probability of .999, if the system is in a failed state after a client call, it will take two time steps to become healthy again.

In all three cases, the constraints are temporal, involving properties of application states that are demarcated by time step transitions, but with different, domain-specific choices of what designates a time step and application state.

2.3 Formal Semantics of Contracts

We can formally define the semantics of contract verification according to an abstract machine model of a system's execution.

The semantics of a system is given as a series of completed executions. Each execution is modelled as set of states, that determine if an atomic proposition is true or false.

Definition 1 (Machines, runs, states). *A* system *consists of a series of runs. A* run *is a finite series of states. A* state *is a truth valuation function of atomic formulae, atom → Boolean. The ith run $exec_i$ of a machine is represented by a function from integers to states: int → state. The integer is the number of time steps taken to arrive at a particular state. For instance, $exec_i(0)$ is the initial state of the run, where no steps have been taken yet, $exec_i(10)$ is the state of the run after* 10 *steps, and so on.*

We can formally define how a PCTL formula is true of a system using this semantics.

Atomic formulae are known to be true or false at any state of a run, by applying the state as a truth valuation function. Compound Boolean statements can also be computed for states in the obvious way. Timing formulae without probabilities are determined to be true with respect to a run, with initial requirements computed against the initial state of the run and subsequent requirements computed against following states within the timing bounds. Probabilistic formulae are computed by taking average nonprobabilistic truth values over the runs for a system.

Definition 2. *We have a recursive definition of truth, given with respect to a number of runs r. Given a formula F, $IsTrueT(F, r, t)$ holds when F is true at run r after t time steps, according to the following recursive definition:*

- *Assume F is atomic. Then $IsTrueT(F, r, t)$ holds if $exec_r(t)(F) = True$.*
- *Assume $F \equiv A$ or B. Then $IsTrueT(F, r, t)$ holds if $exec_r(t)(A) = True$ or $exec_r(t)(B) = True$.*
- *Assume $F \equiv A$ and B. Then $IsTrueT(F, r, t)$ holds if $exec_r(t)(A) = True$ and $exec_r(t)(B) = True$.*
- *Assume $F \equiv$ not A. Then $IsTrueT(F, r, t)$ holds if $exec_r(t)(A) = False$.*
- *Assume $F \equiv A$ leadsto B steps: s. Then $IsTrueT(F, r, t)$ holds when*

$$IsTrueT(A, r, t) \Rightarrow IsTrueT(B, r, j)$$
$$\text{for some } j \leq t + s$$

- *Assume $F \equiv A$ until B steps: s.*

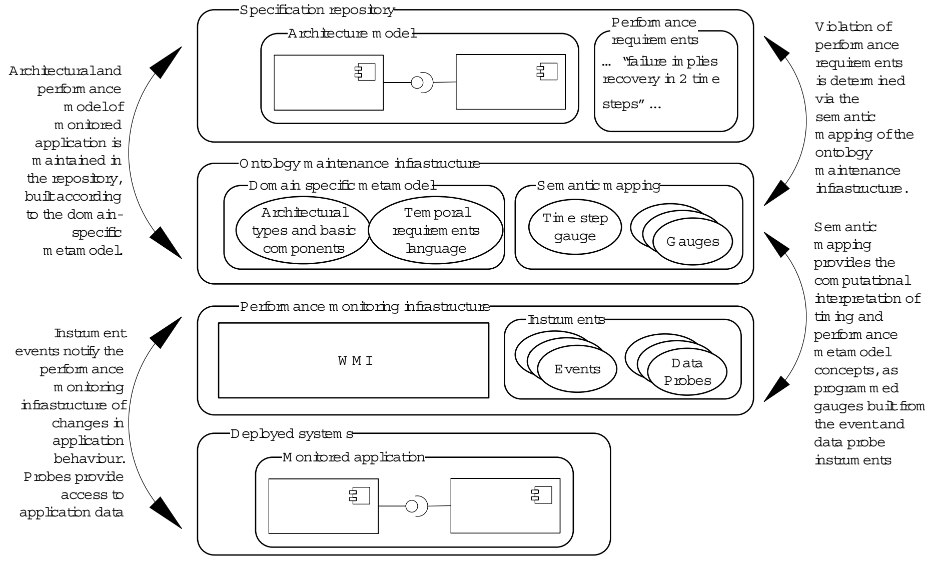
$$\text{there is a } j \leq t + s \text{ such that, for all } t \leq i \leq j,$$
$$IsTrueT(B, r, j) \Rightarrow IsTrueT(A, r, i)$$

- *Assume $F \equiv A$ until B steps: s prob: p. Then $IsTrueT(F, r, t)$ holds when*

$$\frac{\sum_{i=1}^{r} IsTrueT(A \text{ until } B \text{ steps: } s, r, t)}{r} \geq p$$

- *Assume $F \equiv A$ leadsto B steps: s prob: p. Then $IsTrueT(F, r, t)$ holds when*

$$\frac{\sum_{i=1}^{r} IsTrue(A \text{ leadsto } B \text{ steps: } s, r, t)}{r} \geq p$$


Specification repository
Architecture model
Performance requirements
... "failure implies recovery in 2 time steps" ...
Ontology maintenance infrastructure
Domain specific metamodel
Architectural types and basic components
Temporal requirements language
Semantic mapping
Time step gauge
Gauges
Performance monitoring infrastructure
WMI
Instruments
Events
Data Probes
Deployed systems
Monitored application

Architectural and performance model of monitored application is maintained in the repository, built according to the domain-specific metamodel.

Violation of performance requirements is determined via the semantic mapping of the ontology maintenance infrastructure.

Instrument events notify the performance monitoring infrastructure of changes in application behaviour. Probes provide access to application data

Semantic mapping provides the computational interpretation of timing and performance metamodel concepts, as programmed gauges built from the event and data probe instruments

Fig. 2. Framework for monitoring a system with PCTL constraints

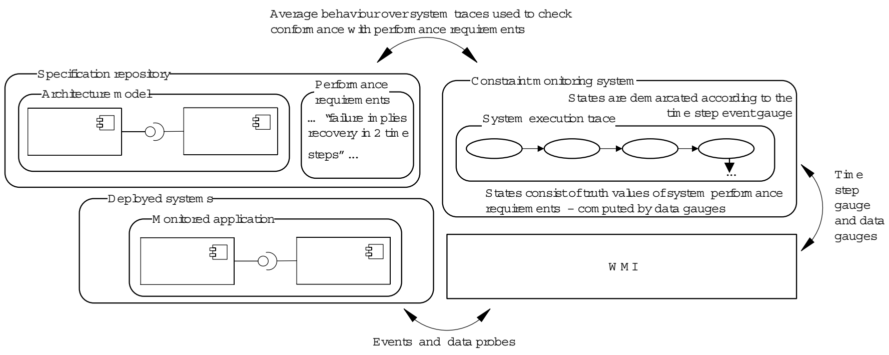

Average behaviour over system traces used to check conformance with performance requirements
Specification repository
Architecture model
Performance requirements
... "failure implies recovery in 2 time steps" ...
Constraint monitoring system
States are demarcated according to the time step event gauge
System execution trace
...
States consist of truth values of system performance requirements – computed by data gauges
Time step gauge and data gauges
Deployed systems
Monitored application
WMI
Events and data probes

Fig. 3. Monitoring process

We define truth for a formula F with initial state defined by system initialization, after r runs, $IsTrue(F, r)$ to be $IsTrueT(F, r, 0)$, the truth value of F at the initial state of the system.

3 Framework for Monitoring Timed Probabilistic Constraints

We sketch our framework for the specification and monitoring of systems with probabilistic temporal constraints. The idea is to specify structural and nonfunctional requirements of a system as an architectural model and then to link the model to the deployed system though a monitoring infrastructure, mapping nonfunctional requirements to interception instruments. The framework is depicted in Fig. 2.

The framework requires that system models are written with respect to domain-specific nonfunctional metamodels. Metamodels identify the architectural roles and relationships that are necessary to construct a model of the monitored system. A distinguishing feature of a metamodel is that it includes the PCTL for specifying temporal and probabilistic constraints.

Metamodels reside within an ontology maintenance infrastructure, a repository that includes an executable semantics of the metamodel elements. The ontology infrastructure associates metamodel elements with the system interception code, determining both how parts of a system model are to be interpreted by a system implementation and also how a system model's nonfunctional constraints can be checked via instruments over a system implementation. Importantly, our framework permits an open interpretation of nonfunctional constraints for different metamodels. In particular, the metamodel interpretation of a time step by a system activity is defined within the ontology infrastructure, this leading to a range of possible notions of timing.

By using the ontology infrastructure to link specification metamodels with instrumentation, our framework separates the concerns of building a nonfunctional metamodel that is appropriate for a particular domain from the concerns of building a monitoring system that checks constraints written in the metamodel.

The four layers of the framework are as follows.

– *A target application layer*, housing the applications to be monitored. We restrict our attention to component-based systems built with a publish-subscribe event-based middleware.

– *Performance monitoring infrastructure.* We assume a middleware technology that can perform instrumentation of target applications. The infrastructure enables the user to view the internal evolution of a target application, observing communication and interaction between its subcomponents, via user-designed instrument code that can be associated with various aspects of the application. Two kinds of instrument are used within our framework:
  - *Data probes.* These provide access to target key application data. The monitoring infrastructure can access these at any time via pull-based collection.
  - *Events.* These are triggered by designated changes in application behaviour. The infrastructure provides access to these events through loosely coupled, push-based subscription.
– *Ontology maintenance infrastructure.* This layer maintains metamodels for application modelling and provides a means of checking if model constraints are satisfied by an application. The infrastructure provides a repository of metamodels for applications in particular domains. These metamodels include a domain-specific temporal probabilistic requirements language, PCTL, as well as architectural types and relationships. Metamodels will vary according to the domain-specific choice of atomic elements of the specification language and basic components and relations of the architectural view.

  The infrastructure stores semantic mappings between metamodel elements and programmed components, called gauges, which use the instruments of the performance monitoring infrastructure to determine if model requirements are satisfied by a deployed application. The most common purpose of a gauge is to provide truth valuation functions for the specification language.

  The semantic mappings define the computational interpretation of a time step for a particular metamodel by providing a time step gauge, associated with a distinguished push-based system event from the performance monitoring infrastructure.
– *Specification repository.* This contains models of monitored systems built from a domain-specific metamodel. The system architecture, including temporal nonfunctional requirements, is defined here and associated with a deployed application. The conformance of the application to model requirements is determined via the semantic mapping of the ontology maintenance layer, enabled by the performance monitoring infrastructure.

4 Framework Implementation

We now outline our approach to implementing the layers of the framework. A model's nonfunctional constraints are checked at runtime over the execution of the model's implementation, according to the formal semantics of PCTL.

Our methodology results in a nonfunctional form of design-by-contract [7]. PCTL assertions specify constraints over an aspect of a system design, while runtime monitoring is used to check for violations of assertion contracts. In the event of a violation, a system administrator is notified, leading to further improvements in the system design and/or implementation.

4.1 Deployed Systems and the Performance Monitoring Infrastructure

In implementing the framework of Fig. 2, we restrict the deployed systems layer to .NET applications. However, by adhering to standards for interlayer interaction, it is possible to adapt our implementation to other environments.

The performance monitoring infrastructure layer is implemented using the Windows Management Instrumentation (WMI). The WMI is the core management-enabling technology built into the Windows 2000/XP/Server 2003 operating systems. WMI is an implementation of the Distributed Management Task Force's Web-Based Enterprise Management (WBEM) Common Information Model CIM standard [3], a platform-neutral language for defining application components and instruments. The WBEM standard as developed to unify the management of distributed computing environments, facilitating the exchange of data across a range of platforms.

Higher layers of the framework implementation interact with the monitoring infrastructure solely according to the CIM. Higher layers of our implementation could therefore be adapted to monitor systems operating under platforms and environments other than Windows and .NET, provided they possess WBEM compliant instrumenation technologies (for example, there are several open source platform independent Java implementations of WBEM).

The WMI enables instrumentation and plumbing through which Windows resources can be accessed, configured, managed, and monitored. The two instruments required of our framework are provided directly by WMI: system event instruments are implemented as WMI events, while data probe instruments are given as WMI objects:

– *WMI events* enable a loosely coupled, subscription-based approach to monitoring important changes in an application. Windows and .NET provide a large base set of important events that can be monitored: for example, component activation, method calls and exceptions are available for monitoring as WMI events without the need for manual instrumentation. In addition, the developer can extend the WMI event model to accommodate domain-specific events.

– *WMI objects* are .NET components that are visible to a monitoring WMI program. They provide a data-centric view of an application. A developer can instrument a program by collecting a range of important data views of the program within a WMI object. The object is instantiated and resides within the same memory space as program. However, using the WMI API, a monitoring program can make inquiries about the WMI object, permitting instrumentation of the data.

4.2 Ontology Maintenance and Specification Repository Layers

In our implementation, all domain-specific metamodels of the ontology maintenance layer are based on a UML2 representation of CIM metaschema extended with PCTL constraints. The CIM specialization is a platform-independent standard for describing the subsystems, networks, applications and services that make up an application. We use the CIM because it is a commonly accepted standard for specifying instrumentation-oriented views of applications, and because it is the metamodel used by the WMI itself, providing a common language to describe systems throughout the three upper layers of the framework.

Application specifications take the form of CIM models, annotated by PCTL constraints. By virtue of the fact that the CIM is the common language of both WMI and the ontology layer's metamodel, there is a trivial mapping between the components of a specification and their corresponding implementations. So, for instance, WMI can find and retrieve all implementation details relating to a disk drive device, if provided a CIM-based specification at the repository layer. Thus it is possible for WMI to take a model from the specification repository, and check that all elements have corresponding .NET implementations at the deployment layer.

However, because the PCTL constraints of our specifications are not standard, their computational meaning needs to be provided for WMI to perform monitoring. The ontology maintenance infrastructure of the framework includes a semantic mapping that defines how PCTL constraints should be computationally interpreted.

– The mapping should associate the atomic propositions of the domain-specific metamodel with Boolean-valued *gauges* that provide a view on system data gathered by WMI instruments at each state of the execution. Gauges are implemented as .NET functor classes that subscribe to instrument data of the monitoring layer. The mapping can be specified programatically through custom attributes or through a console interface. This provides the ability for the user to define the semantics of their atomic PCTL assertions.
– Time steps in PCTL assertions are generic over the choice of a designated regular system event. The semantic mapping specifies the choice of time step event through a special time step gauge – essentially a WMI event from the monitoring layer. WMI events range over method calls, ping heartbeat protocols, specific method call executions, or user-defined events.

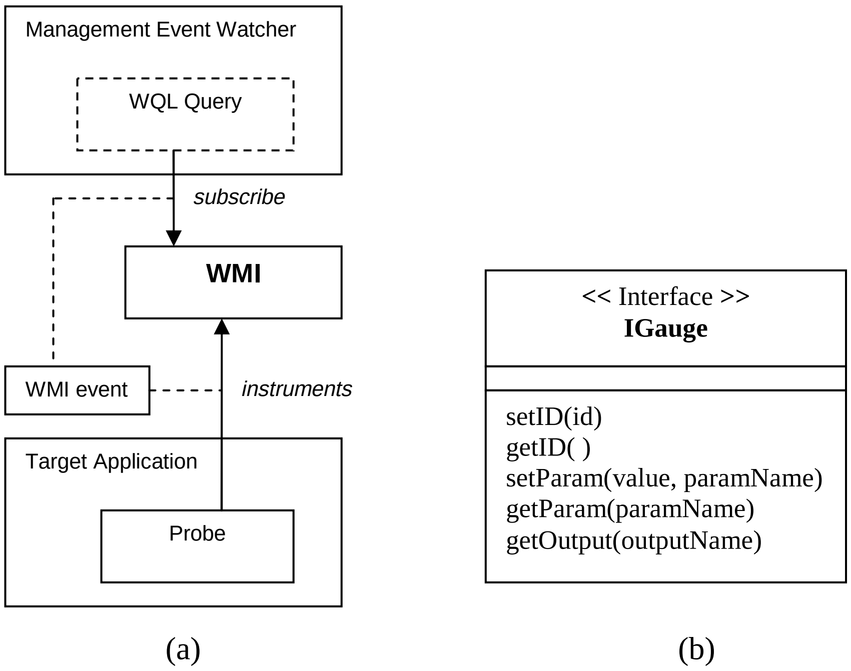


Fig. 4. (a) Probe instrumentation within the WMI implementation. (b) Generic interface for gauges in .NET.

– The point at which probabilistic assertions begin to be checked can be set by the specification expert. This is defined as an adequately sized number of sample runs against which probabilistic formulae can be checked.

4.3 Implementation Using .NET and WMI

Probes are implemented according to the architecture of Fig. 4(a). The instrumentation of data by probes are done using the *ManagementEvent* classes in .NET's *System.Management* class library. Customised events can be designed by extending the *BaseEvent* class, on recipient of these events WMI distribute copies of them to any subscribers. By default WMI events inherit a list of properties from the BaseEvent class, which contains information such as the time for which the event was created. Any local information of primitive types may also be included as *attributes* of a custom event class.

The decoupling of the monitoring system from the target system itself allows much flexibility in the implementation and usage of such system and potentially the entire system can be reused over different software developments. It is easy to extend the functionalities of the system by adding extra components whereas each component may also be modified independently. Also, by replacing the vast amounts of embedded codes that would have been needed inside the target system, the disruption caused by the data mining on the target system is kept to a bare minimum.

The implementation of gauges conform to a common interface, to enable central management by the system. The generic interface for gauges is simple. It is given in Fig. 4(b). The interface specifies the typical *get* and *set* operations of gauges. An unique identifier is included. Optionally the interface may also include a *reset* method for resetting the state in a gauge that aggregates data. Since gauges are designed to be dynamically loaded at runtime, some way has to be found to carry over the setup and configuration requirements and the input/output semantics of gauges, which are usually lost at compile time unless explicitly programmed into code.

Gauges may subscribe to WMI events using the *ManagementEventWatcher* class, which allows specification of target using WQL – a querying language derived from SQL. This publish/subscribe mechanism is provided directly by WMI hence the relatively complex task for managing interprocess data propagation is delegated to the operating platform.

In our example software, we have utilised custom attributes to identify the *types* of parameters which a gauge requires to function.We defined *Gauge RequiredParamAttribute* custom attribute to hold the information required to determine the type, as well as containing the sematical information of the parameter. For instance a gauge subscribing to a WMI event type may require a *name* to identify the relevance of an instance of the event. The respective custom attribute contains information suggesting the parameter is in the form of a *string*, and also a textual description of the parameter's semantics.

With the aid of custom attributes it is made possible to determine at rumtime the deployment and configuration requirements of gauges, hence they can be *dynamically* instantiated and configurated. Our software also defined a *GaugeType-InfoAttribute* custom attribute to identify a class as a gauge definition. This allows multiple gauge implementations to be contained within the same binary file, as well as retaining a semantical descriptions of the gauges. The interface for gauges is

A controller component, in the form of a simple GUI has also been developed. With the interface a user may load *specifications* of gauges from a compiled class library file, and instantiating these gauges by supplying it with its required parameters. Whilst these are done manually it will be possible to be made automatic if it becomes possible to map gauges directly to elements of non-functional properties.

4.4 Runtime Constraint Checking

Our approach to checking PCTL constraints is depicted in Fig. 3.

Verification of formulae can be done according to the formal semantics, given a set of runs for the system. To construct such a set, we monitor property values of WMI component views of a system. The monitoring process proceeds as follows for a single run:

1. We take an empty list, which will represent the run's states.
2. The monitor waits for an event to arrive as this corresponds to a time step.
3. Upon arrival of the event, the monitor should check truth values of each atomic proposition using the semantic map. Mappings between atomic propositions and truth values are stored in a hashtable datastructure, which is added as a node to a list of states.
4. Steps 2 and 3 are repeated until program execution finished and the list for the run is completed.

We can then check compound *nonprobabilistic* constraints with respect to a run, in the sense determined by the semantics. The monitoring system can therefore notify administrators of violations of nonprobabilistic constraints at the end of a run.

The monitoring system detects violations of *probabilistic* constraints as follows. Given a constraint A until B steps: s prob: p, the monitoring process simply records the truth value of the corresponding nonprobabilistic constraint A until B steps: s for each run. The domain-specific metamodel of the ontology maintenance layer includes a specification of the number of system runs to be executed before probabilistic constraints are checked. When this number is exceeded, the probability of A until B steps: s holding is computed for each successive run. This provides the truth value of A until B steps: s prob: p for a particular number of runs, according to the formal semantics. If the value is false, the system notifies the system administrator. If it is true, the value is then checked again after each new run. The process is the similar for leadsto constraints.

5 Related Work and Conclusions

Some work has been done before in combining nonfunctional specification with design-by-contract principles, although not with a probabilistic timed logical specification of contracts.

A similar approach to ours is [2], which provides enforcing correct usage of components through formal policies. The usage policy has to be specified, automatically generating the code at runtime enabling it to be statically verified or dynamically enforced. The usage policy consisted of activation and interaction policies of the components, whereas we use time and probabilities for formal specification at runtime.

Runtime verification and monitoring work in [1] focused on specification based monitoring and on predictive analysis of systems, specific to Java. This approach combined the system specification and the implementation by extending programming languages with specifications taken from, for instance, linear temporal logic. That work couples the code with nonfunctional constraints, so changing required constraints entails opening code. Our approach has the advantage of separating design constraints from application levels.

Frameworks similar to ours will become increasingly useful to coarse-grained system design, implementation and maintenance. Large-scale service-oriented software often has cost tarifs associated with failure to meet availability constraints. Accurately specifying and monitoring such constraints is therefore an important issue. By developing a probabilistic timed language for contracts, it becomes easier to systematically test system deployments against timing and probability constraints and to improve implementation performance. Also, our layered architecture provides a flexible and generic approach to specifying, monitoring and checking contracts that can be adapted easily to a range of different domains.

References

1. Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proceedings of 3rd International Workshop on Runtime Verification (RV'03)*, volume 89(2) of *Electronic Notes on Theoretical Computer Science*. Elsevier Science, 2003.
2. W. DePrince Jr. and C. Hofmeister. Enforcing a lips usage policy for CORBA components. In *Proceedings of 29th EUROMICRO Conference, New Waves in System Architecture, Belek-Antalya, Turkey*, pages 53–60. IEEE Computer Society Press, 2003.
3. Distributed Management Taskforce. Common Information Model (CIM) standard, 2005.
4. David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 12-14 Decemeber, 2001*. DSTC, 2001.
5. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
6. J. Jayaputera, I. Poernomo, and H. Schmidt. Uml specialization for fault tolerant component based architectures. *IDPT: Proceedings of the Seventh Biennial World Conference on Integrated Design and Process Technology*, 2, 2003.
7. B. Meyer. *Object Oriented Software Construction*. Prentice Hall, Englewood Cliffs, 2nd edition, 1997.
8. R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 66(3):241–252, 2003.

Predicting Mean Service Execution Times of Software Components Based on Markov Models

Jens Happe

Software Engineering Group*,
University of Oldenburg, Germany
jens.happe@informatik.uni-oldenburg.de

Abstract. One of the aims of component-based software engineering is the reuse of existing software components in different deployment contexts. With the re-deployment of a component, its performance changes, since it depends on the performance of external services, the underlying hardware and software, and the operational profile. Therefore, performance prediction models are required that are able to handle these dependencies and use the properties of component-based software systems. Parametric contracts model the relationship of provided and required services of a component. In this paper, we analyse the influence of external services on the service execution time applying parametric contracts and a performance prediction algorithm based on Markov chains. We verbalise the assumptions of this approach and evaluate their validity with an experiment. We will see that most of the assumptions hold only under certain constraints.

1 Introduction

Performance and scalability of performance are important influences on the quality of a software system as it is percieved by the users. In industrial applications, poor performance directly translates into financial losses. Despite its importance, performance is only considered during late development stages when at least parts of the system are implemented. This 'fix-it-later' attitude [17] often yields expensive refactorings of the system, since many performance issues are based on design decisions made during early development stages of the project. To prevent these types of mistakes, good performance prediction models are required. These models should be able to evaluate the performance of a software system during early stages of the development. They should allow the evaluation of different design alternatives and guide the software architect. In many cases, this requires a lot of additional effort from the software architect and, therefore, is unattractive. The advantages of component-based software architectures can be used to ease this task: Predefined components limit the degrees of freedom of the architecture and can provide additional knowledge. The compositional structure of component-based software architectures allows the performance prediction for the system for different levels of abstraction [11].

Since a component can be deployed in different contexts, its performance cannot be considered as a constant. The performance depends on external services, the underlying

* This work is supported by the German Research Foundation (DFG), grant GRK 1076/1 (Graduate School TrustSoft)

hardware and software, and the operational profile [3]. We need a formal model to predict a component's performance considering these parameters. In this paper, we focus on the influence of external services on the execution time of the services provided by a component and evaluate the underlying assumptions of the proposed approach.

Different approaches to compute the execution time of a service have been suggested so far [1,13,18,8,4]. Most approaches for performance prediction models use queuing networks, stochastic Petri nets, and process calculi. Many of the analysis techniques of these approaches are based on Markov chains. Several assumptions result from the usage of Markov chains. These assumptions are seldom verbalised or further investigated.

In this paper, we use a modified version of the approach described in [18, pp.352-359] in combination with parametric contracts to compute the expected service execution time based on the mathematical properties of Markov chains. The contribution of this paper is the explicit verbalisation of the assumptions of Markovian-based performance prediction models and the discussion of their consequences. Furthermore, we use our relatively easy and computation efficient approach to evaluate these assumptions and derive some of the preconditions for the application of such prediction models.

This paper is organised as follows. Section 2 briefly introduces parametric component contracts and the required mathematical background of Markov chains. Section 3 explains the approach for the computation of the expected execution time of a service based on Markov models. Section 4 verbalises the mathematical assumptions and the data required by the proposed approach. The prediction model is evaluated in section 5 using a prototypical web server developed in our group. The results are used to discuss the validity of the assumptions. Related work is discussed in section 6. Section 7 concludes the paper and presents future research ideas.

2 Fundamentals

2.1 Parametric Contracts

Component contracts lift the Design-by-Contract principle of Meyer ("if a client fulfils the precondition of a supplier, the supplier guarantees the postcondition" [12]) from methods to software components. A component guarantees a set of provided services if all required services are offered by its environment. Design-by-Contract cannot only be applied to functional properties, but also to non-functional properties. Adding Quality of Service (QoS) attributes to the contract, a component service ensures that it provides a certain QoS if its environment offers the required QoS attributes.

Static QoS contracts have the drawback, that the component developer cannot foresee all possible reuse contexts of his components in advance. Therefore, he cannot provide a component with all the configuration possibilities that will be required to fit into future reuse contexts. In several scenarios, one single pre- and postcondition will not be sufficient, since this might induce too hard restrictions so that the reusability of a component is unnecessarily limited. Even if the required quality attributes of a component are not satisfied, the component is likely to work (with an adjusted quality). The authors of [14] come to the conclusion that: "We do not need statically fixed pre- and postconditions, but *parametric contracts* to be evaluated during deployment-time."

Parametric contracts enable the computation of the quality provided by a component service in dependence on the quality of its environment.

Service effect specifications are required to describe intra-component dependencies of provided and required services. A service effect specification models external service calls executed by a component service. This can be done by a signature list or, if the execution order of the external services is important, by the set of call sequences invoked by the service. Regular expressions, finite state machines (FSMs), or Petri nets are possible methods to describe an infinite set of call sequences. Within the scope of this paper, we use Markov models for this purpose. Markov models can be considered as FSMs that are enriched with transition probabilities. Service effect specifications enable the computation of QoS attributes of the provided services (postcondition) in dependence on the QoS attributes of the required services (precondition). For QoS, the computation of the required attributes yields a solution space and, therefore, its benefits are questionable.

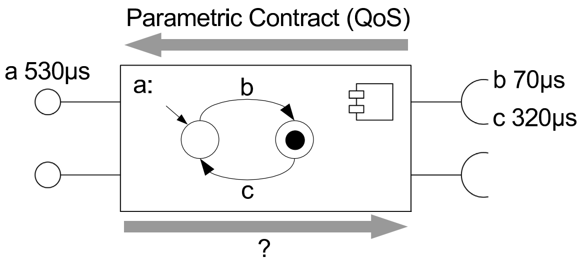


Fig. 1. Parametric Component Contract

Figure 1 illustrates parametric contracts for QoS attributes. The component shown there provides a single service called a, which requires two services b and c. The execution time of a can be calculated from the execution times of b and c. The computation requires the service effect specification of a depicted in the component. As mentioned above, the computation of the parametric contract is only applicable in one direction.

Parametric contracts have already been used to predict quality attributes of component-based software architectures, for example, reliability [14] and performance [6]. We are presenting an approach to compute the mean service execution times in dependence on the mean execution times of the required external services. To do so, we use Markov chains and Markov models as a description of the service effect specifications.

2.2 Markov Chains

Markov chains represent the core of most performance and reliability prediction models. For example, they are used to evaluate queueing networks and stochastic Petri nets. Their analysis can be either stationary or transient. In this paper, we focus on the transient models. Therefore, we introduce a modified version of Markov chains, which eases the task of reasoning about transient models. Furthermore, we clarify the relationship between Markov chains and Markov models.

Definition 1 (Markov Chain). *A stochastic process* $\{X_m,\ m = 0,1,2,3,\ldots\}$ *that takes on countable number of possible values and has the property that, given the present, the future is conditionally independent of the past, is called a Markov chain [15, p. 135].*

For the scope of this paper, we only consider Markov chains with a finite state space. The set of possible values of a process will be denoted by the state space $S = \{s_1, s_2, s_3, \ldots, s_n\}$ where n is the number of states. If $X_m = s_i$, then the process is said to be in state s_i at time m.

A Markov chain with a finite state space can be specified by a *transition matrix* P, whose entries $P(s_i, s_j)$ are given by the probability that, whenever the process is in state s_i, it will be next in state s_j, that is $P(s_i, s_j) = P\{X_{m+1} = s_j | X_m = s_i\}$. Since probabilities are non-negative and the process must take a transition into some state, we have that

$$\forall\, i,j = 1,\ldots,n:\ P(s_i, s_j) \geq 0 \text{ and for all } i = 1,\ldots,n: \quad \sum_{j=1}^{n} P(s_i, s_j) = 1.$$

Even if the stochastic process described by a Markov chain is infinite (each visit to a state must be followed by a visit to another state), some of its state are visited a finite number of times. These states are called *transient*.

Definition 2 (Transient and Recurrent States). *Consider a Markov chain* $X = \{X_m, m = 0,1,2,\ldots\}$ *with the finite state space* S*. For any two states* $s_i, s_j \in S$ *we denote* $Q(s_i, s_j)$ *the probability that, starting in state* s_i*, the Markov chain* X ever *visits* s_j*, that is* $Q(s_i, s_j) = P\left(\text{number of steps from } s_i \text{ to } s_j < \infty\right)$. *State* s_i *is said to be* recurrent *if* $Q(s_i, s_i) = 1$ *and* transient *if* $Q(s_i, s_i) < 1$ *[16,15].*

If state s_i is recurrent, then, starting in state s_i, the process will infinitely often reenter state s_i. Otherwise, if s_i is transient, the process will reenter state s_i only a finite number of times and then never return to state s_i.

Next, we introduce transient Markov chains that describe stochastic processes of an arbitrary but finite length. 'Finite length' means that the process reaches a final state after a finite number of steps and terminates. The finiteness of the process is required to model service effect specifications which, in general, describe terminating services (we assume that all services of a component terminate).

Definition 3 (Transient Markov Chain). *A transient Markov chain is a Markov chain of an arbitrary, but finite length [10].*

This requires that all states of the Markov chain are transient and that the state space S is finite. Otherwise, the process might be infinite long, since once a recurrent state is entered, it is visited infinitely often. The finiteness of the process requires a set of states F whose sum of the outgoing transition probabilities is below one, that is $\sum_{k=1}^{n} P(s_f, s_k) < 1$ for all $s_f \in F$. If the process enters a state s_f in F, there is a certain probability (given by $1 - \sum_{k=1}^{n} P(s_f, s_k)$) that it never leaves s_f and, therefore, terminates. The states in F are called *final states*. This definition contradicts the original definition of Markov chains, where the sum of the Markov probabilities of all outgoing transitions of a state must be one.

2.3 Markov Models

Markov models are visualisations of Markov chains which can be seen as FSM whose transitions are annotated with the probability of taking the transition from the current state. Like in Markov chains, the next state depends only on the current state and is independent of the past (Markov property). The fact that Markov Models are an extension of FSMs implies that the state space of the described Markov chains has to be finite. It is also required that the process has a single start state.

Definition 4 (Markov Model). *A Markov Model $M = (E, S, F, s_0, \delta, u)$ consists of an input alphabet E, a finite set of states S, a set of final states F, a start state s_0, a transition function $\delta : S \times E \to S$, and a total function $u : S \times E \to [0; 1]$ mapping a probability value to each transition [14]. It must hold that*

$$\sum_{e \in E} u(s,e) = 1 \qquad \text{for all } s \in S - F \text{ and}$$

$$\sum_{e \in E} u(s,e) \leq 1 \qquad \text{for all } s \in F.$$

The sum of all outgoing transition probabilities must be one for non-final states and below one for final states. Unlike Markov chains, the transitions are annotated with input symbols. Therefore, it is possible that more than one transition exists from one state to another.

Example 1 (Markov Model). Figure 2 shows a Markov model $M = (E, S, F, s_1, \delta, u)$ with the input alphabet $E = \{d1, d2, d3, d4\}$, the state set $S = \{s_1, s_2\}$, the final state set $F = \{s_2\}$, and the start state s_1. The transition function δ and the probability function u are defined as shown in figure 2. The transition probabilities are printed in braces behind the input symbol of a transition. The Markov model represents the service effect specification of the imaginary service $d0$. Starting in the state s_1 the process returns to

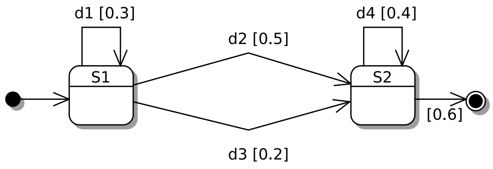


Fig. 2. Service Effect Specification of $d0$ as a Markov Model

this state with a probability of 0.3 or goes to s_2 with a probability 0.7 = 0.2 + 0.5 taking either transition $d2$ or $d3$. The example illustrates the differences between Markov model and Markov chains. Two transitions lead from the start state s_1 to the final state s_2. Both are associated with different input symbols and transition probabilities. This cannot be modelled with Markov chains, since they do not allow more than one transition from one state to another. Furthermore, the transitions of Markov chains are not labelled.

2.4 Expected Number of Visits to a State

We can compute the expected number of visits to each state of a Markov chain using its *potential matrix*. The expected number of visits to a state are used in section 3 to compute the service execution time.

Definition 5 (Potential Matrix). *Consider Markov chain $X = \{X_m, m = 0, 1, \ldots\}$ with the finite state space S. Let the random variable N_j be the total number of visits to state s_j. Then the matrix R whose elements are $R(s_i, s_j) = E[N_j | X_0 = s_i]$, the expected number of visits to state s_j starting in state s_i, is called the* potential matrix *of the Markov chain X [5, p. 123].*

The potential matrix $R(s_i, s_j)$ of a Markov chain contains the expected number of visits to a state s_j starting in state s_i on a path of arbitrary but finite length. The expected number of visits can be calculated with the following theorem [5].

Theorem 1 (Calculation of the Potential Matrix). *Let X be a Markov chain with a transition matrix P. Then the potential matrix R is given by*

$$R = (I - P)^{-1}.$$

Within the scope of this paper, we consider the special case of transient Markov chains with a finite state space. In this case, the inverse matrix $R = (I - P)^{-1}$ of theorem 1 exists, since all state of the Markov chain are transient [10,5,18]. Next, we demonstrate how the potential matrix and its computation presented in theorem 1 can be used to calculate the expected service execution time based on the ideas in [18].

3 Time Consumption of a Service

The proposed prediction model demands that the service effect specifications of a component are given as transient Markov models. The states of the Markov model represent the execution of internal component code. The transitions are associated with external service calls and represent the execution of external code. Both, states and transitions, are associated with random variables representing their execution time. “Random” does not mean that nothing is known about the variables. In fact, a random variable can be specified in different ways, for example, by a probability density function or an expected value.

As discussed in section 2.4, we can compute the expected number of visits for each state in a transient Markov chain. In our case, we need the number of visits to the states as well as to the transitions, since the service effect specifications are given as transient Markov models, whose states and transitions consume time. To compute the service execution time, the Markov model has to be transformed first. This introduces an additional state for each transition so that the number of visits to the transition can be determined. Next, the potential matrix, which contains the expected number of visits to each state, is computed. Combining this with each state’s execution time yields the mean execution time of the service specified by the Markov model.

3.1 Transformation of Markov Models by Transition Substitution

Unlike Markov chains, Markov Models allow multiple transitions from one state to another. To preserve the knowledge about these transitions in a Markov chain and to enable the computation of the number of visits to a transition, we substitute each transition by a calling transition, an intermediate state, and a return transition. So, the transition and its properties can be associated with the intermediate state. In the following, we will call the result of the substitution *transformed* Markov model.

In a transformed service effect specification only states consume time and transitions are timeless. The time consumption of the service call and return might play a significant role for remote procedure calls in distributed systems. For sake of simplicity, their time consumption is included in the time consumption of the states.

Example 2 (Transition Substitution). Figure 3 shows the transformed version of the Markov model in figure 2. A new state is introduced for each transition. For instance, the state s_{d1} is introduced for the transition $d1$. The calling transition of the service $d1$ is labelled with $d1'$. The probability associated with the transition $d1'$ is the probability of the transition $d1$, since the process must visit the intermediate state every time the original transition has been taken. All return transitions in figure 3 are labelled with the symbol ret. Their probability is one, since the process must leave the intermediate state and the probability of the calling and return transition must be the same as the probability of the original transition. In the next step, we demonstrate how the expected number of visits to a state can be used to compute the expected execution time of a service.

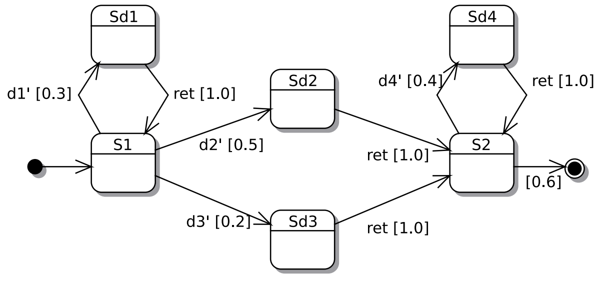


Fig. 3. The Markov Model of Figure 2 with Substituted Transitions

3.2 Computation of the Mean Service Execution Time

We can compute the expected number of visits to each state with the potential matrix of the Markov model. After the transformation of a Markov model, states are the only entities consuming time. Now we can compute the mean service execution time as follows [18].

Let $M_s = (E, S, F, s_0, \delta, u)$ be the transformed Markov model of service d. Given the expected number of visits to each state in M_d and its expected time consumption, we can compute the total expected time consumption of the service. The expected number of visits to a state v_s multiplied by its expected execution time $E[X_s]$ yields the total time spent in s during the execution of the service. The sum of these times over all states in M_d yields the expected execution time of the service d:

$$E[X_d] = \sum_{s \in S} v_s * E[X_s]. \tag{1}$$

Example 3 (Expected Service Execution Time). The transition matrix of the Markov model shown in figure 3 is:

$$P = \left(\begin{array}{c|cccccc} & s_1 & s_{d1} & s_{d2} & s_{d3} & s_2 & s_{d4} \\ s_1 & 0 & 0.3 & 0.5 & 0.2 & 0 & 0 \\ s_{d1} & 0 & 1.0 & 0 & 0 & 0 & 0 \\ s_{d2} & 0 & 0 & 0 & 0 & 1.0 & 0 \\ s_{d3} & 0 & 0 & 0 & 0 & 1.0 & 0 \\ s_2 & 0 & 0 & 0 & 0 & 0 & 0.4 \\ s_{d4} & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right)$$

We added the state names to the matrix to clarify which column and row is associated with a certain state. The rows of the matrix represent the source states and the column the destination state. For example, the probability of going from s_1 to s_{d1} is $P(s_1, s_2) = 0.3$.

The potential matrix (see definition 5) R corresponding to the transition matrix P is:

$$R = (I - P)^{-1} = \left(\begin{array}{c|cccccc} & s_1 & s_{d1} & s_{d2} & s_{d3} & s_2 & s_{d4} \\ s_1 & \mathbf{1.43} & \mathbf{0.43} & \mathbf{0.71} & \mathbf{0.29} & \mathbf{1.67} & \mathbf{0.67} \\ s_{d1} & 1.43 & 1.43 & 0.71 & 0.29 & 1.67 & 0.67 \\ s_{d2} & 0 & 0 & 1 & 0 & 1.67 & 0.67 \\ s_{d3} & 0 & 0 & 0 & 1 & 1.67 & 0.67 \\ s_2 & 0 & 0 & 0 & 0 & 1.67 & 0.67 \\ s_{d4} & 0 & 0 & 0 & 0 & 1.67 & 1.67 \end{array}\right)$$

The matrix R contains the expected number of visits $R(s_i, s_j)$ of each state s_j starting in s_i. The expected number of visits to a state s_j by the whole Markov process is given by the entry $R(s_1, s_j)$. State s_1 is the start state of the process and contains the number of visits of the process to the state. For example, on average s_1 is visited 1.43 times, s_{d1} 0.43 times, and s_2 1.67 times. Note: these values are expectations and therefore, do not need to be an actual possible number of state visits. During the execution of our example Markov model, the process cannot visit s_1 1.43 times. It is either visited once, twice, three times and so on.

Table 1 shows the expected number of visits to each state according to the potential matrix in example 3, the expected time consumption of the state, and their product. The sum of the time consumed by each state is shown at the bottom.

Note that the resulting time consumption is an expected value. The actual service execution time might vary strongly. However, the mean value of a large number of measurements converges to the expected value as the number of measurements grows.

Table 1. Time Consumption of the Service

	State	Number of Visits	Expected Time Consumption	Visits * Time
1	s_1	1.43	$15\mu s$	$21.5\mu s$
2	s_{d1}	0.43	$160\mu s$	$68.8\mu s$
3	s_{d2}	0.71	$2500\mu s$	$1775.0\mu s$
4	s_{d3}	0.29	$120\mu s$	$34.8\mu s$
5	s_2	1.67	$30\mu s$	$50.1\mu s$
6	s_{d4}	0.67	$450\mu s$	$301.5\mu s$
Total				$2251.7\mu s$

4 Assumptions of the Model

The approach described above makes several assumptions regarding the properties and knowledge about the analysed system. The purpose of this section is the explicit listing of the underlying assumptions. In section 5, we evaluate the assumptions with an experiment. We will see that most of the assumptions can be only considered as valid under certain constraints.

The assumptions of the model concern two different areas. On the one hand the usage of Markov models and expected values leads to mathematical assumptions made by these methods. On the other hand, the prediction model assumes that the modeller knows certain details of the analysed component and its environment. Both types of assumptions are discussed in the following.

4.1 Mathematical Model

M1 - Markov Property: The Markov property describes the fact that, for Markov chains, the choice of the next transition only depends on the current state and not on the path to the state. This neglects the fact, that in real world applications the control flow can be influenced by its past.

M2 - Independence of Execution Times: Another assumption results from the addition and multiplication of the expected values in formula 1. By doing so, we assume that the execution times of different states and services are independent. In reality, this assumption might not hold. For example, if due to a high workload of another program the execution of a service takes unexpectedly long, it is very likely that the performance of the next service will be poor as well. This aspect leads to the following assumption.

M3 - No External Influences on the Execution Time: If we consider the execution time of a service, we assume that it is the execution time of the service only. Even if the examined program is the only program running, this is not true in general. On every practically relevant multi-threading operating system several services are running in the background. Furthermore, the execution environment itself influences the time consumption of a service, for example, by a Just-In-Time compiler or a garbage collection.

M4 - Independence of Input Parameters: The execution time of a service is assumed to be independent of its input parameters. This assumption is made no matter if we describe the execution times by probability mass functions or mean values. In the second case, the assumption is stronger, since no variation of the execution time is captured.

It can be easily seen that this assumption is very critical for certain kinds of methods. For example, the lengths of a file strongly influences the execution time of the method `SendContentToClient`, which transfers the file to the client host.
M5 - Appropriateness of Mean Values: Furthermore, we require that the execution times of services and states are given as expected values. This assumption is somewhat related to the assumptions *M3* and *M4* listed above. However, these hold for probability density functions as well. Compared to probability density functions, mean values are easy to specify for the software architect. On the other hand, the expected value neglects important information about the random variable, which it describes. For example, no information about the deviation or the distribution of the random variable is given. This information would allow a more detailed prediction of the service performance [6], but requires a higher effort during its specification. Also, it leads to a computational intensive prediction model.
M6 - Negligible Influences of Concurrency: Limiting ourselves to Markov models, we can only model single-threaded software systems. However, most existing applications are multi-threaded, especially if graphical user interaction is involved or the system is serving several clients simultaneously like the web server in our example. If the concurrency of the system is not to high, one might neglect it, since this results in simpler computational models for performance prediction.

This assumption is closely related to assumption *M3*. However, the difference lies in (a) the source of the influence and (b) the knowledge about the influence. In case of a multi-threaded system, the influence on the execution time is caused by the application itself. So, it is known in advance under which circumstances a new thread is started, which resources it requires, and which job it fulfils. This knowledge can be used to determine the influence of threads on the system's performance. This is not possible for external influences, since, in general, the information about the runtime environment of the system is not sufficient.

4.2 Availability of Data

To predict the execution time of a service, we demand information from the component developer as well as from the component deployer. From the component developer, we require information about the inner structure of the component, which is used to compute the pre- and postconditions of the component in its environment. The component deployer has to specify the deployment context of the component(s). So, we assume that certain data is available for the prediction of the system's performance. In the following, we will list the required data and discuss how it can be obtained.

D1 - Service Effect Specifications: From the component developer, a description of the internal structure of a component is required. This description is given in form of service effect specifications. In general, a service effect specification models external calls executed by a service in form of signature lists or protocols.

There are two ways to determine the service effect specifications of a component depending on the way the system is developed and/or if the component is already implemented or is developed from the scratch. In the first case (bottom up), the source code of the component already exists and can by used (by the component vendor) to generate the service effect specifications automatically. This is a rather difficult task and might

not be applicable in every case. Currently, this approach is investigated in the scope of the development of a round trip engineering tool form component based software systems in our group. In the second case (top down), the service effect specifications must be derived from the component specification. This includes inner component specifications as state charts and intra component specifications such as sequence diagrams. Both can be used to derive service effect specification, since we are only interested in the externally visible states of the components. For our example in section 5, we used state charts to explicitly specify the service effect.
D2 - Transition Probabilities: Despite the specification of the service effect, the transition probabilities of the Markov model are needed. The specification of this probabilities is a hard task and it is not known whether the component developer or component deployer is supposed to do this. On the one hand, the component developer has a better knowledge of the inner component dependency and, therefore, can handle the relationships between input parameters and branching probabilities better. On the other hand, the component deployer knows the components context and has a better idea of the operational profile of the component. The knowledge about both aspects is required to give reasonable good estimation of the transition probabilities. Therefore, we propose the parametrisation of the transition probabilities by the operational profile.
D3 - Execution Times of Internal States: The component developer has to specify the expected execution times of the states of the service effect specifications. These represent the execution of internal component code. Until now, the execution time of a state is specified as a single figure only. It might be useful to define the execution time as a function of the component's deployment context, since external aspects like the underlying hardware and execution environment influence the execution time of the component's code and external services.
D4 - Execution Times of External Service Calls: The component deployer has to specify the execution times of external services, if they cannot be computed by our approach. As for the component code, the execution times of external services depend on the underlying hardware and execution environment. Standard benchmarks can be used, to measure the execution times of the services provided by the environment (e.g., the class libraries of .NET). This allows more accurate predictions for a component in a special environment.

5 Evaluation of the Assumptions

The mathematical assumptions listed in section 4.1 require a detailed evaluation using a real system. We performed two case studies using a prototypical web server developed in our group to evaluate the assumptions. Both case studies give us hints, which assumptions hold and which need to be examined in more detail. Unfortunately, they do not (in)validate any of the assumptions. A more detailed evaluation with different software systems is required to reach that goal. However, the results are good indicators for possible problems and pointers for future research.

For the evaluation of the prediction model introduced above, we compare measured mean execution times of our web server to the computed predictions of the model. The web server is entirely written in C# and is based on the Microsoft .NET 1.1 framework.

It consists of a set of components that communicate over a set of specified interfaces. It is important to mention, that the web server is multi-threaded starting a new thread for each incoming request.

5.1 Evaluation Model

The evaluation does not only require the measurement of the execution time of a service, but also must provide detailed information about the order of service calls to retrace a path taken in the service effect specification to compute the transition probabilities of the Markov model. This information must be stored in a way that it can be easily analysed. We developed a prototype of a profiler, which enables us to log the execution time of each service call and the order in which the services were called. Furthermore, we are able to retrieve the caller of a service. This information is used to enrich the service effect specifications of the web server with transition probabilities. Therefore, the number of visits to each state and transition is counted. The number of visits to a state must be greater than or equal to the number of visits to its outgoing transitions, since the state must be visited before one of these transitions is taken. Furthermore, if the state is a final state, it might happen that the execution terminates in the state. So, the transition probability from a state s_i to a state s_j is:

$$P(s_i, s_j) = T_{i,j}/S_i,$$

where S_i is the number of visits to state s_i and $T_{i,j}$ is the number of visits to the transition from s_i to s_j.

Figure 4 shows the service effect specification of the method `HandleRequest`, which is provided by the `StaticFileProvider` component. The method handles requests to objects stored in the local file system, e.g. pictures and static web pages.

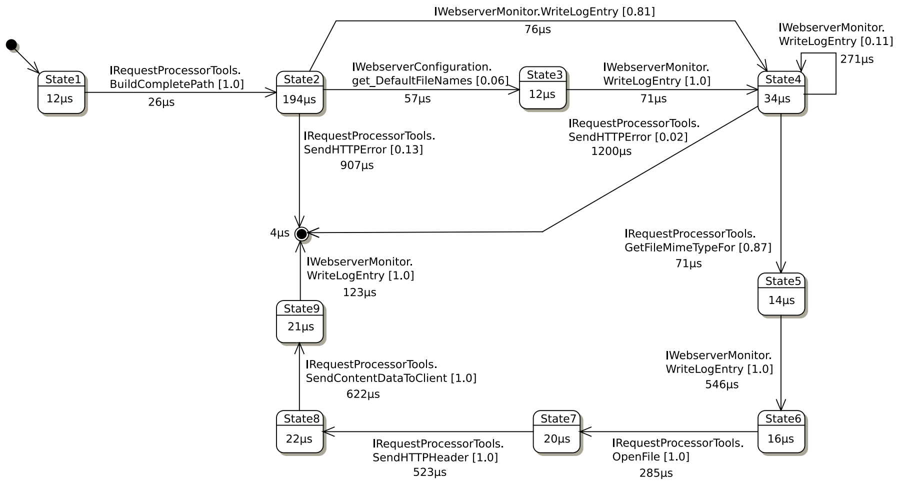


Fig. 4. Markov Model of the Service `HandleRequest` of the `StaticFileProvider` Component

The transition probabilities are printed in braces behind the service name associated to the transitions. For example, the probability of sending a HTTP error to the client from the state `State2` is 0.13, the probability of writing a message to the log file and continue the execution is 0.81, and the probability that the default file name must be retrieved first is 0.06. Moreover, the picture shows the measured execution times of states and transitions in microseconds. We use the service effect specification shown here to evaluate the assumptions.

5.2 Experimental Evaluation

The cases studies performed here show that some of the assumptions made by the proposed models (and many other performance prediction models as well) only hold under certain constraints. To get the big picture, further evaluations, including a more detailed and stricter analysis, are already planned in the context of our project. For the case studies, we used the service effect specification of the method `HandleRequest` shown in figure 4.

Both case studies were conducted as follows. The prototypical web server was running on a machine with a controlled environment, that is, the web server was the only running program. So, interferences of other programs were excluded as good as possible. On another machine a web application stress tool was used to simulate the simultaneous access of ten users on the web server. Both case studies lasted 30 minutes. The example web pages used here contained only links to static HTML and picture files. To make the evaluation more intresting, we included some links to non-existing files in our example pages. For the first case study, all links were clicked equally often. In the second case study, we simply omitted clicking the broken links. This represents only a small change of the operational profile, but already has an effect on the computations and the results. In both case studies, we used the measured data to compute the transition probabilities and average execution times. Figure 4 shows the service effect specification of the service `HandleRequest`. The annotated execution times and transition probabilities are the results of the first case study. The results of the second case study deviated from the first experiment in some points. Most important, the transition probabilities at the states 2 and 4 changed, since the service `SendHTTPError` was not executed in the second case study. This is obviously a consequence of the modified operational profile.

In the first case study, the service `HandleRequest` was called 7617 times in total. Its mean execution time was $2438.15\mu s$. In the second case study, the service was called 6950 times and its average was $3233\mu s$. The probability mass functions of both case studies are shown in figure 5. Both functions show a wide overlap, their differences might be explained be the differences in the operational profile. Considering the probability mass functions, the big difference in the mean values is quite surprising.

A closer analysis of the measured results shows, that about one percent of all measured results deviate strongly from the remaining values. The execution times of these service calls were significantly larger. For example, the maximum time consumption of the service `HandleRequest` was $678395\mu s$, which is 333 times larger than the median. What is the cause of these strong deviations? The logged information shows, that during the execution of the services with extremely long execution times other services

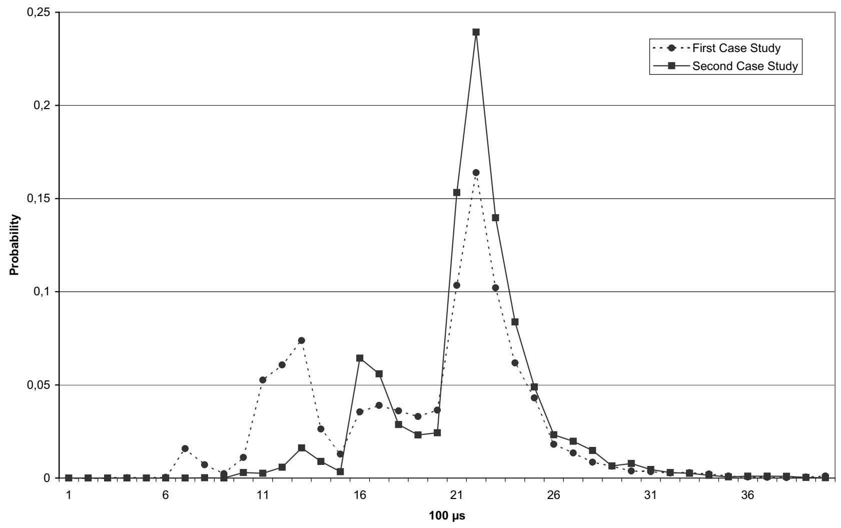


Fig. 5. Results of Both Case Studies as Probability Mass Functions

were started and/or executed. This information is derived from the start and stop time of each service. So, the multi-threading of the web server was one of the causes of the long delay. Therefore, one has to take care when modelling a multi-threaded application, like the web server, only with Markov models. In general, we can state that assumption *M6* does not hold, even for systems with a low level of concurrency. However, we continue the analysis of our results by removing the influence of multi-threading. To do so, we remove the strongly affected measurements. In total, about one percent of the measured results of both case studies is removed for this reason.

Now we have a mean execution time of $1981\mu s$ for the first and $2233\mu s$ for the second case study. The remaining difference in both values (and the probability mass functions in figure 5) can be explained by the differences of the operational profile used in both experiments. Before looking into this, we have to examine the accuracy of the predicted execution times.

5.3 Comparing the Predicted Values with the Computed Results

As first challenge of our prediction model, we chose the computation of the execution time of the service `HandleRequest` based on the exact measured transition probabilities and execution times of states and transitions. This resulted in exact the same execution time of $1981\mu s$ and $2233\mu s$ for both experiments. So, the model is correct if it gets the correct input values. However, in reality it is quite unlikely that a system developer is able to estimate the execution time of a service with that accuracy. For this reason, we the analyse the influence of errors on the predicted values. As we will see, even with inaccurate input values useful predictions can be made. We use overall service execution times instead of estimating the execution time of each transition separately and round the execution times to $100\mu s$ for transitions. In both case studies, the execution times of all states are set to $20\mu s$ except state 2, which is a lot more complex than the other states. It is set to $200\mu s$. The service `SendHTTPError` has an overall measured execution time of $900\mu s$ in the first case study and is not called in the second one. The service `WriteLogEntry` has an overall measured execution time of $100\mu s$ in both case studies. Interestingly, the results for all services (except for the service `SendHTTPError`, which is not called in the second case study) are the same for both case studies. The resulting execution time is $2060\mu s$ for the first and $2216\mu s$ for the second case study, which equals an error of 4% and less than 1% respectively. Next, we use transition probabilities that are rounded to one decimal place instead of two decimal places. This yields an error of 11% for the first and less than 1% for the second case study. If we predict the results of the second case study with the rounded results of the first we get a mean execution time of $2216\mu s$, which corresponds to the result using the rounded values of the second case study, as expected.

The results show, that there are two main sources of errors in our prediction model. The first one concerns the *estimated execution times*. Equation 2 shows that there is a linear relationship between the maximal error of the estimated input times and the computed result.

$$\sum_{s\in S} v_s(aE[X_s]) = a\sum_{s\in S} v_s E[X_s] = aE[X_d] \tag{2}$$

In the equation, a is the deviation of the input time with the greatest error and $E[X_d]$ is the actual expected execution time of service d. The maximal error of the computed result is $e_{\max} = |E[X_d] - aE[X_d]|$. In general, the actual error made during the computation is likely to be significant smaller, since most errors are smaller and different errors equalise each other.

The second source of errors are wrongly *estimated transition probabilities*. The influence of these errors strongly depends on the service effect specification. The probabilities with the strongest effect can be found at loops or cycles and alternatives with strongly varying execution times. The number of visits to a state connected to a cycle depends on the probability associated with a loop. A higher probability yields a higher number of visits which results in a longer execution time. The estimation of transition probabilities is a difficult task, since they heavily depend on the operational profile of the component as we have seen in both case studies. In the service effect specification of figure 4, the probabilities of the outgoing transitions of state `State2` are defined by the incoming requests. For example, the `SendHTTPError` transition is only taken if the incoming request is faulty. This is sometimes the case in our first case study, but not in the second one. So, for the specification of the transition probabilities a certain domain knowledge is required.

5.4 Consequences for the Assumptions

After the presentation of the main results of the case studiess, we have to examine the consequences for the assumptions listed in section 4.1.

Assumption *M1* (Markov Property) seems to be valid only if the operational profile of the system does not change. In other words: The transition probabilities are a function of the operational profile. However, it is very likely that other aspects, like the input data, also influence the transition probabilities.

Assumption *M2* (Independence of Execution Times) seems to be valid if mean values of a single-threaded application in a controlled environment are considered. For multi-threaded systems the case is a little more complex. One percent of the service calls were extremely slow due to this factor. The analysis of these calls shows, that in most cases only one of the methods called by the service consumes the largest amount of the total execution time. The other services are slower as well, but their contribution to the overall execution time is comparatively small.

Unfortunately, we cannot make any statement concerning the assumptions *M3* (No External Influences on the Execution Time) and *M4* (Independence of Input Parameters) from the results of the case studies. So, these require further investigations.

The cases studies show that a mean value (assumption *M5*)is only sufficient to describe a methods execution time, if external and internal influences are removed completely. Especially, the second case study showed that in the case of disturbing influences, the mean value is not a good representation of a random variable, since 95% of all measured values were smaller than the mean. This was caused by only one percent of the measured data. Probability mass functions might be a more appropriate method to specify service execution times [7]. However, they are hard to specify in advance for the system developer.

The blocking of concurrent threads of the web server is one of the causes of the strong deviation of some of the measured values. Therefore, multi-threading cannot be neglected, even if its influence is considered low (assumption *M6*). One might argue that this problem can be solved using probability mass functions instead of mean values. Unfortunately, they can deal only with the influence of constant multi-threaded behaviour. For example, it is not possible to predict the performance of a system with 100 user based on the measurements made with 10 users. Furthermore, it is hard to determine the influence of multi-threading on the execution time of a single service in advance.

6 Related Work

Several performance prediction models have been proposed during the past years. Many of these are based on Markov chains and, therefore, share some of the assumptions of the proposed approach. Bolch *et al.* [4] and Trivedi [18] give a good introduction in the basic concepts of performance and/or reliability prediction models. They introduce Markov chains, queueing networks, and stochastic Petri nets to evaluate performance. Some different performance evaluation methods are evaluated in [2,7]. Gorton *et. al.* evaluate the influence of the middleware on the performance of component-based software architectures [9]. A prediction model that is closely related to ours uses parametric performance contracts[6,13,10] and predicts the execution time of component-based software architectures using probability mass functions. The authors of [8] propose an approach to get the required input data for the prediction of a program's performance and/or reliability. This includes the transition probabilities between components, the execution time, and reliability. Even if their approach is quite similar to ours, concerning the retrieval of data, the selected level of granularity is lower. They consider the consider the interaction between components omitting information about single services.

7 Conclusion and Future Work

In this paper, we developed an approach to calculate the mean execution time of a service using transient Markov models based on the ideas in [18]. The approach considers the influence of the execution times of external services on the performance of the services provided by a component. Therefore, the service effect specification and the mean execution times of external services are required. Usually, both can be obtained with low additional effort. Service effect specifications are closely related to a component's code and can be automatically generated in certain cases. The execution times of external services can either be measured by default benchmarks or be computed using the proposed prediction model (compositionality). Furthermore, we verbalised the mathematical assumptions and the availability of required data. We conducted two case studies to evaluate the predicted values to the measurements and to analyse the validity of the assumptions. For this, a prototypical web server was used. We discussed the mathematical assumptions of the model on the basis of the results. Most of the assumptions hold only under certain constraints, which limit the applicability of the approach. However, the case studies showed that realistic predictions can be made if the discussed constraints hold. As a result of the work presented in this paper, we gained a better understanding of the advantages and problems of Markovian performance prediction models.

Future research has to accomplish two goals, which interact with each other. On the one hand, the validity of the assumptions of this and other approaches need to be evaluated in more detail to determine the exact possibilities and limits of a given approach. On the other hand, methods and techniques have to be developed to overcome the constraints caused by the mathematical assumptions, if and only if this is necessary. The evaluation of the assumptions showed that at least to areas are of great interest: The influence of the operational profile and/or input data and the influence of concurrency.

Acknowledgements. I would like to thank Klaus Krogmann for implementing the web server and members of the Palladio research group and Heinz Schmidt for the fruitful discussions.

References

1. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
2. S. Balsamo, M. Marzolla, A. Di Marco, and P. Inverardi. Experimenting different software architectures performance techniques: a case study. In *Proceedings of the fourth international workshop on Software and performance*, pages 115–119. ACM Press, 2004.

3. S. Becker, V. Firus, S. Giesecke, W. Hasselbring, S. Overhage, and R. Reussner. Towards a Generic Framework for Evaluating Component-Based Software Architectures. In K. Turowski, editor, *Architekturen, Komponenten, Anwendungen - Proceedings zur 1. Verbundtagung Architekturen, Komponenten, Anwendungen (AKA 2004), Universität Augsburg*, volume 57 of *GI-Edition of Lecture Notes in Informatics*, pages 163–180. Bonner Köllen Verlag, Dec. 2004.
4. G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. John Wiley & Sons Inc., 1998.
5. E. Cinlar. *Introduction to Stochastic Processes*. Englewood Cliffs (New Jersey), Prentice-Hall, 1975.
6. 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)*, Electronic Notes in Theoretical Computer Science. ETAPS 2005, 2005.
7. V. Firus, H. Koziolek, S. Becker, R. Reussner, and W. Hasselbring. Empirische Bewertung von Performanz-Vorhersageverfahren für Software-Architekturen. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering 2005 Proceedings - Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 64 of *GI-Edition of Lecture Notes in Informatics*, pages 55–66. Bonner Köllen Verlag, Mar. 2005.
8. S. S. Gokhale, W. E. Wong, J. R. Horgan, and K. S. Trivedi. An analytical approach to architecture-based software performance and reliability prediction. *Perform. Eval.*, 58(4):391–412, 2004.
9. I. Gorton and A. Liu. Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications. *IEEE Internet Computing*, 7(3):18–23, 2003.
10. J. Happe. Reliability Prediction of Component-Based Software Architectures. Master thesis, Department of Computing Science, University of Oldenburg, Dec. 2004.
11. S. A. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau. Packaging predictable assembly. In J. M. Bishop, editor, *Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 2002.
12. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
13. R. H. Reussner, V. Firus, and S. Becker. Parametric Performance Contracts for Software Components and their Compositionality. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)*, June 2004.
14. R. H. Reussner, I. H. Poernomo, and H. W. Schmidt. Reasoning on Software Architectures with Contractually Specified Components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, number 2693 in LNCS, pages 287–325. Springer-Verlag, Berlin, Germany, 2003.
15. S. M. Ross. *Introduction to Probability Models, 4th edition*. Academic Press, 1989.
16. Y. Rozanov. *Probability Theory: A Concise Course*. Dover Publications, 1977.
17. C. U. Smith and L. G. Williams. *Performance Solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.
18. K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall, Englewood Cliffs, NJ, USA, 1982.

An XML-Based Language to Support Performance and Reliability Modeling and Analysis in Software Architectures

Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta

Dipartimento di Informatica, Sistemi e Produzione,
Università "Tor Vergata", Roma (Italy)

Abstract. In recent years, the focus of software development has progressively shifted upward, in the direction of the abstract level of architecture specification. However, while the functional properties of the systems have been extensively dealt with in the literature, relatively less attention has been given until recently to the specification and analysis at the architectural level of quality attributes such as performance and reliability. The contribution of this paper is twofold: first we discuss the type of information that should be provided at the architectural level in order to successfully address the problem of performance and reliability modeling and analysis of software systems; based on this discussion, we define an extension of the xADL architectural language that enables the support for stochastic modeling and analysis of performance and reliability in software architectures.

1 Introduction

A reasonably accurate prediction of the extra-functional Quality Attributes (QA) of the software system being developed is required from the design stage, to support critical design choices. This prediction can be exploited to drive decisions about what components and connectors should be used and how they should be architected so as to meet the extra-functional requirements imposed on the design.

The same kind of information can also be leveraged at run-time to support the system evolution and maintenance (operated by humans) but also to enable the system self-adaptation in response to changes in the environment (such as, for instance, in the underlying platform) or in the user requirements (with respect to performance or reliability constraints, for instance).

The actual realization of QA predictions requires that for all services provided by each architectural element a specification of the QA-related characteristics be explicitly represented and published. In this perspective, it has been argued in the literature that the so called *constructive interface* of an architectural element is not sufficient to convey all the information that is necessary for most analysis tasks. This traditional interface model should be enriched with a proper *analysis-oriented* description, referred to as *analytic interface* [12], that represents a more complete model providing information to support predictions about the QA of an architectural ensemble that element is part of.

Within this framework, we present in this paper a proposal about how to structure an analytic interface for components and connectors of a software architecture, to support the analysis at the architectural level of extra-functional attributes of software applications, focusing on stochastic analysis of performance and reliability. First we give some general arguments about the information that should be included within an analytic interface to support this kind of analysis. Then, based on these arguments, we introduce an extension of xADL, an XML based architectural language [7,8], to give a simple yet sufficiently expressive demonstration of how this information could be actually expressed. We point out that our focus here is on the descriptive issues of analytic interfaces. Another important issue, not covered here, is how to exploit the information expressed by the analytic interfaces of the components and connectors of an architectural ensemble to actually carry out predictive analysis. In view of automating as much as possible this analysis, a key role is played by automatic transformation methodologies from design models to analysis oriented models. In this respect, our extended xADL is an example of machine processable design oriented notation that can be used to define the input for such transformations.

The paper is organized as follows. In section 2 we briefly review related work. In section 3 we present the general guidelines we have followed in the definition of a QA oriented analytic interface. Then, in section 4 we show in detail the implementation of these general guidelines as extensions of some xADL schemas. An example to show the usage of these extensions is presented in section 5. Finally section 6 concludes the paper.

2 Related Work

In recent years, the focus of software development has progressively shifted upward, in the direction of the abstract level of architecture specification. However, while the functional properties of the systems have been extensively dealt with in the literature, relatively less attention has been given until recently to the specification and analysis of extra-functional properties at the architectural level. In the following we shortly review (i) proposed architectural languages that include some aspects of QA specification, and (ii) proposed methodologies to (automatically) generate QA analysis models starting from architectural descriptions.

2.1 Architectural Languages and QA Specification at the Architectural Level

The architectural vision based on the "component-connector" model [2,17] for software applications has been explicitly addressed by several architecture description languages (ADLs) [15]. Most of these languages allow to describe the structural aspects of a software architecture, and some dynamic properties. This information is mainly used to support the analysis of functional properties (like dead-lock freedom) of component assemblies. Few results exists about the use of ADLs to support the analysis of extra-functional properties [17,3].

Another language to be mentioned within this context is UML, because of its *de facto* standard status for software design. UML has recently undergone a thorough revision (that has led to the UML 2.0 version) where in particular some architectural concepts have been more clearly modeled and expressed [22]. Besides its use to model the functional aspects of an application design, UML has been also extended to model performance or reliability characteristics. Regarding the performance characteristics, a standard extension has been defined (UML Profile for Performance, Schedulability and Time, SPT [19]). Regarding the reliability characteristics, some UML extensions have been proposed, for example the Reliability Profile (RE) proposed in [4]. Recently, a general framework for the UML specification of QoS characteristics has been standardized[18].

With respect to these works, in this paper we discuss which kind of information a component should provide at its interface to support compositional analysis of attributes like performance and reliability. Moreover, we also present a simple example of implementation of the general concepts we discuss.

2.2 QA Modeling and Analysis

Regarding QA modeling and analysis, there has been recently a great interest in methodologies for the automatic generation of analysis-oriented models (including performance and reliability models) starting from architectural source models, possibly augmented with suitable QA related annotations.

In particular, several proposals have been presented concerning the generation of performance analysis models. Each of these proposals focuses on a particular type of language for the architectural model (spanning ADLs as Acme [17] or AEmilia [3], UML, Message Sequence Charts, Use Case Maps) and a particular type of target analysis-oriented model (spanning simulation models, Petri nets, queueing networks, layered queueing networks, stochastic process algebras, Markov processes). We refer to [1] for a thorough overview of these proposals.

Some proposals have been also presented for the generation of reliability models. Many of them start from UML models with proper annotations, and generate reliability models such as fault trees [6,14], Markov processes [14] and Bayesian models [5]. A different line of research is presented in [20,21] where the notion of *parametric contracts* is introduced to derive models based on Markov processes.

Finally, the work in [12] presents a general framework to integrate software component technologies with analysis and prediction technologies; the framework is based on an abstract component model, expressed through a purposely defined language.

With respect to these works, we define an input language for methodologies generating performance and reliability models.

3 Augmenting xADL Interfaces to Support Stochastic Analysis

Our goal is to define an analytic interface, that is a representation at a suitable abstraction level of the behavior and requirements of a service offered by a component or connector, to support the application of performance and reliability stochastic analysis methodologies. As a general consideration about the definition of this abstract representation, we observe that the behavior of an architectural element (component or connector) is driven by the values of the parameters that are used in an invocation of the service it offers. Another general consideration is that the QA of the element often depends not only on its intrinsic characteristics, but also on the QA of others elements whose services must be exploited. The former consideration suggests the inclusion into an analytic interface of:

(a) an abstract characterization of the offered service parameters, used to access the service;

while the latter consideration suggests the inclusion of:

(b) an abstract description of the flow of requests that will be possibly addressed to other resources and connectors to carry out that service (abstract usage profile).

With regard to point (a), the goal is to provide a simpler (with respect to their "real" definition) but meaningful characterization of the parameters that can be used for service access, since the service dynamics (and hence its performance and reliability properties) may depend on the value of these parameters. The abstraction we introduce concerns the domains where the service parameters can take value. As a general rule, the abstraction with respect to the real service parameter domains can be achieved by partitioning the real domain into a (possibly finite) set of disjoint sub-domains, and then collapsing all the elements in each sub-domain into a single "representative" element [11].

With regard to point (b), we assume that the abstraction consists of a probabilistic description of the flow of requests addressed to other components and connectors, since we are interested in supporting stochastic analysis. The probabilistic abstraction involves both the flow of control (e.g. by modeling the selection of alternative paths as a probabilistic branching) and the service requests embedded in that flow. For the latter, in particular, we note that in the reality, each request may be characterized by a particular list of actual parameters values. To get an abstract stochastic characterization of these parameters, we argue that they should be represented as a suitable list of random variables, taking values in the abstract domains defined for the parameters of the requested service. Their probability distribution should take into account the likelihood of their possible values[1].

[1] How these random variables are actually specified depends on the required level of analysis detail and complexity. Their description can range from the minimal specification of an average value to the complete characterization of their probability distribution.

Based on the discussion above, in the remainder of this section we make a general survey of how we implemented analytic interfaces for components and connectors defined in the xADL language; details and technicalities are deferred to the next section.

xADL is an XML-based architecture description language defined at UC Irvine and Carnegie Mellon University with the goal of being an open, highly extensible and modular language [7,8]. Around the core modules, that support the fundamental architectural constructs (such as components, connectors and interfaces), other add-on modules have been developed to introduce new features or to redefine and specialize existing ones. Besides the additional modules built by the authors of the language themselves, third-party extensions have been contributed; for example in [16] is presented an extension that supports the modeling of component and connector behavior using state-charts.

The modeling capabilities of xADL span the design-time representation of the basic elements of a software architecture, the definition of their types, and the run-time description of a system composed of such elements. As a design choice, xADL supports only structural constructs, keeping the language simple but open to more specialized extensions and capable of being used in very different contexts. The typical concepts also found in other architectural languages are provided by xADL, thus xADL Components may have Interfaces (i.e. what in many works in the literature are called "ports") which represent the connection points through which they can be attached to each other (through connectors) to communicate and interact [8]. Connectors are specified similarly. xADL also includes many other constructs, but those mentioned here are the most relevant to this work.

Since we are interested in augmenting components and connector interfaces with suitable analytic extensions, we focus on the xADL elements that are related to interface description, both at the instance and type level[2].

The original xADL interfaces are characterized in a minimalistic way as they are described by a textual description, a unique identifier and possibly a *direction*. It is important to remark that Interfaces are *part of* the element (Component or Connector) they belong to, so their description is specific to the context of that element. Interfaces also have a reference to their *InterfaceType*, thus the features shared by many Interfaces can be factored out and collected in just one InterfaceType.

According to the general discussion at the beginning of this section, the extensions we introduce basically consist of:

[2] It should be noted that there is a slight mismatch between the terminology used in this work and the one used in xADL. Here we call *instances* the constructs defined in the context of the *archStructure* element. Their purpose is to define a design-time view, whereas the elements defined in the `instance.xsd` schema are meant to construct a run-time representation of a system [8]. Since this work is mainly focused on devising prediction-enabled *design-time* artifacts, we will not consider run-time models and thus we shall ignore the *instance* schema and use the term "instance" as a shortcut for "design-time instance".

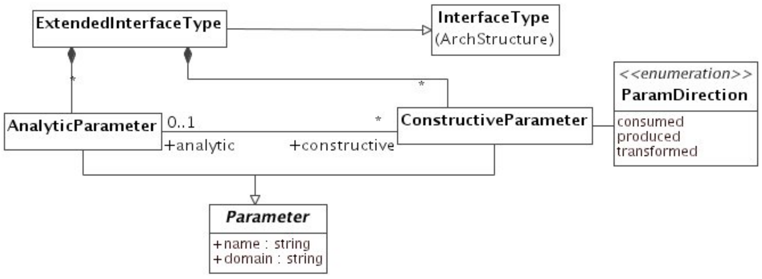


Fig. 1. Metamodel of the QA extension of InterfaceType

- an abstract model of the interface type which augments the basic xADL interface type to explicitly represent the parameters of services offered by the system's constituents;
- an abstract representation of how components and connectors behave when they are asked to carry out one of the services they offer, showing explicitly the flow of requests they address to other components and connectors in order to carry out the service.

The former extension is introduced at the xADL type level, while the latter is introduced at the instance level, as explained in the following.

The conceptual model of the extension we introduced at the type level is depicted in figure 1.

The original *InterfaceType* construct is refined to derive the *ExtendedInterfaceType* element which introduces two new features. The first one is actually a refinement of the constructive characterization of an InterfaceType, giving the possibility of specifying constructive parameters described by a name, a type (set of values) and a direction. The second one concerns the definition of an analytic interface, and consists of analytical parameters whose purpose is to provide a suitable analysis-oriented abstraction of the constructive parameters. This abstraction is intended to be more amenable to the application of a given analysis methodology.

We point out that, for the analysis purposes, only the analytic parameters just described are necessary. We have introduced also the constructive parameters in our ExtendedInterfaceType to support consistency checks between the constructive parameters and the corresponding analytic parameters.

The refinement of the InterfaceType element described so far allows to model analytic properties that are shared by all the instances of the same type and do not depend on the implementation of the single components/connectors.

The second part of our extension deals with modeling the dynamics of interactions between architectural elements. Differently from the modeling of the service parameters, this issue is inherently instance-based, since it is heavily dependent on the logic implemented within each component. Indeed, services characterized by the same interface type could be implemented in different ways by different architectural elements that "host" that interface. This distinction is

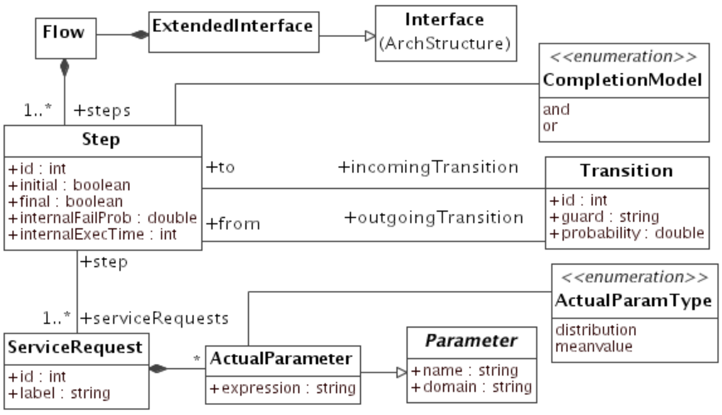


Fig. 2. Metamodel of the QA extension of Interface

consistent with the distinction made in xADL between an InterfaceType and an Interface, where the latter is *part of* the Connector or Component to which it belongs. For this reason we extend xADL at the instance level refining the *Interface* construct as shown in figure 2. The extension consists of the inclusion of a probabilistic representation of the service "behavior". We represent the behavior as a graph whose nodes model the execution of one (or more) service request(s) that are needed for the completion of the task associated with the described Interface. An abstraction of each such request is represented, characterized by suitable performance and/or reliability attributes.

In the next section we present in detail the XML schema implementing these extensions. In section 5 we present an example of how the proposed xADL extension can be used in a practical analysis scenario.

4 QA Schema Definition

ExtendedInterfaceType. The built-in InterfaceType element is extended to contain a list of *ConstructiveParameter*s (lines 5-7) and a list of *AnalyticParameter*s (lines 8-10).

```
 1 <xs:complexType name="ExtendedInterfaceType">
     <xs:complexContent>
 3     <xs:extension base="types:InterfaceType">
         <xs:sequence>
 5         <xs:element name="ConstructiveParameter"
                       type="qa:ConstructiveParameterType"
 7                     minOccurs="0" maxOccurs="unbounded"/>
           <xs:element name="AnalyticParameter"
 9                     type="qa:AnalyticParameterType"
                       minOccurs="0" maxOccurs="unbounded"/>
11       </xs:sequence>
       </xs:extension>
13   </xs:complexContent>
   </xs:complexType>
```

ParamDirection. Parameters are characterized by a direction property, whose value can be one of *produced*, *consumed* or *transformed*.

ConstructiveParameter. Constructive parameters describe the data that are passed to (or produced by) an interface of a given type. For each parameter it can be specified a name, a set of values over which the parameter ranges, and a direction. A set of references to AnalyticParameters can be specified to indicate that the referenced AnalyticParameters are abstractions of the ConstructiveParameter at hand.

```
<xs:complexType name="ConstructiveParameterType">
  <xs:sequence>
    <xs:element name="Analytic" type="inst:XMLLink" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="domain" type="xs:string"/>
  <xs:attribute name="direction" type="qa:ParamDirection"/>
</xs:complexType>
<xs:simpleType name="ParamDirection">
  <xs:restriction base="xs:string">
    <xs:enumeration value="produced"/>
    <xs:enumeration value="consumed"/>
    <xs:enumeration value="transformed"/>
  </xs:restriction>
</xs:simpleType>
```

AnalyticParameter. An analytic parameter is an abstraction of one or more constructive parameters. It is used to highlight a property that is not explicit in the design model but that is relevant to some kind of analysis. The abstraction is realized by mapping the domain(s) of the corresponding constructive parameter(s) to values of the analytic parameter that represents an abstraction of it (them). The actual value for an analytic parameter (in a call to the service that parameters belongs to) is expected to be specified as a random variable whose value ranges over the set specified as the parameter domain. For each AnalyticParameter a reference to the corresponding ConstructiveParameter may be specified.

```
<xs:complexType name="AnalyticParameterType">
  <xs:sequence>
    <xs:element name="Constructive" type="inst:XMLLink"
          minOccurs="0" maxOccurs="1" />
  </xs:sequence>
   <xs:attribute name="name" type="xs:string"/>
   <xs:attribute name="domain" type="xs:string"/>
</xs:complexType>
```

ExtendedInterface. InterfaceTypes, as extended by ExtendedInterfaceType, only deal with structural features of interfaces, an do so at the type level. Thus they can be used just to make statements such as "all the Interfaces of type I will accept a parameter P, represented in some analytical model by the abstract parameter A". Such a specification does not deal with interface dynamics. This is what the ExtendedInterface element does, by extending the existing xADL Interface. This implies that the dynamics of interfaces is modeled on an instance-basis, as it should be. The proposed representation for the description of this abstract behavior is a graph (see *Flow*), that describes the flow of requests that the component, to which the interface instance belongs, addresses to other components when it is asked to carry out one of the services it offers.

```
<xs:complexType name="ExtendedInterface">
  <xs:complexContent>
    <xs:extension base="types:Interface">
      <xs:sequence>
        <xs:element name="flow" type="qa:Flow" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Flow. A flow represents the behavioral model of an interface and is structured as a graph whose nodes are *Step*s and whose edges are *Transition*s.

Step. A step represents the execution of (one or more) service requests (see below *ServiceRequest*) that are necessary for the completion of the task associated with an interface. A step can be characterized by its internal execution time and by its internal failure probability, where the former is the execution time that depends only on the internal service operations, while the latter is the probability that the service fails when none of the required services has failed. We point out that both these attributes do not give a complete picture of the service performance or reliability, which may depend, in general, also on the performance or reliability of other services. If a single step includes the specification of a set of service requests, a completion model attribute, whose value can be one of {"AND", "OR"}, can also be specified. An "AND" model means that all the service requests must be successful before the next step can be run, whereas in an "OR" model it is sufficient to have at least one successfully completed service request.

```
<xs:complexType name="Flow">
  <xs:sequence>
   <xs:element name="step" type="qa:Step" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="transition" type="qa:Transition"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="Step">
  <xs:sequence>
    <xs:element name="serviceRequest" type="qa:ServiceRequest" minOccurs="0"/>
      </xs:sequence>
        <xs:attribute name="id" type="inst:Identifier"/>
  <xs:attribute name="label" type="xs:string"/>
  <xs:attribute name="init" type="xs:boolean"/>
  <xs:attribute name="final" type="xs:boolean"/>
</xs:complexType>
```

ServiceRequest. Service requests specify the services that are required to complete a step. A service request may be characterized by an ordered list of actual parameters whose values are expressed as random variables. As already discussed in section 3, these random variables are intended to be an abstract representation of the "real" actual parameters used in service invocations. Their co-domain must match the domain specified for the corresponding analytic parameter.

```
<xs:complexType name="ServiceRequest">
  <xs:sequence>
    <xs:element name="serviceCall" type="xs:string"/>
    <xs:element name="actualParam"
                type="qa:ActualParameterType" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="inst:Identifier"/>
</xs:complexType>


<xs:complexType name="ActualParameterType">
  <xs:sequence><xs:element name="expression" type="xs:string"/></xs:sequence>
  <xs:attribute name="id" type="inst:Identifier"/>
  <xs:attribute name="name" type="xs:string"/>
  <xs:attribute name="domain" type="xs:string"/>
</xs:complexType>
```

Transition. Transitions connect steps with their successors and predecessors. The firing of a transition is determined by the value of a guard condition, that is an expression, stated in a suitable language, that can be evaluated as true or false based on some random variable. In many cases modelers only need to specify a probability for a transition and they should not be forced to define a random variable to do so; thus, as an alternative to the guard attribute, we provide the probability attribute, that is expected to be a constant real number in the interval [0-1].

```
<xs:complexType name="Transition">
  <xs:sequence>
    <xs:element name="from" type="inst:XMLLink" minOccurs="0"/>
    <xs:element name="to" type="inst:XMLLink" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="inst:Identifier"/>
  <xs:attribute name="guard" type="xs:string"/>
  <xs:attribute name="probability" type="xs:string"/>
</xs:complexType>
```

5 An Example

In this section we introduce a simple example to show how the language can be used in practice.

Let us consider the architecture of a system that is capable to search items in a list. The system is made up of two components: one offers to the outside the search service, the other one is used by the former to sort the list before performing the search, if it is not already sorted. This is represented in figure 3 using a UML component diagram[3].

When an actor (a human user or another component or subsystem) requests the SearchComponent to perform a search, another request could be made in turn by the SearchComponent to the SortComponent. This means that, even though the latter is hidden from the actor who initiated the computation, it may influence the quality perceived by the user with respect to the computation carried out by the system as a whole. For instance the SortComponent could be too slow, so the whole SearchAndSort system becomes slow, or it may fail with a much higher frequency than the SearchComponent, degrading the overall reliability of the system.

[3] Comparing the UML notation with the conceptual model on which xADL is based it should be noted that the mapping of UML to xADL elements (and vice versa) can lead to some confusion since UML ports seem to correspond to xADL Interfaces and UML interfaces correspond to xADL InterfaceTypes.

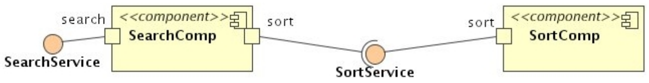


Fig. 3. Architecture of the example

This kind of reasoning, based here on rough, qualitative considerations, can be made more formal and quantitative if the characteristics of the services involved in the system's operation are suitably specified.

In this respect, we point out that a description of the deployment of each architectural element on some supporting platform, and a model of the platform itself are required to reason about QA on a quantitative basis. Moreover, even from a merely structural perspective, descriptions like the one in figure 3 offer an overly simplified view of the system and are not sufficiently detailed for QA analysis. In fact, the components in the figure are shown assuming that interactions happen somehow through an implicit communication infrastructure. Obviously the characteristics of such an infrastructure have a strong influence on most extra-functional properties, such as performance and reliability, not to mention security. This is one of the reasons why we believe that the analysis and prediction of extra-functional properties of complex systems can benefit from an architecture-based methodology: exploiting the notion of connector seems to be the most natural way to encapsulate and represent in a coherent manner all the attributes that pertain to communication and interaction and are relevant to QA analysis.

The listing below shows the type specification for the search component[4] (lines 1-11). The sort component is described similarly (omitted for space reasons). Signatures are used to mean that instances of these types should contain an instance of the interface type specified by the signature. We have also included signatures to be used as attachment point for the processing service offered by some computing node of the underlying platform. This service will be used to carry out internal operations of each component (e.g. the sort algorithm implemented by the sort component). In this way we can specify the deployment of each component to specific platform nodes and the demand addressed to those nodes, which is a relevant information for QA analysis.

In the connector type (a Remote Procedure Call connector in this example) signatures are used to specify the roles that will participate in the interaction mediated by the connector. In addition to a client and a server role, we have also included (analogously to components) signatures to be used as attachment point for the processing and communication services exploited by the connector to carry out its own "interaction" task. We have included two processing service signatures to take into account the processing demand addressed by the connector at the two interaction endpoints (e.g. to carry out parameter marshaling and unmarshaling).

[4] A few details, not relevant to this discussion, have been omitted from some of the listings in this section, so the resulting code could not be strictly valid with respect to the qa.xsd schema or some other xADL schema.

```
<types:componentType types:id="SearchComponentType">
  <types:signature types:id="SearchPort">
    <types:direction>in</types:direction>
    <types:type xlink:href="#SearchInterfaceType"/></types:signature>
  <types:signature types:id="SortPort">
    <types:direction>out</types:direction>
    <types:type xlink:href="#SortInterfaceType"/></types:signature>
  <types:signature types:id="ProcessorPort">
    <types:direction>out</types:direction>
    <types:type xlink:href="#ProcessingInterfaceType"/></types:signature>
</types:componentType>

<types:connectorType types:id="RPCConnectorType">
  <types:signature types:id="ClientRole"><types:direction>in</types:direction>
    <types:type xlink:href="#ClientInterfaceType"/></types:signature>
  <types:signature types:id="ServerRole"><types:direction>out</types:direction>
    <types:type xlink:href="#ServerInterfaceType"/></types:signature>
  <types:signature types:id="Processor1Role">
    <types:direction>out</types:direction>
    <types:type xlink:href="#ProcessorInterfaceType"/></types:signature>
  <types:signature types:id="Processor2Role">
    <types:direction>out</types:direction>
    <types:type xlink:href="#ProcessorInterfaceType"/></types:signature>
  <types:signature types:id="NetworkRole">
    <types:direction>out</types:direction>
    <types:type xlink:href="#NetworkInterfaceType"/></types:signature>
</types:componentType>
```

The specification of the `SearchInterfaceType` as an *ExtendedInterfaceType* is given below. Three constructive parameters are specified, namely the list to be searched, the item to be searched and the output parameter that is used to retrieve the search result (found/not found). These three parameters are abstracted as a single analytic parameter *list_size* which is used to represent the size of the list to be searched (line 9 below), based on the assumption that this is the only relevant information for performance and reliability analysis. With respect to the general discussion made in section 3, note that this abstraction is obtained by collapsing all the lists with same length in the original list domain into a single element of the domain of integer numbers.

```
1 <types:interfaceType xsi:type="qa:ExtendedInterfaceType"
                      types:id="SearchInterfaceType">
3  <qa:ConstructiveParameter qa:name="list" qa:domain="list of items"
                              qa:direction="consumed"/>
5  <qa:ConstructiveParameter qa:name="item" qa:domain="items"
                              qa:direction="consumed"/>
7  <qa:ConstructiveParameter qa:name="result" qa:domain="boolean"
                              qa:direction="produced"/>
9  <qa:AnalyticParameter     qa:name="list_size" qa:domain="integer" />
  </types:interfaceType>
```

The `SortInterfaceType` is defined similarly: it has one constructive parameter called *list* whose type is a list of items and which is "abstracted" into an analytic parameter called *list_size* (line 5).

```
  <types:interfaceType xsi:type="qa:ExtendedInterfaceType"
2                     types:id="SortInterfaceType">
   <qa:ConstructiveParameter qa:name="list" qa:domain="list of items"
4                              qa:direction="transformed"/>
   <qa:AnalyticParameter qa:name="list_size" qa:domain="integer"/>
6 </types:interfaceType>
```

The specifications presented so far cover the type part of the model. We give now the instance specifications, which include dynamic models represented as flows of requests.

The following code is the description of Component-A, an instance of the search component type. The component has one offered interface named "Component-A.searchService" (line 2) whose dynamics is specified as a flow (lines 4-32). In the first part of the flow specification, three steps are defined. The first one (line 5) is just a dummy initial step. The second (6-12) contains a call to the sort service offered by a sort component by means of an interposed connector. When the call is issued, an actual parameter (8-10) is passed to the called (analytic) interface. The third step is a call to the processing service offered by a processor that is required to search an item in the sorted list. Note that in both these steps the actual parameter of the service call is specified as the mean value of some random variable. This mean value is equal to the list size for the call to the sort service (meaning that we are passing the list to that service), while it is equal to the logarithm of the list size for the call to the processing service (assuming that the search component implements a binary search algorithm). The second part of the flow specification (lines 20-31) contains the transitions which specify the execution order of these steps. The first two transitions (lines 20-27) represent the probabilistic choice of whether the sort service is to be executed or not. The specified probability value for the transitions is constant, but in general it can be expressed as a function that is dependent on the interface parameters.

```
   <types:component types:id="Component-A">
 2   <types:interface types:id="Component-A.searchService"
                       xsi:type="qa:ExtendedInterface">
 4     <qa:flow>
         <qa:step qa:id="Component-A.searchService.step-1" qa:initial="true"/>
 6       <qa:step qa:id="Component-A.searchService.step-2">
           <qa:serviceRequest qa:serviceCall="SortPort.sortService">
 8           <qa:actualParam qa:type="meanvalue">
               <qa:expression>$list_size</qa:expression>
10           </qa:actualParam>
           </qa:serviceRequest>
12       </qa:step>
         <qa:step qa:id="Component-A.searchService.step-3" qa:final="true">
14         <qa:serviceRequest qa:serviceCall="ProcessorPort.processingService">
             <qa:actualParam qa:type="meanvalue">
16             <qa:expression>log($list_size)</qa:expression>
             </qa:actualParam>
18         </qa:serviceRequest>
         </qa:step>
20       <qa:transition qa:probability="0.1">
           <qa:from xlink:href="#Component-A.searchService.step-1"/>
22         <qa:to xlink:href="#Component-A.searchService.step-2"/>
         </qa:transition>
24       <qa:transition qa:probability="0.9">
           <qa:from xlink:href="#Component-A.searchService.step-1"/>
26         <qa:to xlink:href="#Component-A.searchService.step-3"/>
         </qa:transition>
28       <qa:transition>
           <qa:from xlink:href="#Component-A.searchService.step-2"/>
30         <qa:to xlink:href="#Component-A.searchService.step-3"/>
         </qa:transition>
32     </qa:flow>
     </types:interface>
```

```
  <types:type xlink:href="#SearchComponentType"/>
</types:component>
```

The dynamics of the connector instance is described similarly as the components. The ClientInterfaceType offered by the connector is defined as follows.

```
<types:interfaceType types:id="ClientInterfaceType"
                     xsi:type="qa:ExtendedInterfaceType">
  <qa:ConstructiveParameter qa:name="list_of_params" qa:domain="array"
                            qa:direction="consumed"/>
  <qa:AnalyticParameter qa:name="size_of_params" qa:domain="integer"/>
</types:interfaceType>
```

The ClientInterfaceType is referenced in the specification of the connector instance given below. It should be noted that the purpose of this model is to represent a generic client-server interaction based on a RPC protocol and it is not specific to the particular interaction between a search component and a sort component. Consequently the service requests are addressed using the names of the connector signatures (used as placeholders for "roles") rather than referring to actual interfaces. It is also important to remark that since the connector is generic, the actual name of the service called at line 21 (where we used "do" as a sort of parameter) is not known before actual component instances are attached to the connector roles. In the specification of the connector dynamics, the parameters of the service requests addressed to processing and communication services are specified as random variables whose mean value is proportional to the size of the parameters "transported" by the connector.

```
   <types:connector types:id="Connector-1">
 2  <types:description/>
    <types:interface types:id="Connector-1.ClientInterface"
 4                   xsi:type="qa:ExtendedInterface">
      <types:description/>
 6    <qa:flow>
        <qa:step>
 8        <qa:serviceRequest qa:serviceCall="Processor1Role.processingService">
            <qa:actualParam qa:type="meanvalue">
10            <qa:expression>k1 * $param_size</qa:expression>
            </qa:actualParam>
12        </qa:serviceRequest>
          <qa:serviceRequest qa:serviceCall="Network1Role.transportService">
14          <qa:actualParam qa:type="meanvalue">
              <qa:expression>k2 * $param_size</qa:expression>
16          </qa:actualParam>
          </qa:serviceRequest>
18        <!-- the 'do' service is a placeholder for the actual name of the
          service required by the component that is attached to the client role
20        of the connector -->
          <qa:serviceRequest qa:serviceCall="ServerRole.do">
22          <qa:actualParam qa:type="meanvalue">
              <qa:expression>k3 * $param_size</qa:expression>
24          </qa:actualParam>
          <qa:serviceRequest/>
26        <qa:serviceRequest qa:serviceCall="Processor2Role.processingService">
            <qa:actualParam qa:type="meanvalue">
28            <qa:expression>k4 * $param_size</qa:expression>
            </qa:actualParam>
30         </qa:serviceRequest>
        </qa:step>
32    </qa:flow>
    </types:interface>
34  <types:type xlink:href="#RPCConnectorType"/>
   </types:connector>
```

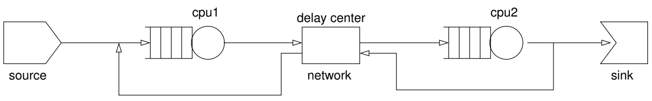


Fig. 4. Queueing network model for the platform

The processors and the network components that model the underlying platform are defined analogously as the components described above, except for the fact that they do not require any service to other components, thus their specification does not include a flow. These components are characterized by attributes such as a processing speed, bandwidth and scheduling discipline, to support the construction of suitable QA analysis models. We report below the listing for a processor component, while the listing of the network component has been omitted for the sake of brevity.

```
<types:component types:id="Processor-1">
  <types:interface types:id="Processor-1.processingService"
                           xsi:type="qa:ExtendedInterface"/>
  <types:type xlink:href="#ProcessorComponentType"/>
</types:component>
```

Using the information embedded into all the analytic interfaces of the architectural elements of this example, we could build performance analysis models such as the queueing network (QN) depicted in figure 4.

In this model, the QN service centers model the processing and communication components of the underlying execution platform, while the circulating jobs model the demand addressed to these (physical) components by the search and sort components connected by an RPC connector. We could also add to the QN model jobs modeling other interfering software load. By solving the QN model, we would get information about, for example, the search service response time for various system loads and characteristics of the underlying platform [13]. We refer to [10] as an example of how the type of information expressed in the analytic interface can be used to build a QN model (even if in [10] it is assumed to start from a UML model). Similarly, we refer to [9] to see an example of how the type of information expressed in the analytic interface can be used to build a reliability model.

6 Conclusion

In this paper we have outlined the requirements that an ADL should satisfy in order to be used for constructing QA prediction-enabled models of complex software systems. We have described a concrete implementation that complies with these requirements, presenting a language that can be smoothly plugged in an existing XML-based ADL (xADL). Our extension of xADL enables a better support for constructive interface descriptions and adds to the language the capability of describing the abstract behavior of component and connector instances in a precise, human- and machine-readable form. The discussion is completed with a basic example that shows the concrete use of the proposed extension in a simple case study. We are currently working toward the implementation of a tool to extract QA related information from an architectural model expressed using the xADL language extension presented in this work, and automatically build a performance or reliability analysis model.

References

1. S. Balsamo, A. di Marco, P. Inverardi, M. Simeoni "Model-based performance prediction in software development: a survey" IEEE Trans. on Software Engineering, Vol. 30/5, May 2004, pp. 295-310.
2. L.Bass, P.Clements, R.Kazman, Software Architectures in Practice, Addison-Wesley, New York, NY, 1998
3. M.Bernardo, P.Ciancarini, L.Donatiello "AEMPA: a process algebraic description language for the performance analysis of software architectures" in Proc. of 2nd Int. Workshop on Software and Performance (WOSP 2000), Ottawa, Canada, 2000
4. V.Cortellessa, A.Pompei "Towards a UML profile for QoS: a contribution in the reliability domain" in Proc. 4th Int. Workshop on Software and Performance (WOSP'04), Jan. 2004, pp. 197-206
5. V. Cortellessa, H. Singh, B. Cukic, E. Gunel, V. Bharadwaj "Early reliability assessment of UML based software models" in Proc. 3rd Int. Workshop on Software and Performance (WOSP'02), July 24-26, 2002, Rome (Italy).
6. D'Ambrogio A. , G. Iazeolla, R. Mirandola "A XML-based method for the prediction of software reliability" in Proc. the 6th IASTED International Conference Software Engineering and Applications, Nov. 4-6, 2002, Cambridge, USA, pp. 234-240.
7. E.M.Dashofy, A.v.d.Hoek, R.N.Taylor. A Highly Extensible XML-Based Architecture Description Language. in *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, 2001, Amsterdam, The Netherlands.
8. E.M.Dashofy. xADL 2.0 Distilled, A Guide for Users of the xADL 2.0 Language. `http://www.isr.uci.edu/projects/xarchuci/guide.html`
9. V.Grassi. "Architecture-based Reliability Prediction for Service-oriented Computing". *WADS 2004*, Workshop on Architecting Dependable Systems, Firenze, Italy, June 2004.
10. V. Cortellessa, R. Mirandola, "PRIMA-UML: a Performance Validation Incremental Methodology on Early UML Diagrams", Science of Computer Programming, Elsevier Science Ed., Vol. 44, p. 101-129, July 2002.
11. D. Hamlet, D. Mason, D. Woit "Properties of Software Systems Synthesized from Components", June 2003, on line at: http://www.cs.pdx.edu/ hamlet/lau.pdf (to appear as a book chapter).
12. S.A.Hissam, G.Moreno, J.Stafford, K.Wallnau. Enabling Predictable Assembly. *Journal of Systems and Software*, vol.65, 2003, pp. 185-198.
13. Lazowska E.D. et al. "Quantitative System Performance: Computer System Analysis using Queueing Network Models" on line at: http://www.cs.washington.edu/homes/lazowska/qsp/
14. C. Leangsuksun, H. Song, L. Shen "Reliability Modeling Using UML" Proceeding of 2003 Int. Conf. on Software Engineering Research and Practice, June 23-26, 2003, Las Vegas, Nevada, USA.

15. N.Medvidovic, R.N.Taylor "A classification and comparison framework for software architecture description languages" IEEE Trans. on Software Engineering, vol. 26, no. 1, Jan. 2000, pp. 70-93.
16. L.Naslavsky, M.Dias, H.Ziv, D.Richardson. Extending xADL with Statechart Behavioral Specification. *WADS 2004*, Workshop on Architecting Dependable Systems, Edimburgh, Scotland, UK, June 2004.
17. B.Spitznagel, D.Garlan "Architecture-based performance analysis" in Proc. 1998 Conf. on Software Engineering and Knowledge Engineering, June 1998.
18. "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms", OMG Adopted Specification ptc/04-09-012, www.omg.org/docs/ptc/04-09-01.pdf.
19. UML Profile for Schedulability, Performance and Time. `http://cgi.omg.org/docs/ptc/02-03-02.pdf`
20. R.H. Reussner, H.W. Schmidt, I. Poernomo "Reliability Prediction for Component-Based Software Architectures" in *Journal of Systems and Software*, pages 241-252, vol. 66, No. 3, 2003
21. R.H. Reussner, V. Firus, S. Becker "Parametric Performance Contracts for Software Components and their Compositionality" in *Proceedings of the 9. International Workshop on Component-Oriented Programming (WCOP 04)*, 2004
22. "UML 2.0 Superstructure Specification" OMG Adopted Specification ptc/03-08-02, www.omg.org/docs/ptc/03-08-02.pdf.

Formal Definition of Metrics Upon the CORBA Component Model

Miguel Goulão and Fernando Brito e Abreu

QUASAR Research Group, Departamento de Informática, FCT/UNL, Portugal
`{miguel.goulao, fba}@di.fct.unl.pt`
`http://ctp.di.fct.unl.pt/QUASAR/`

Abstract. *Objective:* In this paper, we present a formalization of the definition of metrics to assess quality attributes of CORBA components and assemblies. The focus is on the formalization technique, rather than on the evaluation of the metrics themselves. *Method:* We represent a component assembly as an instantiation of the CORBA Component Model metamodel. The resulting meta-object diagram can then be traversed using Object Constraint Language clauses. With these clauses, we construct a formal and executable definition of the metrics. *Results:* We demonstrate the expressiveness of our technique by formally defining metrics proposed informally by several authors on different aspects of components and assemblies' quality attributes. *Conclusion:* Providing a formal and executable definition of metrics for CORBA components and assemblies is an enabling precondition to allow for independent scrutiny of such metrics, which is, in turn, essential to increase practitioners' confidence on predictable quality attributes.

1 Introduction

1.1 Aspects of Component Modeling

Current component models support the functional aspects of component wiring, at least to a certain extent. In simple component models, such as JavaBeans [1], components are accessed through their provided interfaces. In more sophisticated component models, such as the CORBA Component Model (CCM) [2], components may have multiple provided and required interfaces, synchronous and asynchronous operations, as well as the ability to publish and subscribe events.

There is usually less support for the specification of non-functional properties of components than for functional ones. For instance, the new UML 2.0 standard [3, 4] includes constructs for representing several of the above-mentioned component wiring mechanisms, but none for the representation of non-functional properties. The latter are defined as a UML profile [5], rather than as a part of the core language. This may be seen as a limitation in practice, since modeling tools are less likely to support UML extensions on their standard distribution.

1.2 Specificity of CBSE Metrics

One of the goals of Component-Based Software Engineering (CBSE) is achieving predictability of system quality based on the quality attributes of the constituent components [6]. Currently, developers are unable to make such predictions in an automated fashion. Difficulties hampering this task include determining which properties would be useful to component developers and users, how to combine the properties of individual components to predict the properties of assemblies, how to measure such properties, and how to present this information to component users. Current component models used in industry are not *prediction-enabled,* although this is an active topic of research [7, 8]. The term *"prediction-enabled"* is mostly associated with runtime quality attributes, with a focus on those that have objective definitions (e.g. latency) rather than subjective ones (e.g. usability and maintainability). The latter should also be assessed.

Quality attributes can be classified within a quality model. As noted by some authors [9, 10], specific quality models must be developed for CBSE, to address the focus shift from structured, or object-oriented development to Component-Based Development (CBD). For instance, a component user is more concerned with the complexity involved in selecting and composing components than with the implementation details of each component.

From his perspective, components are black-boxes whose evolution he does not control. Existing metrics for structured and object oriented development are not well suited for CBSE, since they are mainly concerned with internal complexity of the components. Several traditional complexity metrics such as the McCabe's [11] and Halstead's [12] ones are useless to a component user, since their computation requires the access to the implementation details, which are not available to component users, due to the black-box nature of components.

1.3 Open Problems in Metrics for CBSE

In a recent survey, we identified several proposals for the quantitative assessment of components and assemblies, based upon their functional properties [13]. We observed several recurrent problems in those proposals, which are also common in metrics proposals for other purposes, such as OO design evaluation:

(1) lack of a quality framework – occurs when a metrics definition is not framed by a particular quality model;

(2) lack of an ontology – occurs when the architectural concepts to be quantified, either of functional or non-functional nature, are not clearly defined, namely in their interrelationships;

(3) lack of an adequate formalism – sometimes, metrics are defined with a formalism that requires a strong mathematical background, which is often not held by practitioners. This limitation hampers metrics usability. On the other hand, it is also common to find metrics defined using natural language in the literature. These definitions tend to be subjective to some extent, thus jeopardizing the correctness and comparability of metrics collection;

(4) lack of computational support – occurs when metrics proponents do not produce tools for metrics collection, or when they do not make them available to other researchers and practitioners;

(5) lack of flexibility – occurs when metrics collection tools are available, but they are either proprietary, or, if they are open source, the metrics definitions are somehow obscurely tangled in the code. The latter hampers the assessment of the correction of their algorithm, as well as the ability to modify them (a recurrent need in experimental software engineering projects);

(6) lack of validation – occurs when independent cross validation is not performed, mainly due to difficulties in experiment replication. Such validation is required before widespread acceptance is sought.

One possible solution to address (1) is using the Goal-Question-Metric (GQM) approach [14]. This approach starts with the definition of goals, which in this case should be aligned with the needs of the CBD actors. These goals should also be defined within the scope of a suitable quality model. Questions are formulated in order to assess the level of achievement of the defined goals. Finally, metrics are defined to provide the information required for answering the formulated questions. In other words, metrics definitions should always be preceded by clear objectives.

In this paper, we present an approach to mitigate the remaining problems (2 to 6). We use the CCM as a representation for components and component assemblies, due to its wide coverage of features provided by current component models used in industry, such as Enterprise JavaBeans, COM or .Net. The CCM has a standard metamodel where the architectural concepts and their interrelationships become more intelligible. We define Object Constraint Language (OCL) expressions upon the metamodel, to specify and collect the metrics, thus solving problem (2). OCL combines formality with a syntax easily understood by practitioners familiar with OO development and is therefore an adequate formalism to help solving problem (3). We circumvent problem (4) by using executable metrics definitions. Indeed, OCL expressions can be evaluated with several existing UML tools, upon a given metamodel. Some of those tools only allow OCL syntax and model conformance checking. Others, such as the USE tool [15], allow loading metadata (objects and links representing model instances and their interrelationships) and evaluate the result of OCL expressions upon that workload. Using OCL expressions makes metrics definitions open and clearly separated from other code, thus solving problem (5). The combination of formality, with understandability and replicability is a facilitator to the independent scrutiny of metrics-based approaches to CBSE, therefore creating conditions to mitigate problem (6).

1.4 Paper Organization

This paper is organized as follows: In section 2 we discuss some related work. In section 3, we briefly present the CCM, as well as its underlying metamodel. In section 4 we formalize metrics for CBSE upon the CCM metamodel. In section 5, we present a component assembly example and the metrics collected upon it. In section 6 we discuss our formalization technique within the framework of the problems identified in the introduction, to stress how it helps mitigating those problems. Conclusions are presented in section 7.

2 Related Work

Recently, researchers have tried to establish requisites and guidelines for CBD metrics, both concerning individual components [16] and component assemblies [17].

Although these proposals do not contribute with concrete metrics, they provide useful insight on the specificities to consider when developing metrics for CBD, mainly in what concerns the focus of such metrics.

Other proposals have contributed to the evaluation of component interfaces and dependencies [18-20]. They focus on different aspects of the interfaces and dependencies of components and are mostly concerned with the complexity involved in understanding those interfaces, and reusing the components. Narasimhan and Hendradjaya proposed metrics to assess component integration density (a measure of its internal complexity) and interaction density (a measure of the complexity of relationships with other components) [21]. Hoek *et al.* proposed metrics to assess service utilization in component assemblies [22].

The previous proposals include, to some extent, informal specifications. Frequently, these contain ambiguous information, particularly in what concerns the basic counting elements used in the metrics formulae. Different interpretations of such elements may lead to different results, when computing the metrics' values. This is a major drawback, when comparing the results from independent metrics collection experiments.

In this paper we present a collection of metrics taken from some of these proposals, formalized with OCL upon the CCM metamodel. Such a formalization implies an explicit interpretation of the original (informal) metrics definition and provides an executable specification for them. We build up on some previous contributions of our team. In [23] we formalized a metrics set for component reusability assessment [20]. We used that formalization to conduct an independent validation experiment on the same metrics set in [24], using the UML 2.0 metamodel. Here, we will use the CCM metamodel, because the latter has more expressive power than UML 2.0 for representing components. Our formalization technique, originally proposed in [25], has also been used in other contexts, therefore with different metamodels. In [26], we formalized well-known OO design metrics upon a OCL functions library named FLAME, aimed at helping metrics extraction in UML 1.x models [27]. More recently, we have successfully applied the same technique with object-relational database schema metrics [28].

3 CORBA Components

The CCM [2, 29, 30] is the Object Management Group (OMG) standard for the specification of software components. As such, it is independent from a specific vendor, both in what concerns the component's programming languages and platforms. CORBA components are created and managed by homes (a home is a meta-type which offers standard factory and finder operations and is used to manage a component instance), run in containers that handle system services transparently and are hosted by generic application component servers. Each component may have several provided and required interfaces (also known as facets and receptacles) as well as the ability to publish and subscribe events (by means of event sources and sinks). Components also offer navigation and introspection capabilities. The CCM also has support for distribution and Quality of Service (QoS) properties.

The CCM specification includes a Meta Object Facility–compliant metamodel [31], where the CCM modeling elements are defined precisely. The metamodel includes three packages (see Fig. 1). The `BaseIDL` package contains the modeling elements concerning the CORBA Interface Description Language (IDL). `BaseIDL` is extended by `ComponentIDL`, to add the component specific constructs. Finally, `ComponentIDL` is extended by the `CIF` package, with the model elements of the Component Implementation Framework, which include the definitions relating to the component life cycle.

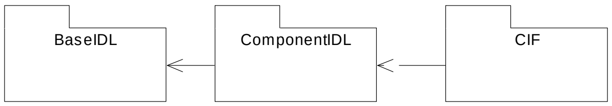


Fig. 1. CCM metamodel packages

The metrics formalizations presented throughout this paper use only metamodel abstractions defined in the `BaseIDL` and `ComponentIDL` packages, as we were able to find all the required abstractions in them. Nevertheless, the `CIF` package abstractions could also be used, with an extended metrics set.

4 Metrics Formalization with OCL

4.1 Formalization Technique

A CCM assembly can be represented as an instance of the CCM metamodel. This instance can be seen as a directed graph of meta-objects (nodes) representing the modeling elements used in the assembly, and the appropriate meta-links (edges) among them. By traversing this graph, with OCL expressions, we can collect information on the assembly we want to analyze. Those expressions provide us the distilled information required for our metrics computation. Consider a small excerpt of the `BaseIDL` package of the CCM metamodel, where the relationship between `InterfaceDef` and `OperationDef` is specified (see Fig. 2).

Let us assume we wish to define a metric to count the operations defined within an interface. We can define the functions `Operations` and `OperationsCount` in the context of the **`InterfaceDef`** meta-class, to represent the set of operations available in that interface and the cardinality of that set, respectively:

`InterfaceDef`

```
Operations(): Set(OperationDef) =
   self.contents->select(o |
     o.oclIsKindOf(OperationDef))->collect(oclAsType(OperationDef))->
     asSet()

OperationsCount(): Integer = self.Operations()->size()
```

To support the metrics formalizations, we built a library of reusable OCL functions [32] that includes several utility functions such as these two. We reuse some of the utility functions in the formalizations presented in this paper. Fig. 3 presents the classes where we defined the reusable OCL functions, represented here through their signature.

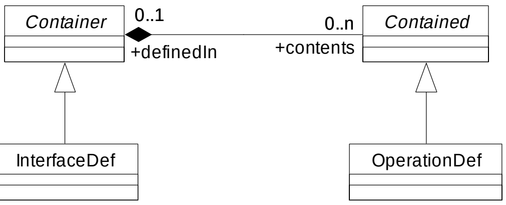
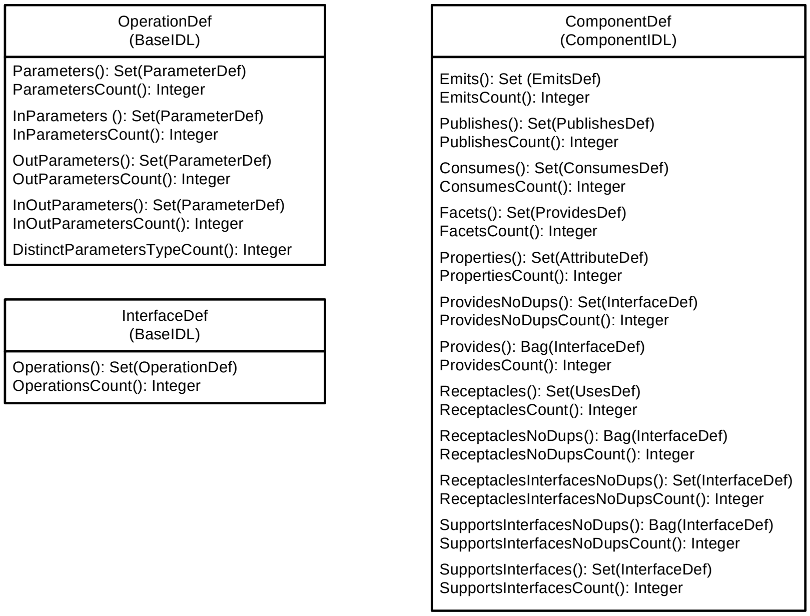


Fig. 2. Interfaces and Operations, in the CCM metamodel



```
OperationDef
(BaseIDL)
Parameters(): Set(ParameterDef)
ParametersCount(): Integer
InParameters (): Set(ParameterDef)
InParametersCount(): Integer
OutParameters(): Set(ParameterDef)
OutParametersCount(): Integer
InOutParameters(): Set(ParameterDef)
InOutParametersCount(): Integer
DistinctParametersTypeCount(): Integer
```

```
InterfaceDef
(BaseIDL)
Operations(): Set(OperationDef)
OperationsCount(): Integer
```

```
ComponentDef
(ComponentIDL)
Emits(): Set (EmitsDef)
EmitsCount(): Integer
Publishes(): Set(PublishesDef)
PublishesCount(): Integer
Consumes(): Set(ConsumesDef)
ConsumesCount(): Integer
Facets(): Set(ProvidesDef)
FacetsCount(): Integer
Properties(): Set(AttributeDef)
PropertiesCount(): Integer
ProvidesNoDups(): Set(InterfaceDef)
ProvidesNoDupsCount(): Integer
Provides(): Bag(InterfaceDef)
ProvidesCount(): Integer
Receptacles(): Set(UsesDef)
ReceptaclesCount(): Integer
ReceptaclesNoDups(): Bag(InterfaceDef)
ReceptaclesNoDupsCount(): Integer
ReceptaclesInterfacesNoDups(): Set(InterfaceDef)
ReceptaclesInterfacesNoDupsCount(): Integer
SupportsInterfacesNoDups(): Bag(InterfaceDef)
SupportsInterfacesNoDupsCount(): Integer
SupportsInterfaces(): Set(InterfaceDef)
SupportsInterfacesCount(): Integer
```

Fig. 3. Reusable OCL functions for CBD metrics collection

4.2 Formalizing Metrics for CBD

In this section, we formalize several metrics for CBD proposed in the literature, using the library introduced in the previous section. For the sake of uniformity, we follow a similar pattern for each metric, or group of related metrics. We start by presenting their *name and original specification* ***(i)***, keeping the notation used by their proponents (thus illustrating the variability of notations commonly used in metrics definitions), and the *metric's rationale* ***(ii)***, in their proponents' view. These are followed by *considerations and assumptions made during the formalization process* and, finally, the *formalization of the metric in OCL* ***(iii)***. The latter may include auxiliary functions. The `AuxilaryFunction` typeface is used to identify these functions. The final expression of the metric is shown within a bounding box.

Component Interface Complexity Assessment. In this section we present the formalization of a selection of metrics concerning the complexity of component interfaces proposed by Boxall and Araban [18]. These metrics aim to assess the understandability of a component interface. Boxall and Araban assume that understandability has a positive influence on the reusability of software components. As such, a higher understandability leads to a higher reusability of the components.

Arguments per Procedure (APP)

(i) The average number of arguments in publicly declared procedures (within the interface) is defined as in (Eq. 1),

$$APP = \frac{n_a}{n_p} \qquad \text{(Eq. 1)}$$

where:
n_a = total count of arguments of the publicly declared procedures
n_p = total count of publicly declared procedures

(ii) The rationale for this metric is that humans have a limited capacity of receiving, processing, and remembering information [33]. The number of chunks of information in the procedure definition (in this case, its arguments) should be limited. Boxall and Araban suggest that an increased number of arguments reduces the interface's understandability and, therefore, its reusability.

(iii) The original proposal of this metric uses C/C++ component interfaces to illustrate the metric definition. Overloaded and overridden procedures (operations, in the CCM) are considered, but not inherited ones. The original metric specification makes no reference to the latter, so we assume them to be outside the scope of this metric. If the component is implemented in an OO language, all public and protected OO methods should be counted, but not the private ones, as these will be invisible to component users.

The metric's definition assumes a single, or at least unified, interface for the component. There is no directly equivalent modelling element in the CCM metamodel. The component equivalent interface is broader, as it includes all implicit operations (a set of operations defined by component homes), operations and attributes which are inherited by the component (also through supported interfaces) and attributes defined inside the component. On the other hand, considering just a single interface as the context would lead to a metric different than the one proposed by Boxall and Araban. To be precise in our formalization, we use the union of procedures in the provided interfaces as the set of procedures to be analyzed.

The context for the metric definition is **`ComponentDef`**. We start by defining `ProvidedOperations`, the set of operations used in the metrics definition, and `ProvidedOperationsCount`, the cardinality of this set. The formalization of the **`APP`** metric becomes straightforward, with these auxiliary functions.

```
ComponentDef

ProvidedOperations(): Set(OperationDef) =
   self.ProvidesNoDups()->collect(Operations())->flatten()->asSet()

ProvidedOperationsCount(): Integer = self.ProvidedOperations()->size()
```

```
NA(): Integer =
   self.ProvidedOperations()->collect(ParametersCount())->sum()

NP(): Integer = self.ProvidedOperationsCount()
```

```
APP(): Real = self.NA()/self.NP()
```

Distinct Argument Count (DAC), and Distinct Arguments Ratio (DAR)

(i) The number of distinct arguments in publicly declared procedures (*DAC*) is defined as in (Eq. 2). Its percentage on the component interface (*DAR*) is defined as in (Eq. 3),

$$DAC = |A| \qquad \text{(Eq. 2)}$$

where:
A = set of the `<name,type>` pairs representing arguments in the publicly declared procedures
|*A*| = number of elements in the set *A*.

$$DAR = \frac{DAC}{n_a} \qquad \text{(Eq. 3)}$$

where:
n_a = total count of arguments of the publicly declared procedures

(ii) *DAC* is influenced by the adoption of a consistent naming convention for arguments in the operations provided by a component. If the same argument is passed over and over to the component's operations, the effort required for understanding it for the first time is saved in that argument's repetitions throughout the interface. The smaller the number of distinct arguments a component user has to understand, the better. Likewise, a lower *DAR* leads to a higher understandability. However, unlike *DAC*, *DAR* is immune to the size of the interface, because its value corresponds to *DAR*, when normalized by n_a.

(iii) Boxall and Araban consider a parameter as a duplicate of another if the pair `<name, type>` is equal in both arguments. `ExistsNameType` returns `true` if a duplicate of the parameter is found in a set of parameters. `DistinctArguments` returns the list of arguments used in the provided interfaces operation signatures, without duplicates. Finally, **`DAC`** computes the distinct arguments count and **`DAR`** their percentage in the component interface.

ParameterDef

```
ExistsNameType(s:Set(ParameterDef)): Boolean =
   s->exists((self.identifier=identifier) and (self.idlType = idlType))
```

ComponentDef

```
DistinctArguments(): Set(ParameterDef) =
   self.ProvidedOperations().Parameters()->
     iterate(p: ParameterDef;
             noDups: Set(ParameterDef) = oclEmpty(Set(ParameterDef)) |
       if (not (p.ExistsNameType(noDups)))
       then noDups->including(p)
       else noDups
       endif)
```

```
DAC(): Integer = self.DistinctArguments()->size()

DAR(): Real = self.DAC()/self.NA()
```

Argument Repetition Scale (ARS)

(i) The *ARS* aims to account for the repetitiveness of arguments in a component's interface (Eq. 4). In other words, it is used for measuring the consistency of the naming and typing of arguments within the publicly declared procedures of an operation.

$$ARS = \frac{\sum_{a \in A} |a|^2}{n_a} \qquad \text{(Eq. 4)}$$

where:
A = set of the `<name,type>` pairs representing arguments in the publicly declared procedures
$|a|$ = count of procedures in which argument name-type a is used in the interface
n_a = argument count in the interface

(ii) The rationale for this metric is that the repetitiveness of arguments increases the interface's understandability, and, therefore, the component's reusability. According to Boxall and Araban, $|a|$ is squared in this definition to create a bias that favors consistent arguments definitions in the interface. Interfaces with a higher ARS *"will tend to be dominated by fewer distinct arguments which are repeated more often"*.

(iii) We define two auxiliary functions `aCount` and `Sum_A`, which compute the count of procedures in which argument is used, and the sum of the squares of `aCount`.

```
ComponentDef
aCount(a: ParameterDef): Integer = self.ProvidedOperations()->
   select(o: OperationDef | a.ExistsNameType(o.Parameters()))->size()

Sum_A (): Integer = self.DistinctArguments()->collect(p|
   aCount(p)*aCount(p))->sum()
```

```
ARS(): Real = self.Sum_A()/self.NA()
```

Component Packing Density. The metric presented in this section was proposed by Narasimhan and Hendradjaya and aims at assessing the complexity of a component, with respect to the usage of a given mechanism [21].

Component Packing Density (CPD)

(i) The *CPD* represents the average number of constituents of a given type (e.g. lines of code, interfaces, classes, modules) in a component (Eq. 5).

$$CPD_{constituent\_type} = \frac{\# < constituent >}{\# components} \qquad \text{(Eq. 5)}$$

where:
constituent_type = type of the constituents whose density is being assessed
#<constituent> = number of elements of *constituent_type* in the assembly
#components = number of components in the assembly

(ii) A higher density indicates a higher complexity of the component, thus requiring, as Narasimhan and Hendradjaya suggested, a more thorough impact analysis and risk assessment. CPD can be defined for a multitude of different constituents, but most of those suggested by Narasimhan and Hendradjaya are not available for users of black-box components.

(iii) We exemplify a possible formalization of this metric, considering the number of operations in the provided interfaces as the constituent type, which is only one of the possibilities. The **`CPD`** function formalizes the metric definition in OCL. To compute the number of operations made available by each component, we reuse the auxiliary function `ProvidedOperationsCount`, which we presented earlier (in the formalization of Eq. 1). Finally, the formalization of this metric is performed using a different context than the previous ones. CPD is computed within the scope of a module (represented by **`ModuleDef`**, in the metamodel), rather than the one of an individual component.

```
ModuleDef

Components(): Set (ComponentDef) =
   self.contents->select(oclIsKindOf(ComponentDef))->
   collect(oclAsType(ComponentDef))->asSet()

ComponentsCount(): Integer = self.Components()->size()

ConstituentsCount(): Integer = self.Components()->
     collect(ProvidedOperationsCount())->sum()

CPD(): Real = self.ConstituentsCount()/self.ComponentsCount()
```

4.3 Increasing the Coverage of the Metrics Set

The metrics formalized so far in this paper focus mainly on the provided interfaces of components. In what concerns their formal definition, adapting these metrics to the required interfaces of components would be straightforward. Rather than using the `ProvidedOperations` auxiliary function, we would replace it by the `RequiredOperations` function. This illustrates the flexibility of the formalization approach. Naturally, these adaptations would only make sense within the scope of a goal-driven extension of the metrics set, to cover quality attributes that are not being sufficiently assessed by these metrics.

To increase the coverage of this metrics set, we include 3 extra metrics, based on simple counts provided by our metrics collection library, so that we can also assess the complexity of understanding the events emitted and consumed, as well as the one resulting from the configurability of each component. Since we are proposing these metrics ourselves, we provide the definition directly in OCL. Therefore, we only present their definition and rationale.

Event Fan-In (EFI), Event Fan-Out (EFO) and Configurable Properties Count (CPC)

(i) The EFI represents the number of Events emitted or published by a component. Conversely, the EFO represents the number of Events consumed by the component. CPC counts the number of configuration properties in each component. Their formal definition in OCL is as follows:

ComponentDef

```
EFI(): Integer = self.PublishesCount() + self.EmitsCount()
EFO(): Integer = self.ConsumesCount()
CPC(): Integer = self.PropertiesCount()
```

(ii) For **`EFI`** and **`EFO`**, the understandability of the component interaction with other components gets lower as the number of events gets higher. In other words, a higher complexity leads to a lower understandability. The same applies to **`CPC`**. More configurable properties imply a higher complexity in configuring a component. On the other hand, they also increase the flexibility of its configuration. In this last example, stresses that sometimes we may find that the same metric may be useful for assessing conflicting quality attributes.

5 Metrics Collection Example

5.1 The Elevator Control System Example

Consider the elevator control system depicted in Fig. 4, with 4 components: `Alarm`, `MotorsController`, `ElevatorsController` and `RequestManager`.

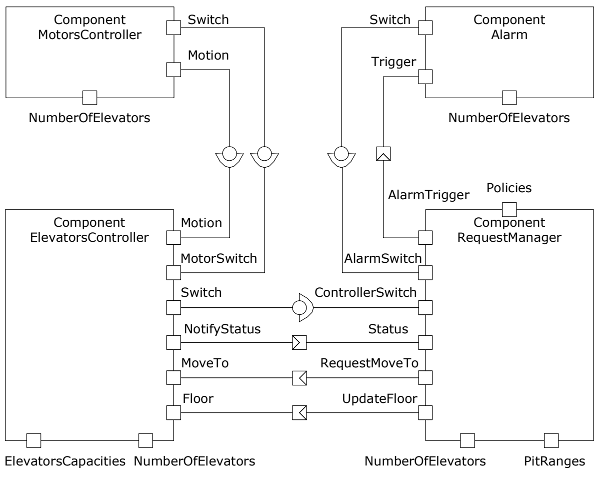


Fig. 4. The Elevator CCM assembly

The `RequestManager` is responsible for handling the requests of the elevator users and sending adequate instructions to the `ElevatorsController` component, as well as handling any interactions with the `Alarm`. The `ElevatorsController` send orders to the `MotorsController`, and notifies the `RequestManager` of any change in the elevators' status. `MotorsController` controls the elevator's motion, ordering it to move up or down, at different speeds, and stop.

The following IDL definitions complement the information on the assembly. For each interface we identify the facet(s) that provide it. For each event type we identify the event source that emits it. The components' configurable properties have the following types: `Policies` is of type `PolicyType`; `PitRanges` is of type `PitRangeSeq`; `Capacities` is of type `ShortSeq` and the several `NumberOfElevators` properties are of type `Short`.

```
enum StatusType {Stopped, Moving, Overload};

enum PolicyType {Closer, Direction};

struct PitRangeType {
   short lower;
   short upper; }


typedef PitRangeSeq sequence <PitRangeType>;

typedef ShortSeq sequence <short>;

interface ISwitch {
   void On(in short motor);
   void Off(in short motor);
} // Used in the several Switch facets

interface IMotion {
   void Up(in short motor, in double speed);
   void Down(in short motor, in double speed);
   void NewSpeed(in short motor, in double speed);
   void Stop(in short motor);
} // Used in the Motion facet

eventtype UpdateStatusEvent {
   public short elevator;
   public StatusType theStatus;
} // Used in the NotifyStatus event source

eventtype MoveRequestEvent {
   public short elevator;
   public short theFloor;
} // Used in the RequestMoveTo event source

eventtype UpdateFloorEvent {
   public short elevator;
   public short theFloor;
} // Used in the UpdateFloor event source

eventtype AlarmTriggerEvent {
   public short elevator;
} // Used in the AlarmTrigger event source
```

5.2 Metrics Results

We computed the formalized metrics for the elevator example by loading the CCM metamodel with the metadata representing the elevator example, and then calculating the results of the OCL expressions presented in section 4.

Table 1 summarizes the metrics values for each of the components. Please note that as some of the metrics are computed as ratios, it is not possible to compute them when the denominator is 0. Those cases are written as N/A.

The remaining metric, CPD, is computed for the whole component assembly. Its value is 2,50. It should be noted that the goal of this paper is not to validate these metrics, but rather to show how they can be formalized with OCL upon the CCM metamodel. The elevator toy example is deliberately simple, to illustrate the metrics computation. With a real-world example, the manual collection of these metrics would require too much effort and be an error-prone task. Automated metrics collection is essential, if they are to be used by practitioners.

Table 1. Metrics for the Elevator example

Context	APP	DAC	DAR	ARS	EFI	EFO	CPC
MotorsController	1,50	2	0,22	5,00	0	0	1
ElevatorsController	1,00	1	0,50	2,00	2	1	2
Alarm	1,00	1	0,50	2,00	1	0	1
RequestManager	N/A	0	N/A	N/A	1	3	3

To illustrate the metrics computation, consider the `MotorsController` component. The following OCL expressions show the partial results of the function calls involved in the APP's metric computation. For simplicity, let us assume that the meta-objects have the same name as the concepts they represent, preceded by an underscore (e.g. the motor controller component is represented by a meta-object called `_MotorsController`, the `Off` operation is represented by the meta-object `_Off`, and so on).

```
_MotorsController.ProvidedOperations()= {_On, _Off, _Up, _Down,
                                         _NewSpeed, _Stop}
_MotorsController.ProvidedOperationsCount()= 6
_MotorsController.NA()= 9
_MotorsController.NP()= 6

_MotorsController.APP()= 1.5
```

6 Discussion

This section enumerates the six problems identified in section 1.3 and addresses how our approach helps dealing with them.

6.1 Quality Framework

Without a clear notion of the quality attributes we wish to assess and the criteria we will use to interpret the metrics values, it is not possible to analyze the results. Although the authors of the proposed metrics provide a rationale for them, the lack of a well-defined quality framework is noticeable.

When analyzing the values presented in Table I, based on the rationale presented during their formalization, one can only make relative judgments on their values. For instance, from the point of view of these metrics, the understandability of components `ElevatorsController` and `Alarm` is similar in what concerns their provided interfaces, but `ElevatorsController` emits and consumes more events, and has more configuration parameters. So, the overall interaction with this component is expected to be more complex than with the `Alarm` component.

6.2 Ontology

The lack of an adequate ontology in the original metrics definitions justifies our need to include several comments on the assumptions made before formalizing each metric (see section ***(ii)*** of all metrics formalizations). An ontology clarifies the used concepts and their interrelationships, providing a backbone upon which we can formalize the metrics definitions with OCL. The combination of the ontology with the OCL expressions removes the subjectivity from the metrics definitions. The ontology is also useful for the automation of metrics collection. In this case, the CCM metamodel was used as an ontology.

6.3 Specification Formalism

We deliberately used the original formalisms in metrics definitions (see section *(i)* of all metrics formalization) to illustrate their diversity. For instance, the concept of collection size is conveyed with three different notations in (Eq. 1-5): a plain identifier (e.g. n_a), an identifier between a pair of '|' characters (e.g. |A|), and the # notation (e.g. #<constituents>). (Eq. 4) uses simultaneously two of these notations. This may lead to misinterpretations of the formulae.

Ambiguity resulting from the usage of natural language is also a problem. Suppose that rather than counting provided operations as constituents for the *CPD* metric, we would like to count provided interfaces. It is possible for different components to provide the same interface. In that case, should we count it once, or several times? If we use the informal version of the definition, we might just write *"constituent_type = provided interface"* and be left with an ambiguous definition. Now, consider the two following alternative `ConstituentsCount` function definitions:

```
ModuleDef

-- Constituents as Interfaces with duplicates
ConstituentsCount(): Integer =
   self.Components()->collect(ProvidesCount())->sum()

-- Constituents as interfaces without duplicates
ConstituentsCount(): Integer =
   self.Components()->collect(ProvidesNoDupsCount())->sum()
```

From the formal definition, it is clear that what we mean is "several times" in the first version and "once" on the second one, thus removing the ambiguity. A similar argument can be made for several of the metrics presented in this paper.

6.4 Computational Support

Most of the computational support required for collecting metrics defined in OCL is either publicly available, or can be built with relatively low effort. The core of the computational support consists of an OCL-enabled UML tool (e.g. the USE tool), with the ability to load a metamodel (as a class model) and create instances of those models (e.g., using object diagrams). In this case, we need to load the CCM metamodel and populate it with the appropriate meta-data, representing the CORBA components we want to evaluate. This metamodel instrumentation can be done manually in several UML tools by creating a meta-object diagram. However, it is more practical to develop a script that generates the meta-data from the original component's specifications. A detailed description of a similar architecture can be found in [34], in that case applied to UML 2.0 components.

6.5 Flexibility

By specifying the metrics definitions with OCL we have completely removed the code tangling between the metrics definitions and the tool computing the metrics. The metrics definitions are loaded in the UML tool just as any other OCL expression. Tailoring the metrics set to one's specific needs is, then, a matter of writing new OCL functions, similar to those presented in this paper.

6.6 Validation

To the best of our knowledge, none of the metrics presented in this paper has undergone a thorough validation, so far. Due to the problems presented in sections 6.1 through 6.5, it should become clear that the ideal conditions for independent scrutiny of these metrics were not present in their original definitions. Several plausible interpretations could be provided for each definition, and no tool support was available to collect them. These conditions hampered experimental replicability. By using the approach presented in this paper, independent validation efforts can be carried out, without jeopardizing the comparability of results.

7 Conclusions

We explored the expressiveness of the CCM metamodel as a valuable ontology upon which we can formally define metrics for CBSE, using OCL expressions. We formally defined 5 metrics found in the literature, along with 3 new metrics, so that the resulting set covers most composition mechanisms used in the CCM.

We discussed our technique with respect to the mitigation of recurrent problems with metrics definitions (lack of a quality framework, lack of an ontology, inadequate specification formalism, computational support, flexibility, and insufficient validation). Having a formal and executable definition of metrics for CORBA component assemblies is an enabling precondition to allow for independent scrutiny of such metrics, when combined with an adequate quality framework. While the provided metrics formalization is in itself a contribution to such an independent scrutiny, the formalization technique is amenable to the definition of new metrics, not only for CCM assemblies, but also for other component models and even other domains.

This paper is one of three essays from our team on the formalization of metrics sets for CBSE proposed by other authors. Both this paper and [34] focus on metrics applicable to components in isolation, using different metamodels (CCM and UML 2.0, respectively), to illustrate the expressiveness of the formalization approach. In [35] we present a similar formalization essay focused on component assemblies, using the CCM metamodel. While metrics of the first kind may somehow help component integrators in their selection process, the current components marketplace has not yet achieved the point where quasi-equivalent parts are available from multi-vendor parties as it is common in other engineering fields. Therefore, we believe that metrics for component assemblies, by allowing evaluating the resulting software architectures, will be much more useful in the short term. They can help in the evaluation and comparison of alternative design approaches, on the identification of cost effective improvements and on long term financial planning (total cost of ownership), allowing the computation of estimates on deployment and evolution costs.

References

1. Matena, V. and Hapner, M., *Enterprise JavaBeans Specification 1.1*, Sun MicroSystems, Inc., (1999)
2. OMG, *CORBA Components - Version 3.0*, Specification, Object Management Group Inc., formal/02-06-65, June. (2002)
3. OMG, *UML 2.0 Infrastructure Final Adopted Specification*, Object Management Group, Inc., ptc/03-09-15, September. (2003)
4. OMG, *UML 2.0 Superstructure Final Adopted Specification*, Object Management Group Inc., ptc/03-08-02, August. (2003)
5. OMG, *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanics*, OMG Adopted Specification, Object Management Group Inc., ptc/04-09-01, September. (2004)
6. Crnkovic, I., Schmidt, H., Stafford, J.A., and Wallnau, K.: 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. *ACM SIGSOFT Software Engineering Notes*. Vol. 29 (3): p. 1-7, (2004).
7. Wallnau, K., *Volume III: A Technology for Predictable Assembly from Certifiable Components*, Technical Report, Carnegie Mellon, Software Engineering Institute, CMU/SEI-2003-TR-009, April. (2003)
8. Larsson, M., *Predicting Quality Attributes in Component-based Software Systems*, PhD, Mälardalen University, (2004)
9. Bertoa, M. and Vallecillo, A. Quality Attributes for COTS Components. *In Proceedings of the 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002)*. Málaga, Spain. (2002)
10. Simão, R.P.S. and Belchior, A.D., Quality Characteristics for Software Components: Hierarchy and Quality Guides, in *Component-Based Software Quality: Methods and Techniques*, Cechich, A., Piattini, M., and Vallecillo, A., Editors, Springer. p. 184-206. (2003).
11. McCabe, T.: A Complexity Measure. *IEEE Transactions on Software Engineering*. Vol. 2 (4): p. 308-320, (1976).
12. Halstead, M.: *Elements of Software Science*. Operating and Programming Systems. New York, EUA: Elsevier Computer Science Library / North-Holland1977).
13. Goulão, M. and Abreu, F.B. Software Components Evaluation: an Overview. *In Proceedings of the 5ª Conferência da APSI*. Lisbon. (2004)
14. Basili, V.R., Caldiera, G., and Rombach, D.H., Goal Question Metric Paradigm, in *Encyclopedia of Software Engineering*, Marciniak, J.J., Editor John Wiley & Sons. p. 469-476. (1994).
15. Richters, M.: A UML-based Specification Environment, University of Bremen: http://www.db.informatik.uni-bremen.de/projects/USE (2001)
16. Gill, N.S. and Grover, P.S.: Component-Based Measurement: Few Useful Guidelines. *ACM SIGSOFT Software Engineering Notes*. Vol. 28 (6): p. 4-4, (2003).

17. Sedigh-Ali, S., Ghafoor, A., and Paul, R.A.: Software Engineering Metrics for COTS-Based Systems. *IEEE Computer*. Vol., (2001).
18. Boxall, M.A.S. and Araban, S. Interface Metrics for Reusability Analysis of Components. *In Proceedings of the Australian Software Engineering Conference (ASWEC'2004)*. Melbourne, Australia: IEEE Computer Society. p. 40-51. (2004)
19. Gill, N.S. and Grover, P.S.: Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Software. *Software Engineering Notes*. Vol. 29 (2): p. 4-4, (2004).
20. Washizaki, H., Yamamoto, H., and Fukazawa, Y. A Metrics Suite for Measuring Reusability of Software Components. *In Proceedings of the 9th IEEE International Software Metrics Symposium (METRICS 2003)*. Sydney, Australia: IEEE Computer Society. (2003)
21. Narasimhan, V.L. and Hendradjaya, B. A New Suite of Metrics for the Integration of Software Components. *In Proceedings of the The First International Workshop on Object Systems and Software Architectures (WOSSA'2004)*. South Australia, Australia: The University of Adelaide. (2004)
22. Hoek, A.v.d., Dincel, E., and Medvidovic, N. Using Service Utilization Metrics to Assess and Improve Product Line Architectures. *In Proceedings of the 9th IEEE International Software Metrics Symposium (Metrics'2003)*. Sydney, Australia: IEEE Computer Society Press. (2003)
23. Goulão, M. and Abreu, F.B. Formalizing Metrics for COTS. *In Proceedings of the International Workshop on Models and Processess for the Evaluation of COTS Components (MPEC 2004) at ICSE 2004*. Edimburgh, Scotland: IEE. p. 37-40. (2004)
24. Goulão, M. and Abreu, F.B. Cross-Validation of a Component Metrics Suite. *In Proceedings of the IX Jornadas de Ingeniería del Software y Bases de Datos*. Málaga, Spain. (2004)
25. Abreu, F.B., *Using OCL to formalize object oriented metrics definitions*, Technical Report, INESC, ES007/2001, May. (2001)
26. Baroni, A.L. and Abreu, F.B. Formalizing Object-Oriented Design Metrics upon the UML Meta-Model. *In Proceedings of the Brazilian Symposium on Software Engineering*. Gramado - RS, Brazil. (2002)
27. Baroni, A.L. and Abreu, F.B. A Formal Library for Aiding Metrics Extraction. *In Proceedings of the International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*. Darmstadt, Germany. (2003)
28. Baroni, A.L., Calero, C., Piattini, M., and Abreu, F.B. A Formal Definition for Object-Relational Database Metrics. *In Proceedings of the 7th International Conference on Enterprise Information System*. Miami, USA. (2005)
29. Wang, N., Schmidt, D.C., and O'Ryan, C., Overview of the CORBA Component Model, in *Component-Based Software Engineering: Putting the Pieces Together*, Heineman, G.T. and Councill, W.T., Editors, Addison-Wesley Publishing Company. p. 557-571. (2001).
30. Estublier, J. and Favre, J.-M., Component Models and Technology, in *Building Reliable Component-Based Software Systems*, Crnkovic, I. and Larsson, M., Editors, Artech House. p. 57-86. (2002).
31. OMG, *Meta Object Facility (MOF) Specification (Version 1.4)*, Object Management Group, April. (2002)
32. Goulão, M.: OCL library for metrics collection in CBD using the CCM: http://ctp.di.fct.unl.pt/QUASAR/resources/DataFiles/CCMEA.use (2005)
33. Miller, G.A.: The Magical Number Seven, Plus or Minus Two : Some limits in our Capacity for Processing Information. *The Psychological Review*. Vol. 63: p. 81-97, (1956).

34. Goulão, M. and Abreu, F.B.: Validação Cruzada de Métricas para Componentes. *IEEE Transactions Latin America*. Vol. 3 (1), (2005).
35. Goulão, M. and Abreu, F.B. Composition Assessment Metrics for CBSE. *In Proceedings of the 31st Euromicro Conference - Component-Based Software Engineering Track*. Porto, Portugal: IEEE Computer Society. (2005)

The Architect's Dilemma – Will Reference Architectures Help?

Martin Haft, Bernhard Humm, and Johannes Siedersleben

sd&m Research,
Carl-Wery-Str. 42,
81739 München,
Germany
{Martin.Haft, Bernhard.Humm, Johannes.Siedersleben}@sdm.de

Abstract. Effective standard architectures promise to improve the efficiency of software projects and the quality of resulting systems. However, hardly any standard architecture has become established in practise to date. They are either too general or too specific to be effective – the architect's dilemma. Reference architectures have a less binding character than standard architectures and still are of high value. This paper presents exemplary parts of the Quasar reference architecture for business information systems, the result of more than seven years of architectural research at sd&m.

1 Introduction

Many ideas have been invested in the standardisation of software architectures. Effective standard architectures would have an enormous impact on quality and efficiency: people could work more efficiently, software reuse would be easier, and project risks could be reduced. Effective standard architectures could be a major improvement to the level of industrialization of the software industry.

However, despite enormous efforts, hardly any standard architecture has been established to date. Why is it so hard, and what can we do about it? In this paper, we describe the lessons we have learnt from more than seven years in trying to establish a standard architecture for business information systems at sd&m.

2 Standard Architectures

What is a standard architecture? We define a *standard architecture* for a specific domain, e.g., business information systems, as a set of specified abstract components, their interactions and their interfaces (*standard interfaces*). The standard interfaces are specified by their syntax and semantics. Concrete components, commercial or non-commercial products, either implement the standard interfaces directly or can be wrapped by the standard interfaces using adapters. Application programmers program against the standard interfaces only and never see the concrete product APIs.

3 The Architect’s Dilemma

3.1 History

Effective standard interfaces would be a major break-through in the industrialisation of the software industry, comparable to standardisation in the fields of mechanical and electrical engineering. However, standardisation of software architectures has hardly ever worked.

CORBA [2] started off more than fifteen years ago with the goal to define a standard architecture according to our definition. Today, there are a number of CORBA implementations on the market. However, CORBA is rarely used as architecture proper in industrial projects today. Instead, CORBA implementations are used as middleware products and their use in industrial projects is decreasing constantly.

J2EE [11] and .NET [13] are platforms that have been widely used in industrial projects and their use is increasing. They allow different architectures to be built on top of their platform. However, they are not architectures themselves.

On the other hand, architectural blue prints such as the classical three layer architecture [4] are so vague that they are not standard architectures according to our definition. The same holds for architectural styles. They are at best patterns that can be used at various stages of a given system, but again, they are not standard architectures.

Wouldn’t it be most advantageous if all application programmers used the same standard interfaces for common functionality, such as persisting application objects (object / relational mapping)? Existing products such as TopLink [18], Hibernate [10] and QuasarPersistence [6, 9, 14] exhibit similar APIs after all. At sd&m, we have worked for more than seven years on developing a standard architecture for business information systems [5, 16]. But despite our efforts, we have not succeeded in establishing a standard persistence interface that all application programmers could program against no matter which concrete persistence product was used – not even for such a well-understood domain within our own company!

3.2 Resistance to Standard Architectures

The reasons we have attributed for our failure to adopt standard interfaces in concrete projects are:

– *Acceptance*: the standard interface is seen by the application programmers as yet another API. Why should they learn a new API when they are already experts in the concrete product’s API?
– *Special features*: In most projects, you need special product features at a few points in a system, e.g., database hints for performance optimization in particularly critical areas. We postulate that it is possible to keep standard operative interfaces simple and general and still provide special features via specific administrative interfaces. This aspect is detailed in the subsequent section.

- *Convenience*: Useful product APIs provide a large number of convenience methods that do not add further functionality but combine existing functionality in a convenient way. Standard interfaces are lean on purpose and, hence, do not contain convenience methods.
- *Costs for implementing adapters*: adapters to hide product APIs behind standard interfaces have to be implemented, and can be expensive, e.g., the translation of a product-specific query language like HQL [10] to a standard query language like OQL [15].

3.3 Operative and Administrative Interfaces

Many commercial products cover numerous special features by covering all foreseeable cases in their APIs. As a consequence, those APIs are huge, hard to understand and error-prone – like an expensive camera suited for professional use and not for the average user. Publishing complicated interfaces to large teams is dangerous. There will be wasted learning efforts and in spite of all training, people will make mistakes. One way out of this problem is to split interfaces into two parts: *operative interfaces* accessible to all application programmers and *administrative interfaces* for restricted use by a few experts only.

Let us take a transaction manager as an example. A transaction manager provides operations for committing work or rolling it back giving the user full control of the transaction modes, e. g., optimistic, pessimistic or anything in-between. A better idea is to split the interface into two: an operative interface providing commit and rollback operations only and an administrative interface providing methods for setting the desired transaction strategy. This approach of splitting interfaces has at least two advantages.

1. It not only separates concerns but also roles: everybody has to know about commit and rollback, but only a few have to be bothered with transaction modes.
2. Many systems need just one transaction mode. It is much better to have this mode set at one place in the system rather than trying to ensure that everybody sticks to the right one.

Unfortunately, this pattern of splitting interfaces into operative and administrative ones is not generally known. Furthermore, it is in most cases much harder to achieve than in our simple transaction example.

3.4 The Dilemma

Everybody will agree on the usefulness of standard architectures and interfaces but, in practice, they don't seem to work. Either they are too specific (e.g., J2EE) or too general (e.g., the three-layer architecture). We define this as the *architect's dilemma*. The challenge of providing simple and general operative interfaces while allowing special features via administrative interfaces is at the heart of the architect's dilemma. What can we do about it? We propose *reference architectures* as a weaker form of standard architectures with a less binding character.

4 Reference Architectures

4.1 Definition

We define a *reference architecture* for a specific domain in analogy to the standard architecture definition as a set of specified abstract components, their interactions and their interfaces (*reference interfaces*). Like standard interfaces, reference interfaces are specified by their syntax and semantics. However, concrete components (products like, e.g., TopLink, Hibernate or QuasarPersistence) do not have to either implement the reference interfaces directly or be wrapped by the reference interfaces using adapters. Instead, APIs of concrete products may differ in the following ways from the reference interfaces [9]:

– *Naming*: Interface and method names may differ, e.g., `prepareQuery` (QuasarPersistence) instead of `defineQuery`;
– *Specialisation*: Methods of product APIs may be more specialised than methods of the reference interface. This can be achieved by more strict pre-conditions or more concrete parameter types. Example: expression of the product-specific query language HQL [10];
– *Special Features*: The product APIs may extend the reference interfaces by special features, e.g., explicit locking;
– *Convenience*: convenience methods may be added to product APIs to ease their use by application programmers. Example: `queryOne` in QuasarPersistence combines `defineQuery` and `executeQuery` and returns the first result.

It is the project architect's decision, whether application programmers program against reference interfaces, product APIs or a combination of both.

Reference interfaces feature the following characteristics:

– *Minimal*: they focus on the essence, i.e., the minimal functionality common to all concrete products (the least common denominator). Special cases are not considered. Redundancy is avoided;
– *Complete*: they subsume all functionality necessary for providing the essence of a component;
– *Disjoint*: functionality relevant for different user groups, particularly application programmers, application administrators and technical configurators, are separated in different interfaces.

4.2 Discussion and Example

Reference interfaces condense the essence of the APIs of concrete products. They help when comparing, understanding and using products with their voluminous APIs.

Is that all? Are reference architectures no-brainers? By no means! We shall demonstrate this with a concrete project example.

A major player in the pharmaceutical industry consulted one of the authors on the implementation of a so-called *portal factory*. The portal factory was to be the organisation and technical platform to implement the Internet portals of all subsidiaries world-wide. It was decided that the portal architecture established in one country was to be taken as a start-point (*source architecture*) for the global portal factory architecture. A blue print for the global portal factory architecture had already been worked out (*target architecture*). In the target architecture, a content management system (CMS) and a portal server were to be used. In fact, there was a great overlap in the products chosen from the source and the target architectures. However, the experts had not succeeded in designing and planning the migration from the source to the target architecture.

What was the reason? All architecture diagrams were product-centric, i.e., the boxes of the architecture diagrams represented concrete products. In practise, portal products often bundle a number of functionalities and use different terminology for them or – even worse – use the same terminology for different functionality. As a result, source and target architecture diagrams were incompatible at all levels. See Figure 1 for an exemplary extract.

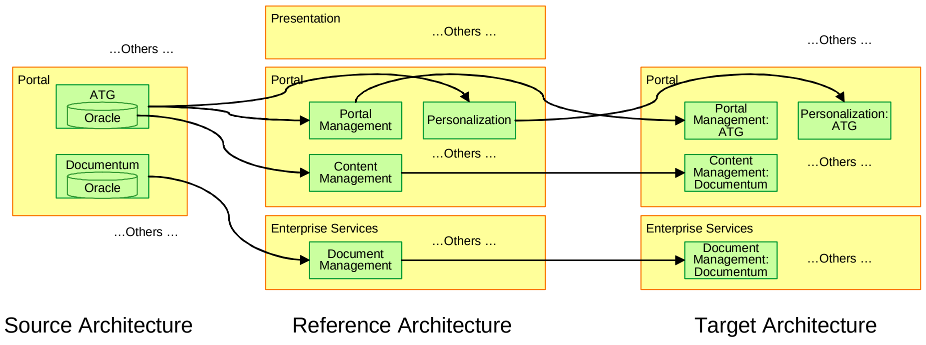


Fig. 1. Migration from source to target architecture via reference architecture

At this point, the author brought into the discussion the sd&m reference architecture for Internet portals. The reference architecture defines the most relevant services of Internet portals – e.g., content management, portal management, and personalization. As a first step, the source and target architecture diagrams were mapped onto the reference architecture. This provided a level of clarity as to which functionality is covered by which product. For example, in the source architecture, the product Documentum [3] was used. Documentum is positioned as a CMS. However, after analyzing details it became evident, that Documentum was not used as the web content management system for this portal but as the enterprise document management system. Web content management for the portal was programmed manually on the basis of the Oracle DBMS. The product suite of ATG [1] was used for portal management, personalization, and other services.

In the target architecture, Documentum and ATG were also to be used. However, this time as packaged solution for web content management, portal management and services like personalization.

Initially, the discussion was about the replacement of products. After the mapping, the discussion was about portal services and about which product or part of a product to cover which service. The result of a three-day-workshop was not only a common understanding of the target architecture but also a defined migration path from the source to the target architecture:

1. Leave the Documentum instance for enterprise document management basically as is;
2. Implement a second Documentum instance for web content management;
3. Re-engineer the application logic for content management, formerly implemented in Oracle technology, now in Documentum technology;
4. Re-engineer the portal server functionality (ATG) to access Documentum as web content management system;
5. Leave the personalization logic implemented in ATG technology basically as is.

In hindsight, the introduction of the sd&m reference architecture for portals was much like the solution of the Gordian knot in this important workshop. In the following years, the target architecture was, indeed, implemented successfully.

5 Reference Architectures and Platform Independence

For some people, the main benefit of a reference architecture is *platform independence*. But neither standard nor reference architectures have anything to do with platform independence. Indeed, what could platform independence possibly mean? MS Windows runs on nearly any hardware you can think of – but would you consider MS Windows as platform independent? The Java Virtual Machine runs on nearly any operation system you can think of – does the use of Java guarantee platform independence? Windows and Java are platforms themselves and once you are caught in a Java or Windows world there is no way out.

There are countless frameworks built on top of databases, application servers and GUI libraries. But in reality, these frameworks replace platform dependence with framework dependence. This can be much worse because numerous frameworks are buggy, hard to understand, awkward to use und fade away as soon as the original authors have left the project. The term "platform independence" has no real meaning and thus cannot be subject of any serious discussion on software architecture.

The question to be addressed is more subtle: How can I minimize the dependencies for every single component of my system? For sure, there will be some platform-dependent components. But, hopefully, they will be few and simple. The best you can think of is a complete independence of anything but the language your system is written in. The Java `String` class has reached this ultimate level of independence. For application classes like `Customer` and `Account` there is no reason whatsoever for not reaching that same level of independence.

One of the main goals of reference architectures is to eliminate all *avoidable* dependencies. They go along with the theory of software categories as explained in [16], Chapter 4.

6 Quasar

Quasar [5, 9, 16, 17] stands for quality software architecture. Quasar as a whole consists of *principles*, *architecture*, and *components* (see Fig. 2) and has been developed at sd&m within the last seven years.

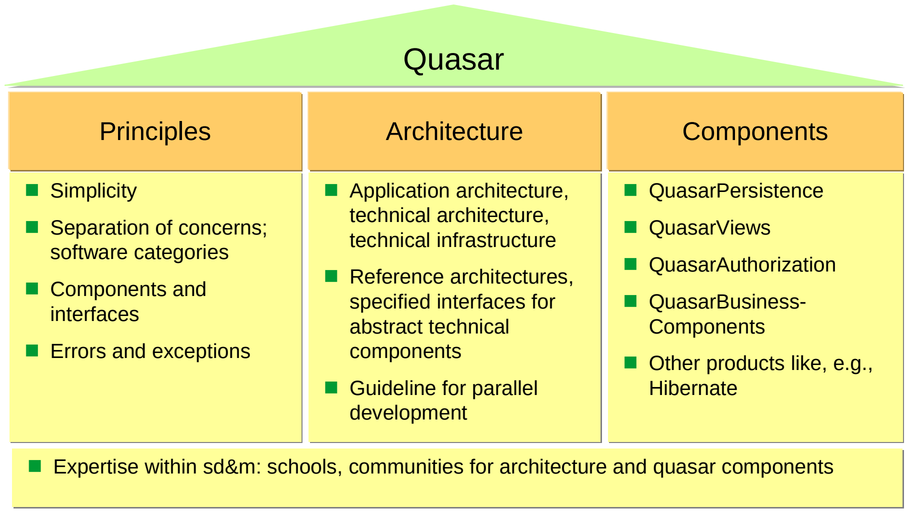

Quasar
Principles
Architecture
Components
Simplicity
Separation of concerns; software categories
Components and interfaces
Errors and exceptions
Application architecture, technical architecture, technical infrastructure
Reference architectures, specified interfaces for abstract technical components
Guideline for parallel development
QuasarPersistence
QuasarViews
QuasarAuthorization
QuasarBusiness-Components
Other products like, e.g., Hibernate
Expertise within sd&m: schools, communities for architecture and quasar components

Fig. 2. Quasar overview

Quasar principles give concrete guidelines for separating concerns in business information systems [5, 16]. Quasar components are concrete, well-proven implementations of the main technical functions of business information systems [6, 9, 14]. This paper focuses on the middle column of Quasar: architecture. In the following sections, we present some exemplary parts of the reference architecture for business information systems and how it is specified. The complete reference architecture can be found in [8].

7 A Reference Architecture for Business Information Systems

7.1 Architecture Overview

Figure 3 gives an overview of the Quasar reference architecture for business information systems as a UML component diagram.

The architecture overview condenses the essence of the architecture of business information systems.

In the centre of the reference architecture are concrete `Application Components` (UML stereotype `<<A Component>>`: ”A” stands for application). Access to those Application Components are wrapped by the `Application Kernel Facade` (UML stereotype `<<Abstract T Component>>`: abstract

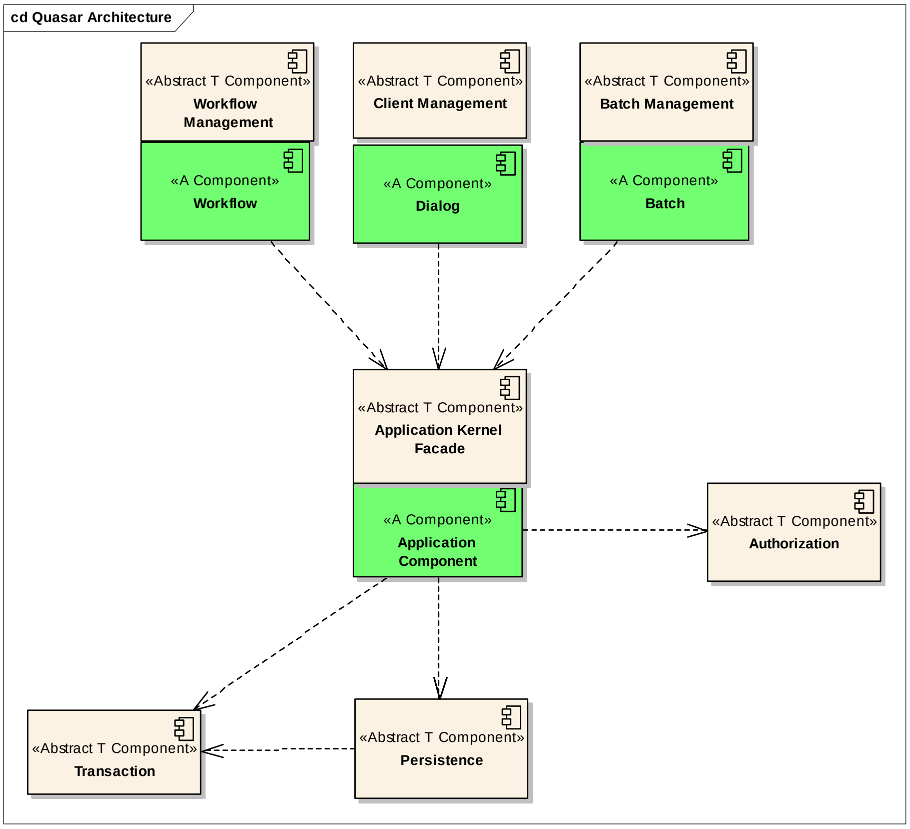


Fig. 3. Quasar reference architecture for business information systems

technical component, the abstraction of a concrete technical product). Services of application components may be invoked by dialogs or workflow applications.

For authentication and authorization purposes, `Application Components` use the abstract technical component `Authorization`. Alternatively, the `Application Kernel Facade` may invoke the `Authorization`. To store and retrieve application data, applications use `Transaction` and `Persistence`.

Concrete `Application Components` may provide transaction control. `Workflows`, `Dialogs` and `Batches` may invoke the services of the `Application Components`. They may also take over transaction control. `Workflows`, `Dialogs`, and `Batches` all use abstract technical components: `Workflow Management`, `Client Management`, and `Batch Management`, respectively.

7.2 Discussion

Important characteristics of the reference architecture overview are:

– *Component-based*: the reference architecture consists of a manageable number of components with defined dependencies;

- *Application and technology separated*: components to implement *application logic* (UML stereotype `<<A Component>>`) are separated from components that provide *technical services* (UML stereotype `<<Abstract T Component>>`). This paper focuses on the reference interfaces of the abstract technical components. In a concrete project architecture, concrete products are used like, e.g., QuasarPersistence, Hibernate, or TopLink;
- *Minimal*: only components are listed that are part of nearly every business information system (least common denominator). Other components which only are occasionally used, like, e.g. a rules engine, are not listed;
- *Acyclic*: dependencies between components form a directed acyclic graph. Cyclic dependencies are avoided.

The reference architecture overview seems self-evident – a no-brainer? By no means! In industrial practise, such a clean separation of concerns on the component level is rare. A project example shall demonstrate that violating this reference architecture may lead to massive project problems. In a project to develop the core application for an Internet auction enterprise, there was initially no clear separation between application entities (application component) and object-relational mapping (technical persistence component) – a naïve use of EJB [11]. With the advent of massive performance problems due to the database access, the application had to be restructured in a major way. This lead to a massive overrun in time and budget.

7.3 Detailed Architecture and Reference Interfaces

In [8], detailed reference architectures including specified reference interfaces are defined for the abstract technical components `Persistence` / `Transaction`, `Authorization`, `Application Kernel Facade` and `Client`. As an example, we present parts of the reference architecture for clients in the following sections.

8 The Reference Architecture for Clients

8.1 Overview

Under the term *client* we subsume all types of user interfaces:

- Graphical and textual
- Web-based (Internet portal) and native (e.g., Java Swing)
- Distributed (client / server) and local

Figure 4 gives an overview of the Quasar reference architecture for clients.

The central component of a client architecture is a `Dialog`. A concrete `Dialog` is application-specific (stereotype `<<A Component>>`). However, its structure and interfaces are application neutral and, hence, part of the reference architecture. A `Dialog` consists of subcomponents `DialogKernel` and `Presentation` with the respective interfaces `DialogKernel` and `Presentation`. `Presentation`

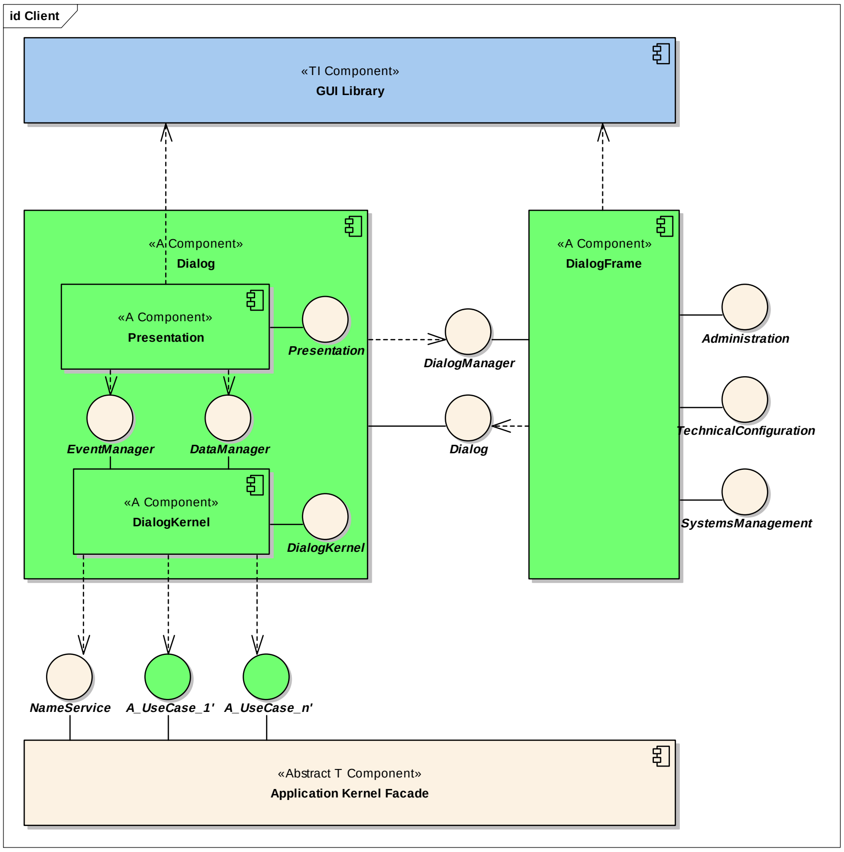


Fig. 4. Reference architecture for clients

manages the visual representation of the `Dialog`, using a technical `GUI Library` (stereotype `<<TI Component>>`: technical infrastructure component). The `Presentation` sends `Events` to the `DialogKernel` via the interface `EventManager` based on user interaction. The `DialogKernel` stores dialog data which is to be accessed (read and written) via the interface `DataManager`.

The `DialogFrame` manages all `Dialogs` of the client. It acts as a factory for `Dialogs`. It provides the interface `DialogManager` and requires the interface `Dialog` from the `Dialog` component. The `DialogFrame` acts as a representative for the whole client and, hence, provides all configuration information:

– `Administration`: interface for the application configuration (administration) of the client. Possible administration items are:
  – *Data management*: administration of data model to be used by the dialog kernel;
  – *Presentation*: administration of the GUI layout;
  – *Application kernel*: administration of the use cases to be invoked by the client;

– `TechnicalConfiguration`: configuration of the concrete products and their dependencies;
– `SystemsManagement`: run-time operating access to start and stop processes and to read and write system parameters.

Use cases of the application kernel are invoked via the `ApplicationKernelFacade` which exhibits application-specific interfaces.

8.2 Dialog States

Part of the reference architecture is the specification of states and state transitions of central objects. Figure 5 shows as an example the state diagram of a `Dialog`.

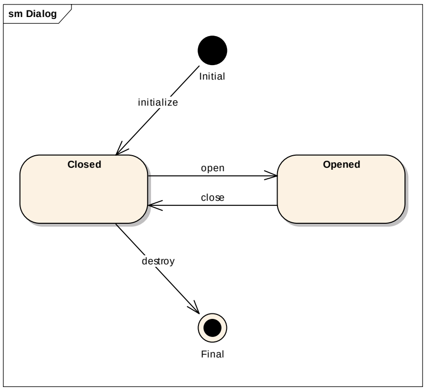


Fig. 5. Dialog state diagram

We distinguish two main states of a `Dialog`:

– `Opened`: the visual representation of the dialog is shown, possibly overlaid by another window;
– `Closed`: the visual representation of the dialog is hidden.

When a `Dialog` is created it is initialized and is then in state `closed`. Methods `open` and `close` (all of interface `Dialog`) perform all necessary actions to show and hide the visual representation. Before destroying a dialog, it must be closed first.

8.3 Interface Specification `DialogManager`

The core of the Quasar reference architecture is the specification of its interfaces. We use the *Quasar Specification Language* (*QSL*, [16]) with a slightly advanced notation. As an example, we show the specification of the interfaces `DialogManager` (Fig. 6) and `Dialog` (Fig. 7). For the specification of the other client interfaces, see [8].

```
interface DialogManager [export, operation]

// Central interface for managing dialog instances.
// The DialogManager makes and releases Dialog
// instances and, thus, acts as a factory for dialogs.
// An implementation of the DialogManager decides which
// type of a dialog is provided (singleton or multiton)




uses
  Dialog,
  DialogType
  DialogState

command

Dialog makeDialog(in DialogType dialogType)

// An instance of a Dialog, denoted by the DialogType,
// is provided.
// Parameters:
//   dialogType - the application-specific type
//   of a dialog, e.g., CustomerManagementDialog.class
// Returns:
//   initialized dialog in state closed

post result.getState() == DialogState.CLOSED

// The Dialog is initialized to state closed.

command

void releaseDialog(inout Dialog dialog)

// An instance of a Dialog is released.
// A released Dialog instance will not be provided
// again. All resources of the dialog will be freed.
// Parameters:
//   dialog - dialog instance to be released

pre dialog.getState() == DialogState.CLOSED

//  to release a dialog it has to be in state closed.
```

Fig. 6. Interface specification `DialogManager`

`DialogManager` is an *operative interface* (keyword `operation`) which is *exported* (keyword `export`). It uses the interfaces `Dialog`, `DialogType` and `DialogState` in its specification. We define two `commands` (as opposed to `simpleQueries` and `derivedQueries`): `makeDialog` and `releaseDialog`. Input parameters are declared as `in`, output parameters as `out`, and input / output parameters as `inout`. For `makeDialog`, a pre condition is specified (`pre`), for releaseDialog a post condition (`post`).

8.4 Interface Specification `Dialog`

Figure 7 shows exemplary the specification of the reference interface `Dialog`.

```
interface Dialog [export, operation]

// This interface defines the view of the DialogManager
// on its dialogs. A dialog managed by the
// DialogManager must implement this interface.
// The Dialog interface has a twofold purpose:
// (1) It defines the life cycle of a dialog
// (2)It provides access to the visual representation
//    of the dialog


uses
  DialogManager,
  DialogState,
  VisualRepresentation

command

void initialize(in DialogManager dialogManager)

// The dialog is being initialized.
// Resources are passed to the dialog.
// Parameters:
//   dialogManager - the DialogManager where
//     this Dialog is registered

post dialog.getState() == DialogState.CLOSED

// The dialog is in the DialogState "closed"


command

void open()

// The dialog is opened. Its visual representation
// is shown and can be used by the end user.

pre dialog.getState() == Dialogstate.CLOSED

// Before the open operation, the dialog must be
// in state "closed"

post dialog.getState() == DialogState.OPENED

// After the operation, the dialog is in state "opened"
```

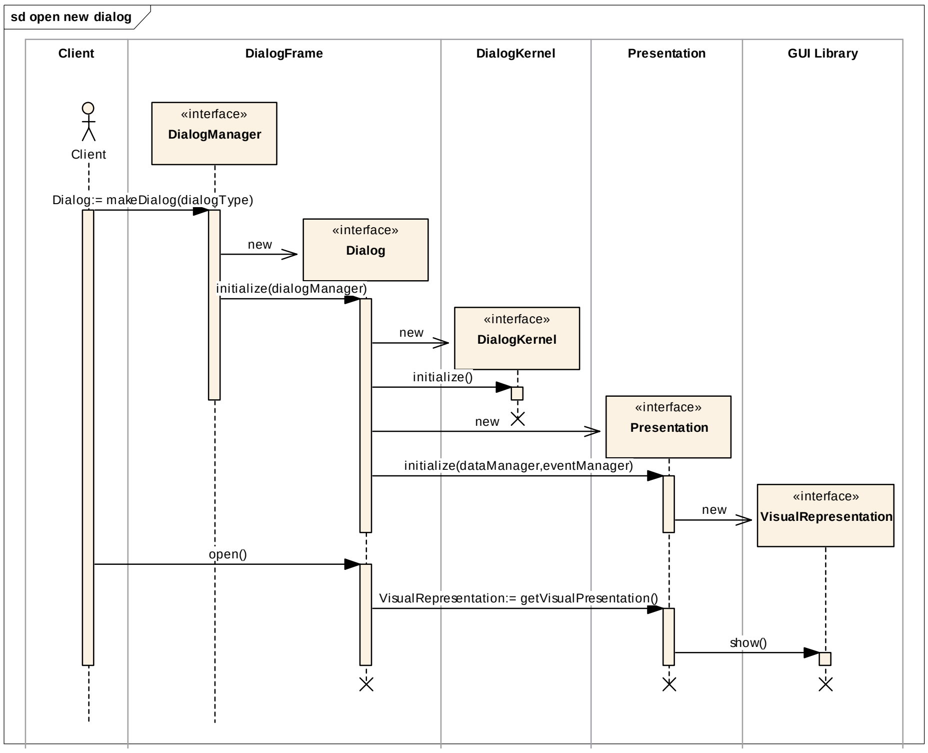
```
post dialog.getVisualRepresentation() != null
// The dialog provides its visual representation


command
void close()
// The dialog is being closed. In this state
// it is invisible to the user and may not be used
// until it is opened, again.
pre dialog.getState() == DialogState.OPENED
// Before the close operation, the dialog must be
// in state "opened"
post dialog.getState() == DialogState.CLOSED
// After the close operation, the dialog is in state
// "closed"


command
void destroy()
// The dialog is destroyed. The dialog may not be
// re-opened again. All resources of the dialog
// are freed.
pre dialog.getState() == DialogState.CLOSED
// Before the destroy operation, the dialog must be
// in state "closed"


basicQuery
DialogState getState()
// The current DialogState (opened or closed)
// of the dialog is returned.
// Returns:
//   Current state of dialog


basicQuery
VisualRepresentation getVisualRepresentation()
// The visual representation of this dialog is returned
// Returns
//   my visual representation
```

Fig. 7. Interface specification `Dialog`

8.5 Client Interactions "Open new dialog"

All typical interactions between client components are illustrated via UML sequence diagrams. Figure 8 shows as an example the opening of a new dialog.

To open a new `Dialog`, it must be made first by the `DialogManager`, the factory for `Dialogs`. The `Dialog` consists of two parts that both need to be created and initialized: `DialogKernel` and `Presentation`. The initialization of the `Presentation` includes the instantiation of a `VisualRepresentation`. Now, the `Dialog` can be opened which includes showing its `VisualRepresentation`.



sd open new dialog
Client
DialogFrame
DialogKernel
Presentation
GUI Library
Client
«interface» DialogManager
Dialog:= makeDialog(dialogType)
new
«interface» Dialog
initialize(dialogManager)
new
«interface» DialogKernel
initialize()
new
«interface» Presentation
initialize(dataManager,eventManager)
new
«interface» VisualRepresentation
open()
VisualRepresentation:= getVisualPresentation()
show()

Fig. 8. Sequence diagram: open new dialog

8.6 Client Interactions "Perform application use case"

Figure 9 demonstrates the interaction between client and application kernel by the sequence diagram "perform application use case".

The sequence diagram starts with the press of a button of a client dialog by an end user. In most GUI Libraries, this results in a message `fireActionEvent` from the `VisualRepresentation` to some `ActionListener`. It then has to send a

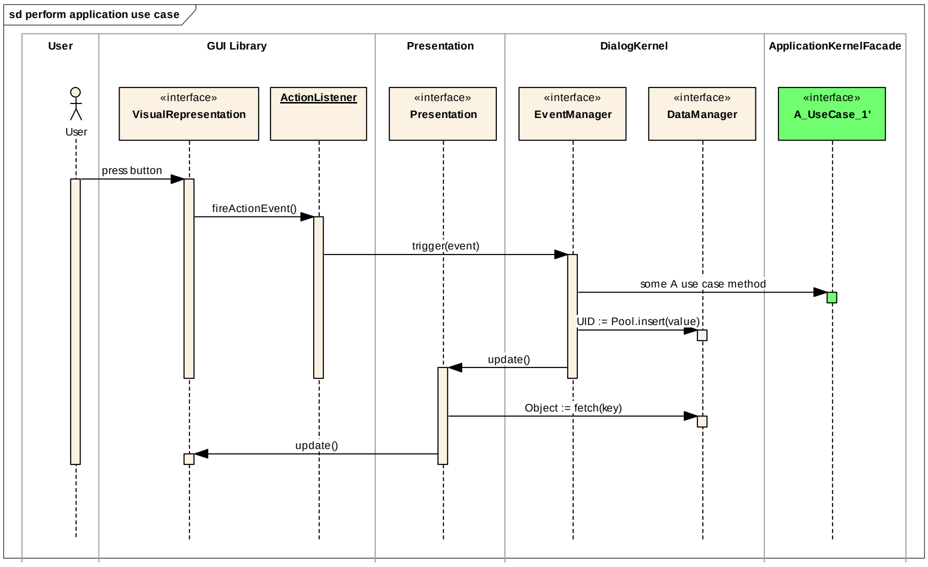


Fig. 9. Sequence diagram: perform application use case

message `trigger(event)` to the reference interface `EventManager` of the `DialogKernel`. Successively, the `EventManager` invokes a concrete application use case method in the `ApplicationKernelFacade`. The `ApplicationKernelFacade` may perform some technical services, e.g., client / server communication, authorization, session and transaction management. It then sends the message to a concrete `ApplicationComponent` which executes the message and returns the result. The result is passed back to the `EventManager` which stores the result in the `DataManager` (method `insert`). Additionally, it informs the `Presentation` on the change (method `update`) which has been registered according to the observer pattern [7]. The `Presentation` may then fetch result from the `DataManager` and update the `VisualRepresentation`.

8.7 Discussion

Client architecture is, to date, the least understood domain in business information systems. Often, client development costs are more than half of the total development cost of industrial business information systems. The costs for finding the adequate project-specific client architecture are enormous. At sd&m, four different client frameworks have been developed and used in more than twenty projects over the last five years. The cost of framework development was tens of person years. The Quasar reference architecture for clients is the result of a one-year long effort of a community of experts and condenses the year-long experience with client development and frameworks. For sd&m, it has been a major investment and we expect a substantial reduction in the costs of client development. It is far from being a no-brainer.

9 Conclusion

Standardisation is a major step towards the industrialisation of the software industry. However, despite substantial effort at sd&m and in many other organisations, standard architectures with specified standard interfaces have hardly ever been established: either they are too specific or too general to be widely applicable – the architect's dilemma. At sd&m, we have deliberately reduced the goal and have established reference architectures instead. Reference architectures reduce the binding character of standard architectures. However, they still have proven benefits in industrial practice.

Reference architectures do not solve the architect's dilemma. Still, they are a most valuable compromise and may act as one step towards standard architectures proper.

References

1. ATG: Art Technology Group. http://www.atg.com
2. CORBA: Common Object Request Broker Architecture. Object Management Group. http://www.omg.org/technology/documents/corba_spec_catalog.htm
3. Documentum enterprise content management system. Emc^2. http://www.documentum.com/
4. Denert, Ernst: Software Engineering – Methodische Projektabwicklung. Springer Verlag Berlin Heidelberg (1991)
5. Denert, Ernst: Siedersleben, Johannes: Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur. Informatik Spektrum (4/2000) 247-257
6. Ernst, Andreas: Quasi-stellares Objekt; Objektbasierte Datenbankzugriffsschicht Quasar Persistence, Javamagazin (3/2004) 85-88
7. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Reading, Massachusetts, USA (1995)
8. Haft, Martin, Humm, Bernhard, Siedersleben, Johannes: Quasar Reference Interfaces for Business Information Systems. Technical Report sd&m Research (December 2004)
9. Humm, Bernhard: Technische Open Source Komponenten implementieren die Referenzarchitektur Quasar. In: Helmut Eirund, Heinrich Jasper, Olaf Zukunft: ISOS 2004 - Informationsysteme mit Open Source, Proceedings GI-Workshop, Gesellschaft für Informatik (2004) 77-87
10. Hibernate: Relational Persistence For Idiomatic Java. http://www.Hibernate.org
11. J2EE: Java 2 Enterprise Edition. Sun Microsystems. http://java.sun.com/j2ee
12. Javadoc: Tool for generating API documentation. Sun Microsystems. http://java.sun.com/j2se/Javadoc
13. .NET: Microsoft Corp. http://www.microsoft.com/net/
14. OpenQuasar: Quasar components. sd&m AG. http://www.openquasar.de
15. OQL: Object Query Language. Object Data Management Group. http://www.odmg.org
16. Siedersleben, Johannes: Moderne Software-Architektur – umsichtig planen, robust bauen mit Quasar. dpunkt Verlag (2004)
17. Siedersleben, Johannes (ed.): Quasar: Die sd&m Standardarchitektur. Parts 1 and 2, 2nd edn. sd&m Research (2003)
18. TopLink: object relational mapper. Oracle Corp. http://www.oracle.com/technology/products/ias/TopLink
19. UML: Unified modeling language. Object Management Group. http://www.uml.org

Architectural Reuse in Software Systems In-house Integration and Merge – Experiences from Industry

Rikard Land[1], Ivica Crnković[1], Stig Larsson[1], and Laurens Blankers[1,2]

[1] Mälardalen University,
Department of Computer Science and Electronics,
PO Box 883, SE-721 23 Västerås, Sweden
[2] Eindhoven University of Technology,
Department of Mathematics and Computing Science,
PO Box 513, 5600 MB Eindhoven, Netherlands
`{rikard.land, ivica.crnkovic, stig.larsson, laurens.blankers}@mdh.se`
`http://www.idt.mdh.se/{~rld, ~icc}`

Abstract. When organizations cooperate closely, for example after a company merger, there is typically a need to integrate their in-house developed software into one coherent system, preferably by reusing from all of the existing systems. The parts that can be reused may be arbitrarily small or large, ranging from code snippets to large self-containing components. Not only implementations can be reused however; sometimes it may be more appropriate to only reuse experiences in the form of architectural solutions and requirements. In order to investigate the circumstances under which different types of reuse are appropriate, we have performed a multiple case study, consisting of nine cases. Our conclusions are, summarized: reuse of components from one system requires reuse of architectural solutions from the same system; merge of architectural solutions cannot occur unless the solutions already are similar, or if some solutions from one are incorporated into the other. In addition, by hierarchically decomposing the systems we make the same observations. Finally, among the cases we find more architectural similarities than might had been expected, due to common domain standards and common solutions within a domain. Although these observations, when presented, should not be surprising, our experiences from the cases show that in practice organizations have failed to recognize when the necessary prerequisites for reuse have not been present.

1 Introduction

Given two high-quality systems with similar purpose and features, how can you create one single system? Can they somehow be merged? In other words, is it possible to reuse the best out of the existing systems and reassemble a new, future system of perhaps even higher quality? This is the challenge and desire of many organizations in today's era of company mergers and other types of close collaborations; this may even be the reason for acquiring a competitor in the first place. The software may be the core products of the companies, or some support systems for the core business. If the software systems are mainly used in-house, performing further evolution and maintenance of two systems in parallel seems wasteful. If the software systems are products of the company, it makes little sense to offer customers two similar products. In either case, the organization would ideally want to take the best out of the existing systems and integrate or merge them with as little effort as possible, i.e. reusing whatever can be reused when building a future system.

However, it can be expected that “architectural mismatch” [8] makes this task difficult. Combining parts of two systems built under different assumptions, following different philosophies, would likely violate the conceptual integrity [3] of the system. It has been observed that qualities largely are determined by the architecture of a system [2], but no matter what the quality of the existing systems are, integrating or merging them – if possible at all – does not necessarily lead to a new high-quality system. There seems to be a range of possibilities in practice, and the choices are influenced by many factors. Possibilities include a tight merge where the old systems are no longer distinguishable, a looser integration, to discontinue some systems and evolve others, or even to not integrate at all but rather start a new development effort, or even (just to be exhaustive about the alternatives) doing nothing but let the existing systems live side by side. The current paper outlines the prerequisites for a tighter merge, in terms of the existing systems (other important points of view are e.g. processes, people, organization, and culture). That is: under what circumstances it is possible and feasible to reuse parts of the existing systems, and when is it more appropriate to only reuse experiences and implement something new?

In order to investigate this we have carried out a multiple case study [21]. It contains nine cases from six organizations working in different domains. The cases are situations where two or more systems were found to overlap and the intention was to create a new system for the future. This paper takes the viewpoint of reuse, and two specific questions are, within the context outlined:

Q1. Which are common experiences (good and bad) concerning reuse when merging two or more systems?

Q2. To what extent are the lessons learned from these experiences possible to generalize into recommendations for other organizations?

The rest of the paper describes the cases and provides answers to these questions. Section 1.1 describes related work, section 1.2 describes the methodology used in the research, and section 1.3 introduces the cases. Section 2 categorizes reuse and section 3 presents reuse in the cases. Section 4 consolidates the observations from all cases, summarized according to the categorization given. Section 5 concludes the paper by summarizing the observations made and outlining future work.

1.1 Related Work

Software integration may mean many different things, and there is much literature to be found. Three major fields of software integration are component-based software [20], open systems [16], and Enterprise Application Integration, EAI [17]. In a previous survey of existing approaches to software integration [13], we found no existing literature that directly addresses the context of the present research: integration or merge of software *controlled and owned within an organization*.

Software reuse usually means finding existing software pieces possible to use in a new situation [10,12]. In the context of the present paper, the procedure is rather the opposite: given two (or more) systems to reuse from, what can be reused? In addition, reuse traditionally concerns reusing implementations, while the cases also mention reuse of experiences, i.e. of requirements and known architectural and design solutions.

1.2 Research Methodology

The multiple case study [21] consists of nine cases from six organizations that have gone through an integration process. Our main data source has been interviews, but in some cases we also had access to certain documentation. In one case (F1) one of the authors (R.L.) also participated as an active member. Details regarding the research design and material from the interviews are available in a technical report [14].

1.3 Overview of the Cases

The cases come from different types and sizes of organizations operating in different domains, the size of the systems range from a maintenance and development staff of a few people to several hundred people, and the types of qualities required are very different depending on the system domain. What the cases have in common though is that the systems have a significant history of development and maintenance.

The cases are summarized in Table 1. They are labelled A, B, etc. Cases E1, E2, F1, F2, and F3 occurred within the same organizations (E and F). For the data sources, the acronyms used are I_X for interviews, D_X for documents, and P_X for participation, where *X* is the case name (as e.g. in I_A, the interview of case A), plus an optional lower case letter when several sources exist for a case (as e.g. for interview I_{Da}, one of the interviews for case D). I_X:*n* refers to the answer to question *n* in interview I_X. The complete copied out interview notes, and details about the participation activities and documents used are found in [14]. In the present paper, we have provided explicit pointers into this source of data.

2 Categorizing Reuse

This section first describes development artefacts (i.e. not only implementations) that can be reused that were mentioned in the cases (section 2.1) followed by a presentation of three basic reuse types that can apply to each of these artefacts (section 2.2).

2.1 What Software Artefacts Can Be Reused?

Although software reuse traditionally means reuse of implementations [12], the cases repeatedly indicate reuse of experience even if a new generation is implemented. In order to capture this, we have chosen to enumerate four types of artefacts that can be reused: requirements, architectural solutions (structure and framework), components and source code. The first two means reuse of concepts and experiences, and the two latter reuse of implementations. We have chosen the terms "component" and "framework", although aware that the terms is often given more limited definitions than is intended here, "component" in e.g. the field of Component-Based Software Engineering [19], and "framework" in e.g. object oriented frameworks [6,9] and component based frameworks [5].

- **Reuse of Requirements.** This can be seen as the external view of the system, what the system does, including both functionality and quality attributes (performance, reliability etc.). Reusing requirements means reusing the experience of features and qualities that have been most appreciated and which needs improvement compared to the current state of the existing systems. (Not discussed in the present paper is the important aspect of how the merge itself can result in new and changed requirements as well; the focus here is on from which existing systems requirements were reused.)
- **Reuse of Architectural Solutions.** This can be seen as the internal view of the system. Reusing solutions means reusing experience of what have worked well or less well in the existing systems. With architectural solutions, we intend two main things:
  - *Structure* (the roles of components and relations between them), in line with the definition given e.g. by Bass et al [2]. Reusing structure would to a large part explicitly recognize architectural and design patterns and styles [1,4,7,18].
  - *Framework.* A definition suitable for our purposes is an "environment which defines components, containing certain rules to which the components must adhere to be considered components (it can be compliance to component models, or to somewhat vaguer definitions)". A framework embodies these rules in the form of an implementation enforcing and supporting some important decisions.
- **Reuse of Components.** Components are the individual, clearly separate parts of the system that can potentially be reused, ideally with little or no modification.
- **Reuse of Source code.** Source code can be cut and pasted (and modified) given the target programming language is the same. Although it is difficult to strictly distinguish between reusing source code and reusing and modifying components, we can note that with source code arbitrary chunks can be reused.

For a large, complex system, the system components can be treated as sub-systems, i.e. it is possible to discuss the requirements of a component, its internal architectural solutions, and the (sub-) components it consists of, and so on (recursively). If there are similar components (components with similar purpose and functionality) in both systems, components may be "merged". We can thus talk about a hierarchical decomposition of systems. Large systems could potentially have several hierarchical levels.

Table 1. Summary of the cases

	Organization	**System Domain**	**Goal**	**Information Resources**
A	Merged international company	Safety-critical systems with embedded software	New Human-Machine Interface (HMI) platform to be used for many products	*Interview:* project leader for "next generation" development project (I_A)
B	Organization within large international enterprise	Administration of stock keeping	Rationalizing two systems within corporation with similar purpose	*Interview:* experienced manager and developer (I_B)
C	Merged international company	Safety-critical systems with embedded software	Rationalizing two core products into one	*Interviews:* leader for a small group evaluating integration alternatives (I_{Ca}); main architect of one of the systems (I_{Cb})
D	Merged international company	Off-line management of power distribution systems	Reusing Human-Machine Inter-face for data-intensive server	*Interviews:* architects/developers (I_{Da}, I_{Db}).
E1	Cooperation defense research institute and industry	Off-line physics simulation	Creating next generation simulation models from today's	*Interview:* project leader and main interface developer (I_{E1}) *Document:* protocol from startup meeting (D_{E1})
E2	Different parts of Swedish defense	Off-line physics simulation	Possible rationalization of three simulation systems with similar purpose	*Interview:* project leader and developer (I_{E2}) *Documents:* evaluation of existing simulation systems (D_{E2a}); other documentation (D_{E2b}, D_{E2c}, D_{E2d}, D_{E2e}, D_{E2f})
F1	Merged international company	Managing off-line physics simulations	Possible rationalization by using one single system	*Participation:* 2002 (R.L.) (P_{F1a}); currently (R.L.) (P_{F1b}). *Interviews:* architects/developers (I_{F1a}, I_{F1b}); QA responsible (I_{F1c})
F2	Merged international company	Off-line physics simulation	Improving the current state at two sites	*Interviews:* software engineers (I_{F2a}, I_{F2b}, I_{F2f}); project manager (I_{F2c}); physics experts (I_{F2d}, I_{F2e})
F3	Merged international company	Software issue reporting	Possible rationalization by using one single system	*Interview:* project leader and main implementer (I_{F3}) *Documentation:* miscellaneous related (D_{F3a}, D_{F3b})

Reusing, decomposing and merging components means that the interfaces (in the broadest sense, including e.g. file formats) must match. In the context studied, where an organization has full control over all the systems, the components and interfaces may be modified, so an exact match is not necessary (and would be highly unlikely). For example, if two systems or components write similar info to a file, differences in syntax can be overcome with reasonable effort, and the interfaces can be considered compatible. However, reuse of interfaces also requires semantic compatibility, which is more difficult to achieve and determine. The semantic information is in most cases less described and assumes a common understanding of the application area.

Although reuse of all artefacts is discussed in the present paper, the focus is on reuse of architectural solutions and components, and on the recursive (hierarchical) decomposition process.

2.2 Possible Primitive Types of Reuse in Software Merge

Assuming there are two systems to be integrated, there are three basic possibilities of reuse: *a)* reuse from both existing systems, *b)* reuse from only one of the existing systems, and *c)* reuse nothing. See Fig. 1. In reality there may be more than two existing systems (I_A:1, I_{E1}:1, I_{E2}:1, D_{F2a}, I_{F3}:1), and more reuse alternatives can easily be constructed by combining these primitive reuse types, i.e. reuse from one, or two, or …, or *n*-1 of the *n* existing systems. For simplicity, we only discuss this in text in connection to the cases.

Different types of reuse can be applied at each of the above mentioned/enumerated artefacts. For example, requirements might be reused from all systems (type *a*), but only the architecture and components of one is evolved (type *b*). An important pattern to search for in the cases is how different types of reuse for different artefacts are related. For example, is it possible to reuse architectural solutions from only one of the existing systems but reuse components from both? If so, under what circumstances?

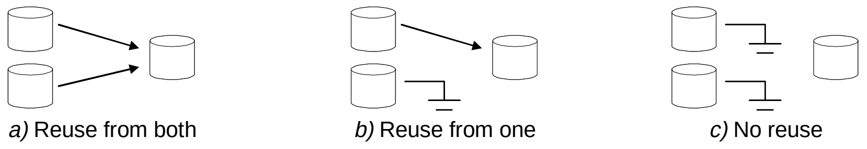


Fig. 1. The three basic types of reuse in the integration context

We want to emphasize that this is a very simple way of describing a complex phenomenon that cannot capture everything. For example, the focus of the paper is on reuse, not new influences. A problem with the three simple types is where to draw the border between type *a*, “reuse from both”, and *b*, “reuse from one”, in the situation when only very little is reused from one of the systems. However, it would be problematic to describe different amounts of reuse – what would “reuse of 34% of the architectural solutions of system A” mean? This difficulty is taken into account in the analysis (section 4), by discussing the implications of classifying each case into either reuse type.

3 Reuse in the Cases

This section will present the type of reuse applied to the different artefacts in all of the nine cases, including considerations and motivations. One of the cases (case F2) is described in depth, followed by more summarized descriptions of the others. More details for all cases can be found in a technical report [14].

The motivation for selecting case F2 for the in-depth description is that being halfway into full integration it represents all types of reuse, at different hierarchical levels. It is also the case with the largest number of interviews made (six), and one of the authors (R.L.) has worked within the company (in case F1) and taken part of information not formalized through interview notes.

3.1 Case F2: Off-Line Physics Simulation

Organization F 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, physics computer simulations are conducted. Central for many simulations made is a 3D simulator consisting of several hundreds of thousands lines of code (LOC) (I_{F2e}:1, I_{F2f}:1). Case F2 concerns two simulation systems consisting of several programs run in a sequence, ending with the 3D simulator (I_{F2a}:1, I_{F2b}:1). The pipe-and-filter architecture and the role of each program is the same for both existing systems, and all communication between the programs is in the form of input/output files of certain formats (I_{F2a}:1,9, I_{F2b}:7, I_{F2c}:10,11, I_{F2d}:8, I_{F2e}:5, I_{F2f}:8). See Fig. 2.

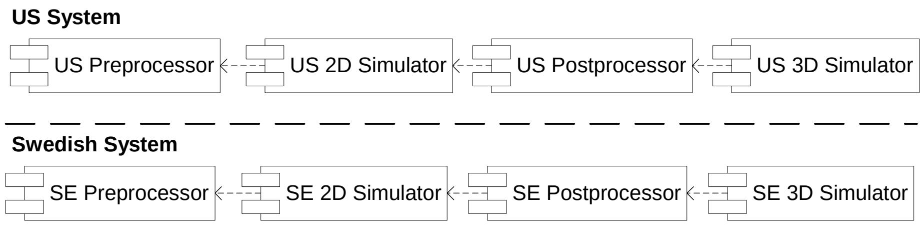


Fig. 2. The batch sequence architecture of the existing systems in case F2. Arrows denote dependency; data flows in the opposite direction.

The 3D simulator contains several modules modelling different aspects of the physics involved. One of these modules, which we can call “X”, needs as input a large set of input data, which is prepared by a 2D simulator. In order to help the user preparing data for the 2D simulator, there is a “pre-processor”, and to prepare the data for the 3D simulator, a “post-processor” is run; these programs are not simple file format translators but involve some physics simulations as well (I_{F2a}:1, I_{F2b}:1).

It was realized that there was a significant overlap in functionality between the two simulation systems present within the company after the merger. It was not considered possible to just discontinue either of them and use the other throughout the company for various reasons. In the US, a more sophisticated methodology for their 2D simulations was desired, a methodology already implemented in the Swedish system (I_{F2a}:3). In the Swedish system on the other hand, fundamental problems with their model had also been experienced (I_{F2a}:3). In addition, there are two kinds of simulations made for different customers (here we can call them simulations of type A and B), one of which is the common type among US customers, the other common among Swedish customers (I_{F2a}:1, I_{F2c}:10). All this taken together led to the formation of a common project with the aim of creating a common, improved simulation system (I_{F2c}:3). The pre-processor, post-processor, and the “X” module in the 3D simulator are now common, but integration of the other parts are either underway or only planned. There are thus still two distinct systems. The current state and future plans for each component are:

- **Pre-processor.** The pre-processor has been completely rewritten in a new language considered more modern (I_{F2b}:1,7). Based on experience from previous systems, it provides similar functionality but with more flexibility than the previous pre-processors (I_{F2b}:7). It is however considered unnecessarily complex because two different 2D simulators currently are supported (I_{F2b}:7,9).
- **2D Simulator.** By evolving the US simulator, a new 2D simulator is being developed which will replace the existing 2D simulators (I_{F2a}:9, I_{F2b}:7, I_{F2d}:7,8). It will reuse a calculation methodology from the Swedish system (I_{F2a}:3, I_{F2c}:9). Currently both existing 2D simulators are supported by both the pre- and post-processor (I_{F2b}:7,9, I_{F2d}:7).
- **Post-processor.** It was decided/assumed that the old Swedish post-processor, with three layers written in different languages, would be the starting point, based on engineering judgments (I_{F2a}:7, I_{F2c}:7, I_{F2d}:6). This led to large problems as the fundamental assumptions turned out to not hold; in the end virtually all of it was rewritten and restructured, although still with the same layers in the same languages (I_{F2a}:9, I_{F2c}:7,9, I_{F2d}:6,7, I_{F2e}:7).
- **3D simulator.** The plan for the (far) future is that the complete 3D simulator should be common (I_{F2a}:3, I_{F2c}:3, I_{F2f}:3). “X” physics is today handled by a new, commonly developed module that is used in both the Swedish and US 3D simulators (I_{F2e}:7). It has a new design, but there are similarities with the previous modules (I_{F2d}:7). In order to achieve this, new data structures and interfaces used internally have been defined and implemented from scratch (I_{F2e}:7, I_{F2f}:6,7); common error handling routines were also created from scratch (I_{F2e}:7); these packages should probably be considered part of the framework rather than a component. All this was done by considering what would technically be the best solution, not how it was done in the existing 3D simulators (I_{F2e}:7,8, I_{F2f}:6). This meant that the existing 3D simulators had to undergo modifications in order to accommodate the new components, but they are now more similar and further integration and reuse will arguably become easier (I_{F2e}:7).

Fig. 3 shows the current states of the systems. Although there are two 3D simulators, some internal parts are common; this illustrates the hierarchical decomposition described in section 2.1 and is visualized in Fig. 4.

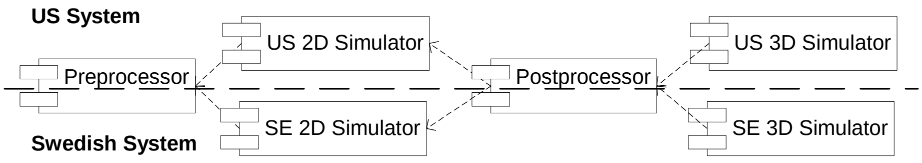


Fig. 3. The currently common and different parts of the systems in case F2

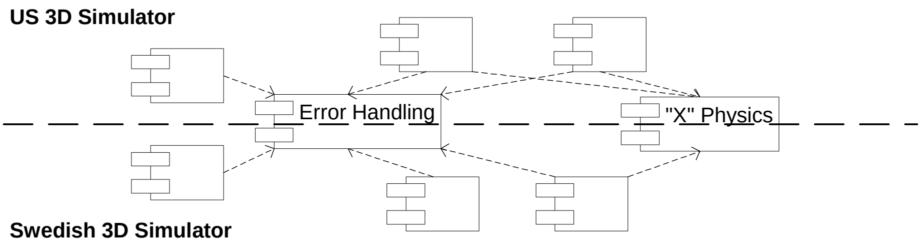


Fig. 4. The currently common and different parts (only exemplified) of the 3D simulators in case F2

3.2 Other Cases

This section presents the most relevant observations in each of the remaining cases.

Case A. Each of the previous separate companies had developed software human-machine interfaces (HMIs) for their large hardware products (I_A:1,2). To rationalize, it was decided that a single HMI should be used throughout the company (I_A:2,3). One of the development sites was considered strongest in developing HMIs and was assigned the task of consolidating the existing HMIs within the company (I_A:2). New technology (operating system, component model, development tools, etc.) and (partly) new requirements led to the choice of developing the next generation HMI without reusing any implementations, but reusing the available experience about both requirements and design choices (I_A:5,6,7). This also included reuse of what the interviewee calls “anti-design decisions”, i.e. learning from what was not so good with previous HMIs (I_A:7). The most important influence, apart from their previous experience in-house, was one of the other existing HMIs which was very configurable, meaning it was possible to customize the user interface with different user interface requirements, and also to gather data from different sources (I_A:3,5).

Case B. One loosely integrated information system had been customized and installed at a number of daughter companies within a large enterprise (I_B:1). In one such daughter company, a tightly integrated system had already been built, but for the future, it should be merged with the loosely integrated system (I_B:3). The total integrated system was stripped down piece by piece, and functionality rebuilt within the framework of the loosely integrated system (I_B:3,7). Many design ideas were however reused from the totally integrated system (I_B:7).

Case C. Two previously competing safety-critical and business-critical products with embedded software had to be integrated (I_{Ca}:1, I_{Cb}:1). The systems and development staff of case C is the largest among the cases: several MLOC and hundreds of developers (I_{Cb}:1,9). The systems' high-level structure were similar (I_{Ca}:7, I_{Cb}:1), but there were differences as well: some technology choices, framework mechanisms such as failover, supporting different (natural) languages, and error handling, as well as a fundamental difference between providing an object model or being functionally oriented (I_{Cb}:1,6,7). Higher management first demanded reuse of one system's HMI and the others underlying software in a very short time, which by the architect's was considered unrealistic and the wrong way to go (I_{Ca}:6,8, I_{Cb}:6). The final decision was to retire one of the systems and continue development of the other (I_{Ca}:6, I_{Cb}:6). Once this decision was made, reusing some components of the discontinued system into the other became easier (I_{Cb}:7). It took some twenty person-years to transfer one of the major components, but these components represented so many person-years that this was considered "modest size" compared to rewrite, although "the solutions were not the technically most elegant" (I_{Cb}:7).

Case D. After the company merger between a US and a European company the two previously competing systems have been continued as separate tracks offered to customers; some progress has been made in identifying common parts that can be used in both systems, in order to eventually arrive at a single system (I_{Da}:1,3, I_{Db}:5,6). As the US system's HMI was considered out of fashion, two major improvements were made: the European HMI was reused, and a commercial GIS tool was acquired (instead of the European data engineering tool) (I_{Da}:1, I_{Db}:3,8). Reuse of the European HMI was possible thanks to its component-based architecture (I_{Da}:1, I_{Db}:3,7,8) and the similarities between the systems, both from the users' point of view (I_{Db}:6) and the high-level architecture, client-server (I_{Da}:7, I_{Db}:7,8). The similarities were partly due to a common ancestry some twenty years earlier (I_{Da}:1). In the European HMI, a proprietary component framework had been built and it has been possible to transfer some functionality from the US HMI by adding and modifying components (I_{Db}:7). The servers are still different and the company markets two different systems (I_{Da}:12, I_{Db}:7).

Case E1. A number of existing simulation models were implemented in FORTRAN and SIMULA, which would make reuse into an integrated system difficult (I_{E1}:6). Also, the new system would require a new level of system complexity for which at least FORTRAN was considered insufficient; for the new system Ada was chosen and a whole new architecture was implemented using a number of Ada-specific constructs (I_{E1}:6,7). Many ideas were reused, and transforming some existing SIMULA code to Ada was quite easy (I_{E1}:7).

Case E2. A certain functional overlap among three simulation systems was identified (I_{E2}:1, D_{E2a}). Due to very limited resources, one of these was retired, and the only integration between the remaining two has been reuse of the graphical user interface of one into the other (I_{E2}:6). Even though the systems belong to the same narrow domain, the same language were used and the integration was very loose, this reuse into the other system required more effort than expected, due to differences in input data formats, log files, and the internal model (I_{E2}:7).

Case F1. After a company merger there was a wish to integrate the software simulation environment within the company, focused around the major 3D simulators used, but also including a range of user interfaces and automation tools for different simulation programs (I_{F1a}:1, I_{F1b}:1, I_{F1c}:1,2, D_{F1a}, P_{F1a}, P_{F1b}). An ambitious project was launched with the goal of evaluating three existing systems; the final decision was to make the system share data in a common database (I_{F1a}:3, I_{F1c}:3, D_{F1a}, P_{F1a}). However, nothing has yet happened to implement it; it appears as this solution was perceived as a compromise by all involved (I_{F1c}:6, P_{F1a}, P_{F1b}). Subsequent meetings has not led to any tangible progress, and four years after the company merger there are still discussions on how to arrive at a future integrated environment, although even this goal itself is being questioned by some of the people concerned (I_{F1b}:3,9, I_{F1c}:6,9). A difficulty has been to integrate the existing data models (I_{F1a}:6, I_{F1b}:6, I_{F1c}:6,7,9, D_{F1a}, P_{F1a}, P_{F1b}). Part of the problem might also be that the scope of such a future system is unclear; discussions include the whole software environment at the departments (I_{F1b}:6,9, I_{F1c}:1, P_{F1b}).

Case F3. Three different software systems for tracking software issues (errors, requests for new functionality etc.) were used at three different sites within the company, two developed in-house and one being a ten-year old version of a commercial system (I_{F3}:1). Being a mature domain, outside of the company's core business, it was eventually decided that the best practices represented by the existing systems should be reused, and a configurable commercial system should be acquired and customized to support these (I_{F3}:6).

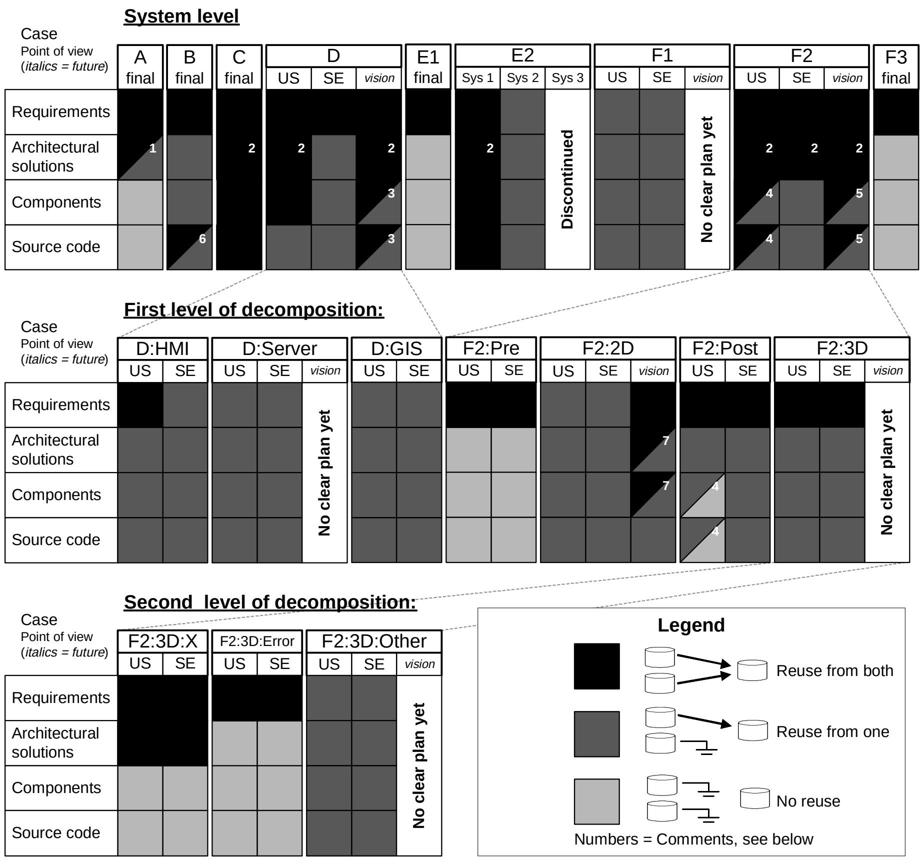
4 Analysis

The cases are summarized in Table 2. In four cases (D, E2, F1, and F2) the systems are not (yet) integrated or merged completely, and there are still two (or more) separate systems deployed, sharing some common parts. This means that if system X has reused something from system Y but not the other way around, and the two systems are still deployed separately, we would have instances of reuse type *a*, "reuse from all" (from X's point of view) and type *b*, "reuse from one" (from Y's point of view). In order to be able to describe this in terms of the "primitive" reuse types described above, reuse is considered from the point of view of the currently deployed systems (where applicable) as well as the future envisioned system (where applicable). In addition, there is a possibility to recursively look into components and consider the requirements, architectural solutions, and (sub-) components of the components, etc. This is done for cases D and F2 where we consider us to have enough material to do so (for case F2 in two levels); for most of the others, this did not make sense when reuse from more than one did not occur.

1. Architectural solutions were reused mainly from one, with heavy influence from one of (several) other systems.
2. The systems had already similar architecture, and the integrated system have thus reused the same solutions from two systems.
3. It is unknown what will be reused in the future integrated system.

4. One component (the post-processor) started out as an attempt to reuse from the Swedish system, but in the end only a small fraction of the original component was left and should probably be considered source code reuse.
5. It is unknown what will be reused in the future integrated system. Comment 5 also applies.
6. It is unsure whether any source code was reused from the retired system (not enough information).
7. The future systems will be an evolution of the US system, while incorporating a methodology from the Swedish system; it is not known whether this means reuse of certain architectural solutions and components (the latter seems unlikely).

Table 2. The types of reuse for the different artefacts in the cases



System level

Case	A	B	C	D			E1	E2			F1			F2			F3
Point of view (*italics = future*)	final	final	final	US	SE	*vision*	final	Sys 1	Sys 2	Sys 3	US	SE	*vision*	US	SE	*vision*	final
Requirements	■	■	■	■	■	■	■	■	▓	Discontinued	▓	▓	No clear plan yet	■	■	■	■
Architectural solutions	■/▓ 1	▓	■ 2	■ 2	▓	■ 2	░	■ 2	▓		▓	▓		■ 2	■ 2	■ 2	░
Components	░	▓	■	■	▓	■/▓ 3	░	■	▓		▓	▓		■/▓ 4	▓	■/▓ 5	░
Source code	░	■/▓ 6	■	▓	▓	■/▓ 3	░	■	▓		▓	▓		■/▓ 4	▓	■/▓ 5	░

First level of decomposition:

Case	D:HMI		D:Server			D:GIS		F2:Pre		F2:2D			F2:Post		F2:3D		
Point of view (*italics = future*)	US	SE	US	SE	*vision*	US	SE	US	SE	US	SE	*vision*	US	SE	US	SE	*vision*
Requirements	■	▓	▓	▓	No clear plan yet	▓	▓	■	■	▓	▓	■	■	■	■	■	No clear plan yet
Architectural solutions	▓	▓	▓	▓		▓	▓	░	░	▓	▓	■/▓ 7	▓	▓	▓	▓	
Components	▓	▓	▓	▓		▓	▓	░	░	▓	▓	■/▓ 7	▓/░ 4	▓	▓	▓	
Source code	▓	▓	▓	▓		▓	▓	░	░	▓	▓	▓	▓/░ 4	▓	▓	▓	

Second level of decomposition:

Case	F2:3D:X		F2:3D:Error		F2:3D:Other		
Point of view (*italics = future*)	US	SE	US	SE	US	SE	*vision*
Requirements	■	■	■	■	▓	▓	No clear plan yet
Architectural solutions	■	■	░	░	▓	▓	
Components	░	░	░	░	▓	▓	
Source code	░	░	░	░	▓	▓	

Legend

■	Reuse from both
▓	Reuse from one
░	No reuse

Numbers = Comments, see below

In the table, for each system we have listed the four artefacts considered (requirements, architectural solutions, components, and source code) and visualized the type of reuse with black for reuse of type *a* “reuse from all”, dark grey for reuse of type *b* “reuse from one”, and light grey for reuse of type *c* “no reuse”. Fields that have not been possible to classify unambiguously are divided diagonally to show the two possible alternative classifications. These and some other fields have been marked with a number indicating a text comment (to be found below the table).

Based on Table 2, we can make a number of observations:

Observation 1. A striking pattern in the table is the transition when following a column downwards from black to dark grey to light grey, but not the other way around (not considering transitions between components and source code). This means that:

- *If it is not possible to reuse requirements from several of the existing systems, then it is difficult, if not impossible, or makes little sense to reuse architectural solutions and components from several systems.*

and:

- *If it is not possible to reuse architectural solutions from several of the existing systems, then it is difficult, or makes little sense to reuse components from several systems.*

There is only one possible exception from these general observations (system F2:2D, see comment 8). We can also note that the type of reuse of architectural solutions very often comes together with the same type of reuse for components. This means that if architectural solutions are reused, components are often reused.

Observation 2. In the cases where "reuse from all" occurred at the architectural solutions level, this did not mean merging two different architectures, but rather that the existing architectures were already similar (comment 2). In the only possible counter-case (case A), the development team built mainly on their existing knowledge of their own system, adapted new ideas, and reused the concept of configurability from one other existing system. This is a strong indication of the difficulty of merging architectures; merging two "philosophies" (I_{E1}:1), two sets of fundamental concepts and assumptions seems a futile task [8]. This means that:

- *For architectural solutions to be reused from several systems, there must either be a certain amount of similarity, or at least some architectural solutions can be reused and incorporated into the other (as opposed to being merged).*

That is, the fundamental structures and framework of one system should be chosen, and solutions from the others be incorporated where feasible.

Observation 3. In case D and F2 where the overall architectures structure were very similar (client-server and batch sequence respectively), the decomposed components follow observations 1 and 2. This means that:

- *Starting from system level, if the architectural structures of the existing systems are similar and there are components with similar roles, then it is possible to hierarchically decompose these components and recursively consider observations 1 and 2. If, on the other hand, the structures are not similar and there are no components with similar purpose and functionality, it does not make sense to consider further architectural reuse (but source code reuse is still possible).*

In the other case with similar system structures (case C) the approach was to discontinue one and keep the other, in spite of the similar structures. The reasons were: differences in the framework, the high quality of both systems, and the very large size of the systems. This shows that architectural structure is not enough for decomposition and reuse to make sense in practice. Nevertheless, in case C it was possible to reuse some relatively small parts from the other system (with some modification). We believe that by considering the roles of the components, it might be possible to find similarities and possibilities for component reuse, even if the structure of the components is dissimilar; this however remains a topic for further research.

Observation 4. In several of the cases, the architectures were similar (cases C, D, F2, and possibly E2). The explanations found in the cases were two: first, in two of the cases the systems had a common ancestry since previous collaborations as far back as twenty years or more (I_{Da}:1, I_{F2a}:1). Second, there seems to be common solutions among systems within the same domain, at least at a high level (e.g. hardware architecture); there may also be domain standards (I_{Cb}:1,7¸ I_{F2a}:1). This is also illustrated by the fact that all instances of similar architectures found were at the system level, in no case within a component in a decomposed system (although we do not think this would be impossible).

Although not directly based on the table, we would like to make an additional remark. When it is not possible to reuse architectural solutions or components it might still be possible to reuse and modify arbitrary snippets of source code. The benefit of this type of reuse is the arbitrary granularity that can be reused (e.g. an algorithm or a few methods of a class) combined with the possibility to modify any single line or character of the code (e.g. exchanging all I/O calls or error handling to whatever is mandated in the new framework). There seems to be a simple condition for reusing source code in this way, namely that the programming language stays the same (or maybe “are similar enough” is a sufficient condition), which should not be unlikely for similar systems in the same domain. Source code thus requires a much smaller set of assumptions to hold true compared to combining components, which require the architectural solutions to be “similar enough” (involving both structure and framework).

We have met the opinion from researchers and software practitioners that these observations confirm intuition, on the border to being trivial. On the other hand, we have also encountered contrary views mainly from management (for example in the cases), that it should be straightforward to reuse and merge the best parts (i.e. implementations) of the existing systems. Our contribution is thus the confirmation of knowledge that some call common sense, but others apparently are not fully aware of.

5 Conclusion

The topic of the present paper is how organizations in control over two or more similar software systems, typically as a result of a company merger or some other close collaboration, can arrive at a single software system. Let us recapitulate the two questions asked in the beginning:

Q1. Which are common experiences (good and bad) concerning reuse when merging two or more systems?

Q2. To what extent are the lessons learned from these experiences possible to generalize into recommendations for other organizations?

The experiences from the cases answer Q1, and can be used as a reference for how specific problems were solved in the past. The observations were structured into three types of reuse (reuse from all, reuse from one, and no reuse) of four artefacts (requirements, architectural solutions, components, and source code), and some general patterns were described on when it seems appropriate to reuse.

Answering Q2 means motivating the external validity of the research. The cases all concern large-scale software systems with development and maintenance histories of many years or decades, and the wide range of domains represented hints at the observations being representative for a large number of integration and merge efforts.

One can argue that studying other cases in slightly other contexts would lead to different results. For example, companies using a product-line architecture approach, or COTS-based development might give different observations concerning reuse. There are also known patterns or best practices that are not directly in line with our observations. For example, the existence of architectural patterns that are reused in completely different systems with completely different requirements could be taken as a contradiction of observation 1 (which says that without reusing requirements, reuse of the architecture is difficult). While this can be true for functional requirements, it is not so with the non-functional requirements; in the case when we use particular architectural patterns we want to provide solutions related to specific concerns (for example reliability, robustness, or maintainability) that might not necessarily be explicitly specified as requirements.

Similarly, observation 2 (which says that the reuse of components is difficult or impossible, without reusing the architecture) may seem to contradict the component-based approach in which the systems are built from already existing components (i.e. components can be developed independently of the systems, and the component developers can completely be unaware of the systems which will use these components). However, it is known that with a component-based approach, an architectural framework is determined and many architectural decisions are assumed. Indeed, it is known that it is very difficult to reuse components that assume different architectural styles or frameworks. In this way, observation 2 confirms the experience from component-based approach.

Finally, we would like to comment on the remark that reuse of source code is opportune even if the components or architecture is not reused. This is of course true, although it is about reuse on a low level, which can very likely result in abuse (for example if the same source code is reused on several places without some synchronization mechanism). This observation might be an indication of a lower maturity of the development organisations, or for a need to encapsulate such code in components. This also suggests that when the new, integrated system is released the existing systems should be discontinued.

With this argumentation, we can claim a certain generality of our observations.

5.1 Future Work

Further cases could reveal more details and would also either support or contradict the observations presented in the present paper. Other case studies and theoretical reasoning could provide further advice on how systems built with different architectural styles and in different frameworks can, and should, be integrated.

To succeed with integration, not only technology is important. We have analyzed the case study material from a process point of view [15], and will continue by describing how to choose between high-level strategies such as merging existing systems, discontinuing some systems and evolve others, starting a new development effort, or even doing nothing but let the existing systems live side by side. The selection among these strategies arguably involves much more than technical aspects; the observations on reuse in the present paper are only one among many influences.

A fundamental question is what *architectural compatibility* or *similarity* means in practice, and how dissimilarities can be overcome; the answer would presumably involve both what we have labelled "structure" (including e.g. architectural styles and patterns) and "framework". Even if the structure is dissimilar, the role of some components may be similar enough to allow for component reuse; the circumstances under which this could be possible need to be identified. One important issue for compatibility could be the notion of *crosscutting concerns* from the field of aspect-oriented programming [11]. The data models of the existing programs would also need to be taken into account, something we have only touched upon in the present paper.

There exist standardized and commercial solutions for integrating information systems (such as enterprise resource planning systems) that has been acquired and cannot be modified (only customized, wrapped, etc.). One question to study is how to choose between a tight merge and a loose integration in the context we have studied, i.e. when such systems have been developed and are fully controlled within a single organization. Such a study would presumably need to focus around data integration.

Acknowledgements

We would like to thank all interviewees and their organizations for sharing their experiences and allowing us to publish them.

References

1. Abowd G. D., Allen R., and Garlan D., "Using Style to Understand Descriptions of Software Architecture", In *Proceedings of The First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.
2. Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.
3. Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition* (20th Anniversary edition), ISBN 0201835959, Addison-Wesley Longman, 1995.
4. Bushmann 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.
5. Crnkovic I. and Larsson M., *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.
6. Fayad M. E., Hamu D. S., and Brugali D., "Enterprise frameworks characteristics, criteria, and challenges", In *Communications of the ACM*, volume 43, issue 10, pp. 39-46, 2000.
7. Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2, Addison-Wesley, 1995.

8. 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.
9. Johnson R. E., "Frameworks = (Components + Patterns)", In *Communications of the ACM*, volume 40, issue 10, pp. 39-42, 1997.
10. Karlsson E.-A., *Software Reuse : A Holistic Approach*, Wiley Series in Software Based Systems, ISBN 0 471 95819 0, John Wiley & Sons Ltd., 1995.
11. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J.-M., and Irwin J., "Aspect-Oriented Programming", In *Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, Springer-Verlag, 1997.
12. Krueger C. W., "Software reuse", In *ACM Computing Surveys*, volume 24, issue 2, pp. 131-183, 1992.
13. Land R. and Crnkovic I., "Existing Approaches to Software Integration – and a Challenge for the Future", In *Proceedings of Software Engineering Research and Practice in Sweden (SERPS)*, Linköping University, 2004.
14. Land R., Larsson S., and Crnkovic I., *Interviews on Software Integration*, report MRTC report ISSN 1404-3041 ISRN MDH-MRTC-177/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, 2005.
15. Land R., Larsson S., and Crnkovic I., "Processes Patterns for Software Systems In-house Integration and Merge - Experiences from Industry", In *Proceedings of 31st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement track (SPPI)*, 2005.
16. Meyers C. and Oberndorf P., *Managing Software Acquisition: Open Systems and COTS Products*, ISBN 0201704544, Addison-Wesley, 2001.
17. Ruh W. A., Maginnis F. X., and Brown W. J., *Enterprise Application Integration*, A Wiley Tech Brief, ISBN 0471376418, John Wiley & Sons, 2000.
18. Schmidt D., Stal M., Rohnert H., and Buschmann F., *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, Wiley Series in Software Design Patterns, ISBN 0-471-60695-2, John Wiley & Sons Ltd., 2000.
19. Szyperski C., *Component Software - Beyond Object-Oriented Programming* (2nd edition), ISBN 0-201-74572-0, Addison-Wesley, 2002.
20. Wallnau K. C., Hissam S. A., and Seacord R. C., *Building Systems from Commercial Components*, ISBN 0-201-70064-6, Addison-Wesley, 2001.
21. Yin R. K., *Case Study Research : Design and Methods* (3rd edition), ISBN 0-7619-2553-8, Sage Publications, 2003.

Supporting Security Sensitive Architecture Design

Muahmmad Ali Babar[1,2], Xiaowen Wang[2], and Ian Gorton[1]

[1] Empirical Software Engineering, National ICT Australia, Australian Technology Park, Alexandria, 1435 Sydney
{malibaba, ian.gorton}@nicta.com.au

[2] School of Computer Science and Engineering, University of New South Wales, Australia
{xiaowen}@cse.unsw.edu.au

Abstract. Security is an important quality attribute required in many software intensive systems. However, software development methodologies do not provide sufficient support to address security related issues. Furthermore, the majority of the software designers do not have adequate expertise in the security domain. Thus, security is often treated as an add-on to the designed architecture. Such ad-hoc practices to deal with security issues can result in a system that is vulnerable to different types of attacks. The security community has discovered several security sensitive design patterns, which can be used to compose a security sensitive architecture. However, there is little awareness about the relationship between security and software architecture. Our research has identified several security patterns along with the properties that can be achieved through those patterns. This paper presents those patterns and properties in a framework that can provide appropriate support to address security related issues during architecture processes.

1 Introduction

Quality is one of the most important issues in software development. It has been shown that software architecture (SA)[1] greatly influences the achievement of various quality attributes (such as security, performance, maintainability and usability) in a software intensive system [1]. That is why a number of formal and systematic techniques have been developed to ensure that the quality issues are addressed early in the software development lifecycle [2-5]. The principle objective of these techniques is to provide support in order to identify the required quality attributes, help design architectures to achieve the desired quality attributes, assess the potential of the designed architecture to deliver a system capable of fulfilling the required quality requirements, and identify potential risks [6].

Security has become one of the most important quality attributes required in networked applications, which support business- and mission-critical processes. The

[1] We use the definition of software architecture provided by Bass et al. [1] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*. 2 ed. 2003: Addison-Wesley.: "*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.*"

reported cases of security breaches have been growing exponentially over the past decade [7]. Software intensive applications are usually composed of several heterogeneous platforms and components provided by different vendors which have different levels of concerns and considerations for security related issues.

Attackers apply highly sophisticated technologies to wage increasingly malicious attacks, which cause different types of damage to the stakeholders of the application. Examples are the unavailability of required medical records for urgent surgery because of a denial of service attack, or financial loses caused by security compromises on the server holding customers' credit card details. Poor quality software has been found the main reason of such vulnerabilities [8-10]. Despite the increasing importance of the security, many systems are designed and implemented without sufficient considerations for security related issues, which are often not dealt with until late in the development lifecycle [10].

Some security related measures may be easily incorporated into an implemented system. For example, adding a password protected login screen. However, making an application sufficiently secure by implementing appropriate and proven security solutions (e.g. security patterns) requires a much greater degree of modification, which may be prohibitively expensive at a late stage of system development. For instance, once security checks have been implemented in various components separately, it is difficult and expensive to introduce the "*check point*" pattern[2] to centralize security policies and algorithms to achieve the "*maintainability*" security attribute.

One of the major reasons for insufficient attention to security issues during early stages of software development is that many system designers do not have the required security expertise. Moreover, software designers and security engineers have different preferences [11, 12]. Security experts are primarily concerned with security, while software designers need to consider not only security but other quality attributes (e.g. performance, usability, and so on) as well. Furthermore, knowledge about techniques addressing security issues has not been captured and documented in a format that is easily accessible and understandable for designers, which makes engineering for security early in the design process difficult [10, 13].

Security engineering experts have developed and validated several known solutions to recurring security issues in the form of security patterns [10, 14]. These security patterns embody expert "wisdom" and raise security awareness among software practitioners. These patterns facilitate effective and efficient reasoning about security issues at a higher level of abstraction [13]. These security patterns prescribe the mechanics of addressing security issues during the software design phase. However, software engineers often do not realize that certain security patterns have architectural implications and cannot be easily applied after a certain stage of the development lifecycle because architectural decisions are harder and expensive to change [1].

The main contribution of the work reported in this paper is a framework for systematically considering and addressing those security issues that need architectural support. This framework presents a set of security attributes and properties along with the design patterns known to satisfy them in an integrated format. The proposed framework can help identify those security sensitive solutions, which cannot be cost-

[2] A list of architecturally sensitive security patterns used in this research and their brief description is provided in Section 3.3.

effectively retro-fitted into an implemented system - rather such solutions are only applicable during the architecture design stage. This framework is expected to be an important source of architecturally sensitive security knowledge to help understand the relationship between software architecture and security related quality attributes and to improve the architecture design and review processes.

The reminder of this paper is organized as follows: In the next section, we discuss the concepts and issues that motivate our research. Section 3 presents a framework for supporting reasoning about security related issues during software architecture design and discusses the elements of this framework: security attributes, security properties, and security sensitive patterns and the usage of the framework. Conclusion and future work complete the paper.

2 Theoretical Background and Motivation

A quality attribute is a non-functional requirement of a software system, such as reliability, modifiability, performance, and usability. According to [15], software quality is the degree to which the software possesses a desired combination of attributes. There are a number of classifications of quality attributes. In [16], McCall listed a number of classifications of quality attributes developed by software engineering researchers. A later classification of software quality is provided in [17]. Quality attributes of large software intensive systems are usually determined by the system's software architecture. It has been widely recognized that quality attributes of complex software intensive systems depend more on the overall architecture of such systems than on other factors such as platform and framework selection, language choice, detailed design decisions, algorithms and data structures [1].

Since software architecture plays a vital role in achieving system wide quality attributes, it is important to address the non-functional requirements during the software architecture process. The principle objective of addressing quality related issues at the architecture level is to identify any potential risks early, as it is quicker and less expensive to detect and fix design errors during the initial stages of the software development [1, 6].

A pattern is a known solution to a recurring problem in a particular context. Patterns provide a mechanism for documenting and reusing the design knowledge accumulated by experienced practitioners [18]. Software architectures of complex and large systems are usually developed using many different patterns. The architectures of such systems evolve by successively integrating several patterns that may be described at different levels of abstractions to deal with particular design problems [19]. One of the main goals of using patterns is to design an architecture with known quality attributes [20]. Each pattern either supports or inhibits one or more quality attributes.

Security patterns capture the security expertise inherent in worked solutions to recurring problems. Security patterns are aimed at documenting the strengths and weaknesses of different approaches in a format easily understandable by those who are not security experts [10, 11]. According to Schumacher, "A security pattern describes a practical recurring security problem that arises in a specific security context and presents a well-proven generic scheme for a security solution" [13]. A security pattern's documentation includes at least four sections: context, problem, solution and security pattern relations.

The security of a system can be characterized in four ways: identity management, transaction security, software security and information security [21]. Each of these focuses on different concerns and requirements. Digital identity management is aimed at defining the users' capabilities according to their respective roles, tracking their actions, and verifying their identity. Transaction security ensures the secure communication between various parts of a system. Software security protects a system from viruses, software pirates, and certain types of hacking. Information security is responsible for protecting the data and information stored in the backend servers like databases and email servers.

In order to satisfy the requirements for each type of the security, there are different techniques such as encrypting data to prevent it being intercepted, altered, or hijacked. It is usually not necessary to devote equal attention to all categories of the security. A software designer should first assess the risks and come up with an appropriate security policy. The design and implementation strategies should reflect the priorities assigned to different categories of the security in the security policy [21, 22].

Another important issue regarding the design of a secure system is careful consideration of several attributes of security requirements, including authentication, authorization, integrity, confidentiality and auditability. How a smart attacker can compromise the security of a system can be demonstrated by a scenario, for example:

"An unauthorized user logs into an E-commerce application as an administrator after several attempts to guess an admin password. This unauthorized user browses through all the sensitive data such as transaction histories and credit card numbers. Moreover, this user creates a new account for themself and gains the highest privilege."

In this single scenario of unauthorized access, all the above-mentioned security aspects have been compromised. The hacker successfully guesses the administrator's password, which violates authentication and authorization. The intruder reads sensitive data, which infringes confidentiality and integrity. Furthermore, several unsuccessful attempts at guessing the password go unnoticed, which breaks down auditability.

Apart from addressing the above-mentioned security aspects, a secure system must also satisfy other security attributes. For example, it should be easy to modify a security policy (maintainability), and should provide secure operations in all circumstances (reliability).

Architectural decisions made by taking security into consideration can sufficiently address most of the security breaches characterized by the above-mentioned scenario. For example, it is possible to modify or integrate the security policy late in the development and avoid huge code rewriting by using a suitable security sensitive architecture pattern like *check point* [14]. However, security expertise embodied by security patterns and its significance for software architecture is not well understood outside the security community. Moreover, software designers usually do not have access to an integrated set of solutions to security issues that needs to be addressed at the software architecture level [10].

3 An Approach to Support Security Sensitive Architecture Design

We have been developing and assessing various techniques to capture and represent quality attribute sensitive architectural solutions in a format that can be an important source of knowledge during architecture processes [23-25]. Apart from our own work on discovering architecture knowledge from patterns, the idea of developing a framework that can help understand the relationship between software quality attributes and software architecture has been inspired by the work of [26, 27], on linking usability and software architecture.

To support the design and evaluation of security sensitive architectures, we decided to systematically analyze existing security patterns in order to identify those patterns which have architectural implications. We have rigorously studied them to understand the mechanics these patterns provide to achieve security. We have also found that there is no standard definition of the security quality attribute. From the security engineering literature, we identified the most common interpretations of security and grouped them in a small set of security attributes (see section 3.1). In order to establish a relationship between security and software architecture, we analyzed several security patterns (see section 3.3) to study their effect on the identified security attributes.

Since it is extremely hard to draw a direct relationship between quality attributes and software architecture [27], we decomposed the identified security attributes into more detailed elements and properties (see section 3.2), which can be considered a form of high level requirements as we can use general scenarios to characterize them [28]. We have put these identified relationships into a framework that relates the problem and solution domains for the security attribute. This framework consists of three layers: attributes, properties, and patterns. Before presenting the framework and discussing the ways in which it can support architecture design and evaluation activities, we briefly describe each of the layers and its elements.

3.1 Security Attributes

We mentioned in Section 2 that a quality attribute is a non-functional requirement of a software system. Bass et al. describe security as a measure of a system's ability to resist unauthorized usage without effecting the delivery of system's services to legitimate users [1]. Our literature review to identify architecturally sensitive security solutions revealed that the security quality attribute has also been described differently by different researchers and practitioners. For example, Singh et. al. [29] describe a security attribute as security concepts/mechanisms, Schneier [30] calls it security needs, and Proctor and Byrne [22] call it security objectives. However, a commonly found concept is that a security quality attribute is a precise and measurable aspect of a system's ability to resist unauthorized usage and different types of attempts to breach security. Having consulted the work of various security experts, we have identified a set of security attributes that need to be sufficiently satisfied by a secure system. We have selected only those attributes, which are most commonly used in the security domain. Following are the security attributes considered in this study.

Authentication: The identity of a system's clients (users or other systems) should be validated to thwart any unauthorized access.

Authorization: This attribute defines an entities' privileges to the different resources and services of a system and limits interactions with resources according to the assigned privileges.

Integrity: There should be a mechanism to protect the data from unauthorized modification while the data is stored in an organizational repository or being transferred on a network.

Confidentiality: A system should guarantee data and communication privacy from unauthorized access. Resource hiding is an important aspect of confidentiality.

Auditability: This means keeping a log of users' or other systems' interaction with a system. Auditability helps detect potential attacks, find out what happened after assaults, and gather evidence of abnormal activities.

Maintainability: Facilitates introducing or modifying a security policy easily during later stages of the software development lifecycle.

Availability: Ensures that authorized users can access data and other resources without any obstruction or disturbance. If a disaster occurs, it ensures that a system recovers quickly and completely.

Reliability: This secures the operations of the system in the wake of failure or configuration errors. It also ensures the availability of the system even when a system is being attacked.

3.2 Security Properties

Software designers apply several design principles and heuristics to achieve different quality attributes [31]. These principles and heuristics are called security properties, which provide a means to link appropriate patterns to a desired quality attribute [27]. In the architecture design stage, architects can consider these properties as requirements, for instance "*the system shall have a robust error handling mechanism*". Since security often conflicts with other quality attributes (such as performance and usability), architects needs to decide how and at which levels these properties are implemented using appropriate security patterns.

For instance, when user inputs an incorrect password, a system should provide an informed error message and guide the user to input a correct password. This is a user friendly mechanism for supporting the user login process. But, how can a system distinguish between a user's mistakes and a malicious attack? Should the system execute a strict security strategy or implement a loose security strategy to improve the usability of the system? Architects can use architecturally sensitive security patterns to achieve the required properties with known affects on other quality attributes [14].

The security properties used in our research have been drawn from the work of different authors in the security domain. We have chosen the most commonly cited security properties in [10, 13, 32]. Following are the properties we considered so far.

Error management: A system should provide a robust error management mechanism to support error avoidance, error handing, fallback procedures and failure logging.

Simplicity: A system should encapsulate initialization check processes, ensure security policy and low-level security, manage permissions and share global information. Systems should also be easy to use and keep the user interface consistent.

Access Control: This property requires the system to support user identification, access verification, least privilege and privacy.

Defense in depth: This includes data verification, reduced exposure to attack, data protection, and communication and information protection.

3.3 Security Patterns

Following on the practice of documenting known solution to a recurring problem in a certain contexts in the form of design pattern, security engineering experts have also discovered and validated several known solutions to recurring security issues in the form of security patterns [10, 14]. A security pattern is supposed to document a particular recurring security problem that arises in a certain context and a well-proven solution to address that problem [13]. A catalogue of such patterns, also known as security pattern language, describes proven solutions to some common security problems. They summarize the strengths and weaknesses of different approaches to address known security issues and make security conscious design knowledge accessible to software designers. These patterns also document the knowledge of the trade-offs that may need to be made in order to use a particular security pattern.

To select the patterns used in our research, we have drawn upon several sources of security engineering knowledge [10, 13, 14, 32], which explain different security patterns with examples and scenarios. The security engineering community has documented many more patterns than we consider here. For example, [14] explains 26 patterns and 3 mini-patterns. However, our research is concerned with only those security patterns that are architecturally sensitive. The list of architecturally sensitive security patterns described in this paper is not exhaustive. We plan to extend this list to identify more patterns, which support security related architectural decisions. These patterns can be organized into an architecturally sensitive security pattern language, which can explicate the relationships between the identified patterns. For example, Yoder and Barcalow [10] describe seven security patterns along with the relationship between them (see Fig. 1). We have found that all these seven patterns are architecturally sensitive and are usually used in tandem to achieve a particular set of security attributes.

In the following, we provide a brief description of each of the security patterns considered in this paper:

Single Access Point: This pattern ensures that there is only one entry point to a system. Anyone accessing the system is validated at the entry point. Having only one entry point makes it easy to perform the initial security checks by encapsulating the initialization process. The drawback of this pattern is inflexibility in terms of non-availability of multiple entry points.

Check Point: This pattern centralizes and enforces security policy and encapsulates the algorithm to put the security policy into operation. The algorithm can contain any number of security checks. This pattern can also be used to keep track of the failed security breaches, which helps take appropriate action if the failures are malicious activities.

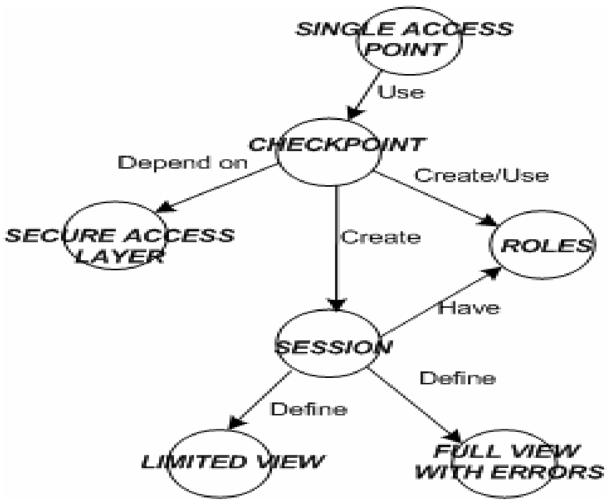


Fig. 1. Architecturally sensitive security pattern system [13]

Roles: This pattern enables the management of the privileges of a system's users at group level. It divides the relationship between a user and their privileges into two new relationships, user-role and role-privilege, which makes the user's privileges easily manageable. However, using the role pattern may result in some complication as implementing roles adds extra complexity for developers.

Session: This creates a session object to store and share variables throughout the system. It is an easy way to share global information among several components of a system. The usage of this pattern needs careful consideration for the propagation of session instance variables and appropriate structure to organize the values stored in the session.

Limited View: This pattern provides a dynamic GUI for different users based on their roles. The users can access content according to their privileges, which prevents unauthorized users from performing illegal operations. However, it can be difficult to realize and training materials for the application must be customized for each set of users.

Full View with Errors: Contrary to *Limited View*, this pattern provides the users of a system with a full view of the GUI. However, it ensures that users can only perform legal operations. In case of illegal operations, an error message is generated to notify the users. This pattern can be easier to implement, however, valid operation can be more difficult to identify as a user can perform any operation. Moreover, frequent error messages usually frustrate a user.

Secure Access Layer: This pattern provides an isolation layer to protect the data and services when integrated with other systems. It encapsulates lower-level security and isolates the developers from any changes made at other levels of security.

Authoritative Source of Data: This pattern is used to verify the validity of data and its origin. It prevents the system from using outdated and incorrect information and reduces the potential risk of processing and propagating fraudulent data.

Layered Security: This pattern is aimed at dividing a system's structure into several layers to improve the security of the system by securing all of the layers. One major drawback of using this pattern is increased complexity at the architecture level.

3.4 Architecture Sensitive Security Framework

3.4.1 Development Process

In order to support security sensitive architectural decisions, the security attributes, properties and patterns identified in our research have been placed in a framework that we call security framework. The security framework is aimed at explicating and instigating systematic reasoning about the relationship between software architecture and security quality attributes. Figure 2 shows the current form of the security framework that captures and presents the relationships that exist among security sensitive attributes, properties, and patterns. To develop a framework for supporting security sensitive architecture design and evaluation, we used several approaches to identify the relationships between security quality attributes, their properties and appropriate patterns, namely:

- We studied many sources on building secure information system (e.g. [22, 30, 33-35]) , and selected the essential quality properties of a system required to satisfy security policies. These are called "security attributes" in this paper.
- We studied several patterns (such as [10-14]) suggested for building and deploying secure systems. We rigorously verified the architectural sensitivity of those patterns for security attributes. We found several patterns are not architecturally sensitive - for example, the "account lockout" and "server sandbox" patterns [14]. The former is a mechanism against a password-guessing attack by limiting the number of incorrect attempts, which are implementation or deployment related decisions depending upon organizational policies. The latter is used to contain any damage by deciding the minimum privileges required to run a web server, which is again an implementation and context dependent decision that can be implemented anytime during the life of an application. This classification exercise helped us select nine patterns that have architectural implications. However, this list is extensible.
- We organized the identified patterns into a template [23] (see Appendix 1 for an example) to summarize their context, forces, available tactics, affected attributes, and supported general scenarios. We reviewed the architecturally sensitive information captured using the template and identified the security properties that can be characterized by the general scenarios supported by each pattern and also can be considered as requirements to achieve security attributes.
- Like [27], we put the discovered relationships into a framework for linking the software architecture and security quality attributes.

The security framework shown in Fig. 2 presents a collection of security related attributes, properties, and patterns along with the links that form the relationship between software architecture and security quality attribute. To understand one of the

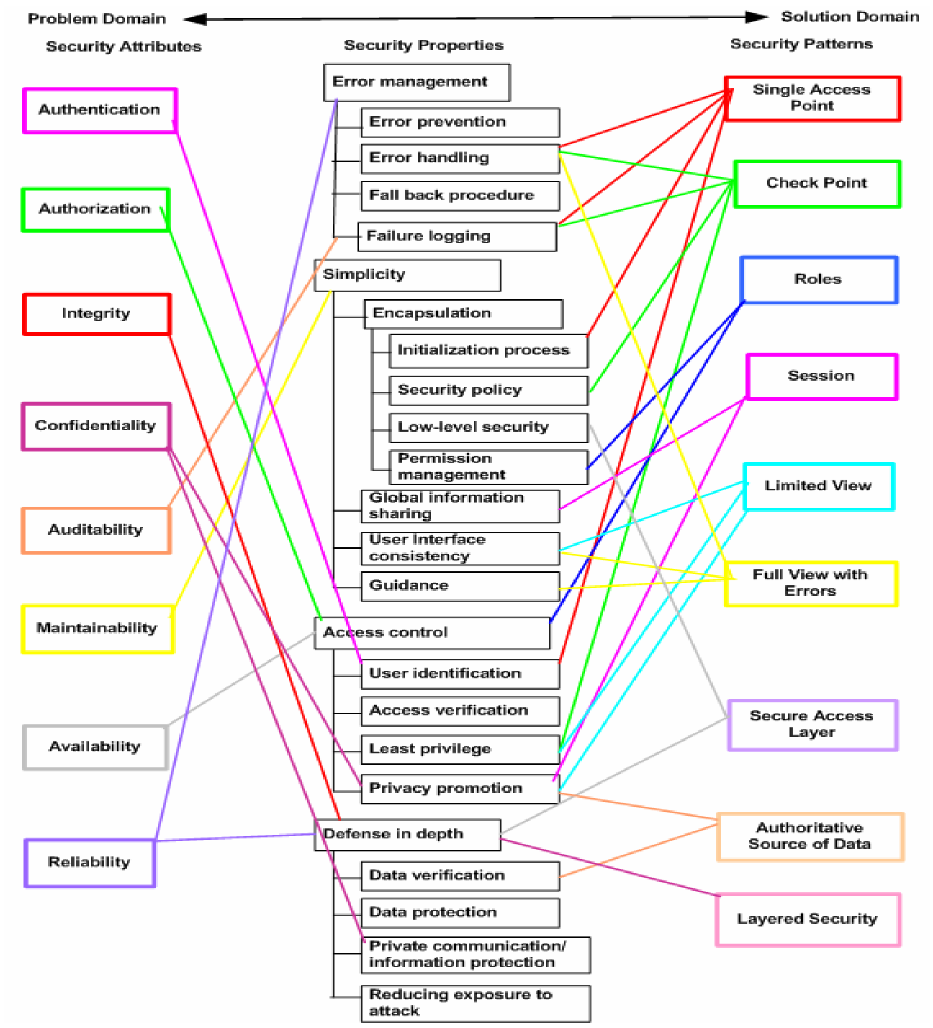


Fig. 2. A Framework for supporting security sensitive architecture design and evaluation

ways of using this framework, let us assume that one of the non-functional requirements to be addressed is ease of modifying the security policy, which can be characterized by the "maintainability" quality attribute presented in the left column of the framework. The next step is to identify the security property (or properties) that characterize "maintainability". According to this framework, the attributes are decomposed into properties, which are placed in the middle column of the framework. This shows that "maintainability" is characterized by the "simplicity" property. This security property itself is decomposed into sub-properties. The security property of interest for this example is "encapsulation of security policy".

The next step is to identify a pattern that promises to satisfy the desired property. These patterns are presented in the right column of the security framework. For this example, the framework makes it obvious that the "check point" pattern promises to support "security policy", which in turn supports the ease of modifying security policy requirement.

This framework is expected to be used as a high level guidance to aid in analyzing the architecturally sensitive security issues and their potential solutions. In this way, the security framework connects the security problem domain with the security solution domain. A security attribute is characterized by one or more security properties, which specify security requirements using scenarios, and patterns are used to satisfy those scenarios and in turn properties. Thus, security patterns provide a mechanism to bridge the gap between the problem and solution domains [27].

The framework also demonstrates that the relationships between patterns, properties and attributes are not necessarily binary. Nor are the relationships necessarily positive, however, to keep the diagram uncluttered, only positive relationships have been shown. For example, the "Limited View" pattern has a negative relationship with the "Guidance" property. This is because different types of guidance material need to be provided to different categories of users of the system, which can increase the cognitive load for a user who belongs to many classes of users. Moreover, it is difficult to implement [13].

3.4.2 Framework Usage

With security attributes decomposed into security properties, suitable patterns identified to satisfy those properties, and relationships among them established, we have constructed a mechanism to support security sensitive architecture design and evaluation. This framework can be used by a design team in a several ways. The team may find a certain security property vital for the secure operations of a system. They may realize the importance of a particular property either by looking at the framework, from the stakeholders' scenarios that characterize a specific property, or from studying a similar system's properties.

Having realized that a particular security property is important for the system to be designed, the team can use the framework to identify the potential patterns that need to be introduced and which security attributes will be affected (positively or negatively). For example, to develop an online virtual training system various levels of access are required depending upon the status of a user (i.e. trainers, students, coach and others). Having realized the necessity of an appropriate access control mechanism, the team can consult the security framework, which shows that the "Roles" pattern is linked to the "access control" security property, which in turn is linked to four security attributes (i.e. authentication, authorization, integrity, and availability). That means if architect needs to achieve the "access control" security property, they can introduce the "Roles" pattern to support the "access control" property. However, the "Roles" patterns needs to be supported by the "single access point", "check point", "session", "Limited view", and "authoritative source of data" patterns to achieve all four security attributes linked with the "access control" property.

In another situation, a design team may find that the "auditability" attribute is required. The team can use the security framework to identify the property that is needed to achieve that attribute, namely "Failure logging" in this case. Having identified the security property that characterizes the desired property "auditability", the team can use the security framework to find out that the “Single Access point” and “Check point” patterns are needed in order to achieve the “auditability” quality attribute. Moreover, the security framework also helps the team understand why those two patterns need to be used in tandem and which other security properties and attributes are expected to be achieved by using all these patterns in tandem.

In a third scenario of potential use of the security framework, a software architecture evaluation team may find out that a particular pattern has been used in the architecture being evaluated. They can use the security framework to identify the properties and attributes that are supported by that pattern. Since security properties are non-functional requirements, the security framework can help the evaluation team easily appreciate the security related non-functional requirements that have been considered in the architecture by using a particular security pattern. For example, if the team finds the “Authoritative Source of Data” pattern being used in the architecture being reviewed, they can use the security framework to see that this pattern promotes the “Data Verification” security property, which is positively related to the “Integrity” and “Reliability” security attributes.

4 Conclusion and Future Work

Our main conclusion is that it is important that security issues are sufficiently taken into account during architecture design because certain security sensitive solution need architectural support and it is very difficult and costly to retro-fit security into an implemented system. We observe a gap between security engineering and architecture engineering knowledge. Our objective is to identify and capture architecturally sensitive security knowledge from different sources, and present it in a format that can bridge that knowledge gap. We have identified an initial set of security attributes, properties and patterns, and put them in a security framework that is expected to help software designers address security issues during software architecture design and evaluation processes.

Our main goal is to improve architectural support for security by providing security knowledge in a format that can support design decisions with an informed knowledge of the consequences of those decisions. More specifically, we intend to raise the awareness about the importance of addressing security related issues during architecture design and review process. Based on our work reported in [23-25], We believe that a systematic approach to identify, capture, and explicitly document the relationships of security attributes, properties, and patterns is an important step towards that goal.

The work reported in this paper cannot be described as complete in its current form as there are a number of things that need to be done before the users - architects/designers/evaluators - of the approach can experience significant benefits in terms of improved realization of security issues during design. These include improved knowledge of the proven security solutions that need architectural support, and ease of identifying and resolving conflicts between security and other quality attributes. In the short term, we plan the following tasks to refine and assess the proposed framework:

- Assess the usefulness and generality of the approach with controlled experiments and case studies.
- Develop a repository to store and access the architecturally sensitive security knowledge.

Acknowledgement. National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

[1] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice. 2 ed. 2003: Addison-Wesley.
[2] Kazman, R., M. Barbacci, M. klein, and S.J. Carriere. Experience with Performing Architecture Tradoff Analysis. Proc. of the 21th International Conference on Software Engineering. 1999. New York, USA: ACM Press.
[3] Kazman, R., L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. Proc. of the 16th ICSE. 1994.
[4] Bosch, J., Design & Use of Software Architectures: Adopting and evolving a product-line approach. 2000: Addison-Wesley.
[5] Boehm, B. and H. In, Identifying Quality-Requirement Conflicts. IEEE Software, 1996. **13**(2): p. 25-35.
[6] Lassing, N., D. Rijsenbrij, and H.v. Vliet. The goal of software architecture analysis: Confidence building or risk assessment. Proceedings of First BeNeLux conference on software architecture. 1999.
[7] CERT. CERT/CC Statistics 1988-2004. Last accessed on 26th February 2005, Available from: http://www.cert.org/stats/cert_stats.html.
[8] Viega, J. and G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way. 2001: Addison-Wesley.
[9] Lamsweerde, A.v. Elaborating Security Requirements by Construction of Intentional Anti-Models. Proc. of the 26th Int'l. Conf. on Software Eng. (ICSE). 2004. Endinburgh, Scotland.
[10] Yoder, J. and J. Barcalow. Architectural Patterns for Enabling Application Security. Proc. of the 4th Pattern Languages of Programming. 1997. Washington, USA.
[11] Kienzle, D.M. and M.C. Elder. Final Technical Report: Security Patterns for Web Application Development. Last accessed on 18th February 2005, Available from: http://www.scrypt.net/~celer/securitypatterns/.
[12] Kienzle, D.M. and M.C. Elder. Security Patterns: Template and Tutorial. Last accessed on 18th February 2005, Available from: http://www.scrypt.net/~celer/securitypatterns/.
[13] Schumacher, M., Security Engineering with Patterns, in Lecture Notes in Compuer Science. 2003, Springer-Verlag GmbH.
[14] Kienzle, D.M., M.C. Elder, D. Tyree, and J. Edwards-Hewitt. Security Patterns Repository - Version 1.0. Last accessed on 18 February 2005, Available from: http://www.scrypt.net/~celer/securitypatterns/.
[15] IEEE Standard 1061-1992, Standard for Software Quality Metrics Methodology. 1992, New York: Institute of Electrical and Electronic Engineers.

[16] McCall, J.A., Quality Factors, in Encyclopedia of Software Engineering, J.J. Marciniak, Editor. 1994, John Wiley: New York, U.S.A. p. 958-971.

[17] ISO/IEC, Information technology - Software product quality: Quality model. ISO/IEC FDIS 9126-1:2000(E).

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

[19] Petersson, K., T. Persson, and B.I. Sanden, Software Architecture as a Combination of Patterns. CrossTalk The Journal of Defense Software Engineering, Oct., 2003.

[20] Buschmann, F., Pattern-oriented software architecture: a system of patterns. 1996, Chichester; New York: Wiley. xvi, 457 p.

[21] Hohmann, L., Beyond Software Architecture: Creating and sustaining winning solutions. 2003: Pearson Education, Inc.

[22] Proctor, P.E. and F.C. Byrnes, The Secured Enterprise: Protecting your information assets. 2002: Prentice Hall PTR.

[23] Ali-Babar, M. Scenarios, Quality Attributes, and Patterns: Capturing and Using their Synergistic Relationships for Product Line Architectures. Proc. of the Int,l. Workshop on Adopting Product Line Software Engineering. 2004. Busan, South Korea.

[24] Zhu, L., M. Ali-Babar, and R. Jeffery. Mining Patterns to Support Software Architecture Evaluation. Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture. 2004.

[25] Ali-Babar, M., B. Kitchenham, P. Maheshwari, and R. Jeffery. Mining Patterns for Improving Architecting Activities - A Research Program and Preliminary Assessment. Proc. of 9th Int'l. conf. on Empirical Assessment in Software Engineering. 2005. Keele, UK.

[26] Bass, L. and B.E. John, Linking usability to software architecture patterns through general scenarios. Journal of Systems and Software, 2003. **66**(3): p. 187-197.

[27] Folmer, E., J.v. Gurp, and J. Bosch, A Framework for Capturing the Relationship between Usability and Software Architecture. Software Process Improvement and Practice, 2003. **8**(2): p. 67-87.

[28] Bass, L., M. Klein, and G. Moreno, Applicability of General Scenarios to the Architecture Tradeoff Analysis Method, Tech Report CMU/SEI-2000-TR-014, Softwar Engineering Institute, Carnegie Mellon University, 2001

[29] Singh, I., B. Stearns, M. Johnson, and E. Team, Designing Enterprise Applications with the J2EE™ Platform. 2002: Addison Wesley Professional.

[30] Schneier, B., Secrets and Lies: Digital Security In a networked world. 2000: Wiley Computer Publishing.

[31] Bass, L., M. Klein, and F. Bachmann. Quality Attribute Design Primitives and the Attribute Driven Design Method. Proceedings of the 4th International Workshop on Product Family Engineering. 2001. Bilbao, Spain.

[32] Romanosky, S. Security Design Patterns. Last accessed on 21th February 2005, Available from: http://www.cgisecurity.com/lib/securityDesignPatterns.pdf.

[33] Feghhi, J., J. Feghhi, and P. Williams, Digital Certificates: Applied Internet Security. 1999: Addison Wesley Longman, Inc.

[34] Juric, M., et al., Patterns Applied to Manage Security, in J2EE Patterns Applied: Real World Development with Pattern Frameworks. 2002, Peer Information.

[35] Ellison, R.J., A.P. Moore, L. Bass, M. Klein, and F. Bachmann, Security and Survivability Reasoning Frameworks and Architectural Design Tactics, Tech Report CMU/SEI-2004-TR-022, SEI, Carnegie Mellon University, USA, 2004

Appendix 1

Table 1. A template to document and analyse architecturally significant information found in a security pattern

Pattern Name: The Check Point Pattern			**Pattern Type:** Security Pattern	
Brief description		The Check Point Pattern centralizes and enforces security policy.		
Context		Design an application with a centralized security policy management.		
Problem description		An application needs to be secure from break-in attempts, and the appropriate actions should be taken when such attempts occur.		
Suggested solution		One object should be responsible to encapsulate the algorithm for managing security policy.		
Forces		a) It is important to have a centralized mechanism to authenticate and authorized users. b) If users make mistakes, they should receive suitable message and be able to correct them. c) In case of many failed attempts to perform an operation, a suitable action should be taken. d) Error checking code makes it difficult to debug/maintain an application.		
Available tactics		a) Make a security check part of Check Point algorithm, e.g. password checks and time-outs to spoofing. b) For distributed systems Check Point can logically divided into authentication and authorization. c) Consider failure actions, repeatability and deferred checks in designing a Check Point component. d) Failure actions should prompt different actions based on the level of severity. e) Include counters to keep track of the frequency of security violations and parameterize the algorithm. f) Create pluggable security components that can be incorporated in different applications. g) Make security algorithm configurable by tuning on and off some options according to requirements.		
Affected Attributes		**Positively**		**Negatively**
		Maintainability, Reliability, Auditability		Some checks needs a secondary interface
Supported general scenarios	S1	If a user makes a non-critical mistake, system returns a warning message.		
	S2	Security policy shall be easily modified over the life of the application.		
	. Sn			
Examples	Ftp server uses *Check Point*, Xauth uses *Check Point* to enable X-windows communicate securely with clients.			

Exploring Quality Attributes Using Architectural Prototyping

Jakob Eyvind Bardram, Henrik Bærbak Christensen, Aino Vonge Corry, Klaus Marius Hansen, and Mads Ingstrup

Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Århus N, Denmark
{bardram, hbc, apaipi, klaus.m.hansen, ingstrup}@daimi.au.dk

Abstract. A central tenet of software architecture design is to base this on a formulation of desired quality attributes, such as buildability, performance, and availability of the target system. Thus there is a need for architectural evaluation—ensuring the architecture's support for desired quality attributes—and a variety of evaluation techniques have been developed, described, and used. Architectural prototyping is an experimental approach that creates executable 'skeleton' systems to investigate architectural qualities of a future system. Architectural prototyping is a learning vehicle for exploring an architectural design space as well as an evaluation technique. The contribution of this paper is to explore the evaluation aspect of architectural prototypes from an analytical standpoint. We present an analysis and discussion of architectural prototyping in the context of two well-established quality frameworks. Our analysis concludes that architectural prototyping is a viable evaluation technique that may evaluate architectural quality attributes and especially valuable in cases where the balance between opposing qualities must be assessed.

1 Introduction

Bardram et al. [1] has defined architectural prototyping and an architectural prototype as:

> An *architectural prototype* consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. *Architectural prototyping* is the process of designing, building, and evaluating architectural prototypes.

They outline a number of characteristics of architectural prototyping: It is a viable and cost-efficient technique for exploring an architectural design space and for addressing issues regarding quality attributes for the target system. Architectural prototypes typically implement only an architectural skeleton without business-oriented and user-oriented behaviour making them cost-efficient to produce. Architectural prototypes often address architectural risks, and, finally, they address issues of knowledge transfer and architectural conformance.

Floyd's seminal paper on prototyping [8] identified several classes of prototypes following which, Bardram et al. makes a similar classification of architectural prototypes as either *exploratory* or *experimental*. Exploratory prototypes are built to explore alternative architectural solutions and clarify requirements while experimental prototypes are used to gauge the adequacy of a proposed architecture, typically by directly measuring or analyzing software architecture qualities.

At Department of Computer Science, University of Aarhus, we are presently exploring and developing the architectural prototyping technique and major research questions that we are working to answer are:

- *Which types of architectural quality attributes (e.g., performance, modifiability, or security) are architectural prototypes helpful in evaluaing?*
- *What characterizes the quality attributes that architectural prototypes may or may not evaluate?*
- *What potential pitfalls must architects be aware of if they use architectural prototyping to design or evaluate a software architecture?*
- *What is the cost/benefit of architectural prototyping in terms of, e.g., spent effort, staff hours, or training?*
- *What characterizes the architectural prototyping process? Is it possible to define concrete steps and guidelines to follow?*

The main contribution of the present paper is to report findings on the first three questions—the other questions are still very much open issues. Our conclusion is that architectural prototypes can indeed by used to assess and evaluate the types of architectural qualities we have analyzed, both those observable at run-time (like performance) as well as those observable only at design-time (like modifiability). There are a number of sources of errors, however, that architects must be aware of. Finally, one of the important aspects of architectural prototypes is that they evaluate quality attributes in concert, not in isolation, and the technique is thus especially valuable in giving a concerted perspective on quality.

The approach taken in the present paper is analytical rather than empirical. Though we present a set of architectural prototypes that we are using to analyze architectural presently, they are described primarily to define context and to serve as illustration in the subsequent discussion of quality frameworks. The empirical data in terms of qualitative and quantitative assessment of the described architectural prototypes will be reported elsewhere.

The main body of the paper revolves around the analysis and discussion of two quality frameworks for software architecture. The analysis and discussion concern whether architectural prototypes may evaluate the qualities defined in the framework. The two frameworks analyzed are:

- **SAIP:** The quality attribute framework defined in the book *Software Architecture in Practice* by Bass et al [2]. This framework is chosen because it is directly aimed at architectural evaluation and because it is an influential book with a practical and industrial perspective but rooted in an academic tradition.

– **ISO:** The ISO-9126 standard [14,15,16,17] is a quality standard for software products, and we thus had to recast it to fit an architectural context. It was chosen because it is well-known and thus a candidate in an industrial setting.

By performing the same analysis with respect to architectural prototypes' ability to evaluate (architectural) quality as defined in both frameworks, our analysis and thus conclusions cover a spectrum of quality frameworks.

The remainder of the paper is organised as follows. Section 2 sets a stage for our discussion of the relation between architectural prototypes and quality attributes by discussing a small set of architectural prototypes and the reasons why they were envisioned, designed, built, and evaluated. The next two sections, 3 and 4, shortly outline the two quality frameworks and embark on a discussion of architectural prototypes' relevance as exploration, evaluation, and measuring technique. Finally, we discuss and conclude in section 5 and 6.

2 Setting the Stage: Empirical Background

Over the last years we have used architectual prototypes for exploration and experimentation in a number of projects with different types of architectures. Some of these have been discussed in detail in [1].

In the discussion of using architectural prototypes to investigate software architecture qualities, we will refer to three architectural prototypes developed as part of the EU-funded Integrated Project "Palpable Computing: PalCom" [24]. The main objective of this project is to design a new software architecture for pervasive computing which should help users understand how such pervasive computing technology works and what happens when they fail—hence the systems built using this architecture should be *'palpable'* for the users.

The architectural prototypes were chosen because they span a range of architecture qualities and they are used in the paper as illustration of on-going work to evaluate architectural qualities.

2.1 Assembly Architectural Prototype

A core challenge in understanding how a set of distributed and networked devices work in a future pervasive computing infrastructure is to understand how the components are connected and how they work together. For example, it may be hard to understand how a future audio-video setup in a smart house is working including its advanced use of, e.g., distributed loudspeakers, TV tuners, display surfaces, and storage media. To help the designer and programmer make understandable applications, we introduced the concept of an 'assembly' in the architecture [13]. An assembly orchestrates the cooperation between distributed components, services, and connectors in the system in a dynamic, contingent way. By providing this concept on the architectural level we were hoping to help the user in understanding the purpose of larger building blocks of the system in terms of special-purpose assemblies that work together. Initial prototyping session with future users involved, supported this claim.

Hence, in order to validate if the assembly and its architectural design was appropriate for the PalCom architecture, we implemented an architectural prototype. This architectural prototype were used for experimentation of the exact design of the 'Assembly' concept and for investigating if this concept would help maintain a conceptual integrity in the overall goal of the PalCom architecture.

2.2 PRE-VM Architectural Prototype

An important quality of the PalCom architecture is *modifiability.* In order to support this quality, a tactic explored in PalCom is to apply the publish/subscribe pattern for distributed communication [6]. In the proposed architecture, all systems are compositions of independent services, each potentially offering their own functionality to users. To make the system modifiable, the services are loosely coupled and in this way it is possible to change the system while it is running. When a service has been removed or inserted, the system's behavior should be able to change without causing as few problems as possible for users. The architecture of the services enables them to discover each other and communicate using publish/subscribe.

This communication is made possible by the implementation of publish/subscribe in the 'PRE-VM' (Palpable Runtime Environment Virtual Machine) on which the prototype services are deployed. The PRE-VM is built partly to evaluate and experiment with modifiability aspects of the future system.

2.3 Contingency Management Architectural Prototype

Another central software quality of the PalCom architecture is *availability*, i.e., how roboust the architecture is to system failure and associated consequences [2]. To support the development of highly available systems in a pervasive and distributed execution environment, the PalCom architecture contains a *Contingency Management* mechanism. Contingency management is basically a proactive distributed exception handling mechanism available for the programmer.

Again, in order to evaluate the design of this particular part of the PalCom architecture, an architectural prototype for contingency management has been developed and embedded in the PRE-VM. In this way we were able to judge the appropiateness of the current design of contingency management in development of applications using the PRE-VM.

3 Software Architecture in Practice

Software Architecture in Practice is an influential book that strikes a balance between the emphasis on terminology, concepts, and characterization and operational and practical techniques—thus a balance between an academic and practical or industrial standpoint [2].

Bass et al. discuss *quality attributes* as yardsticks to measure quality aspects of software architecture. In contrast to the ISO/IEC 9126 standard (see Section 4, the authors do not provide a definition of the quality attribute concept per se, but point out their historic background in various research communities. This background has resulted in the development of diverse vocabolaries: the performance community speaks of "events" arriving at a system, the security community speaks of "attacks", the usability community of "user input", etc. An important contribution is the introduction of *quality attribute scenarios* as a way to define a common vocabulary and to make quality attribute definitions operational.

Quality attribute scenarios identify the context of a quality attribute and state measurable properties of an architecture, thus in essense defining metrics. They are expressed using a standard template having six parts: the ① *source of stimulus*, the ② *stimulus* itself, the ③ *artifact* that is stimulated, the ④ *environment* that expresses the condition under which the stimulation of the artifact happens, the ⑤ *response* that the simulus generates, and finally the ⑥ *response measure* that defines a metric to measure the response. An example of a performance scenario is that ① 10 users ② concurrently initiate transactions on the ③ system ④ under normal operations and as a result the ⑤ transactions are processed ⑥ with average latency of two seconds. An example of a modifiability scenario is that ① developers ② wishes to change the user interface in ③ the code ④ at design time and the ⑤ modifications are made without side effects ⑥ in three hours. Thus quality attribute scenarios provides both a context for a quality attribute as well as a way to concretely measure whether an architecture fulfills the requirements of the scenario.

Bass et al. provide a coarse classification of qualities into three classes: *system*, *business*, and *architectural* qualitys that each classify a set of fundamental quality attributes. Bass et al. present qualities in this order, but the nature of architectural prototypes calls out for discussing architectural quality attributes first.

3.1 Architectural Quality Attributes

Architectural qualities are about the architecture itself though they indirectly affect other qualities—for instance conceptual integrity influence modifiability.

Conceptual integrity is *doing similar things in similar ways*, i.e. that the architecture has a clearly identifiable theme—as opposed to an eroded architecture with, e.g., feature creep. Architectural erosion can often be attributed to lack of knowledge of the overall vision in the development team and lack of knowledge of the architectural principles or patterns that must be employed to add or change behaviour of the system. A major benefit of architectural prototypes is that the architectural blueprint for developers may be working code in addition to more traditional means of documentation such as Unified Modeling Language (UML) diagrams and large bodies of text. Here developers can see and learn the architectural vision in action, and find patterns and principles for required functionaly. As argued by Bardram et al. architectural prototypes are a way of ensuring architectural conformance between architecture *as-designed* and *as-built*. This point is strengthened by the fact that several development processes stress the value of a skeleton system of the architecture that can serve as starting point for the development of the functional aspects, a prominent example being Unified Process [26].

The assembly architectural prototype helped to explore and design the architectural details of how the concept of an assembly should work in the architecture, including its role as a connector and component.

Buildability is a most prominent quality that architectural prototypes serve to analyze. Buildability is the quality of an architecture that it can be built by the available team in a timely manner. Architectural prototypes may be built to learn about the technological platforms (e.g., J2EE), programming models (e.g., knowledge-based system), and other aspects that may be new to the organization or to the architects.

Exploratory architectural prototypes are typically used to assess this quality. Architects that are unclear about the appropriateness of a given technology, a given architectural pattern, or some architectural tactic for the architectural problem at hand, may learn and explore about the approach's buildability by crafting one or more prototypes.

There are aspects of buildability, however, that architectural prototypes do not address. An example is developer team skill set. An architectural prototype may be made by the architect that uses advanced techniques, like design patterns, exception handling, etc, that provides little help to the development team if the underlying principles are unknown to them. On the other hand, an important aspect of the architectural prototype may indeed be a to teach developers these techniques as there is typically very little functional code to blur the picture.

3.2 System Quality Attributes

Qualities of the system in itself is classified in this category.

Modifiability. Modifiability is a measurement of the cost of change. Again, a global requirement of modifiability is senseless, instead artifacts that have a high probability of change must be identified as well as likely changes and the context of each change. Often, modifiability is not a quality attribute that is observable at run-time but must be assessed by analyzing source code, configuration options, and other design time artifacts.

Architectural prototypes are by definition executing systems however architectural prototypes are obviously designed and implemented and the architects will have all the design time artifacts available for modifiability analysis. Indeed, several of the case studies reported by Bardram et al. [1] dealt explicitly with the modifiability aspect.

The PRE-VM architectural prototype sketched in Section 2.2 is being developed primarily to explore modifiability aspects. In this case the modifiablity is two-fold. First and foremost, the modifiability we strive to obtain in the architectural prototype is modifiability for the application programmer working on top of the VM. Second, it is a central goal in PalCom that modifiablity on the design level enables modifiability for the end user. The PRE-VM architectural prototype currently provides us with the necessary modifiablity and thereby give us a very direct way of exploring and experimenting with the modifiability of the PalCom systems.

Performance. Performance has generally and historically been a major focus of computer science.

Architectural prototypes are a supplement to predictions and analyses. Exploratory prototype implementations of a set of potential "architectures-to-come" allows relative judgements of performance characteristics to be identified. From this starting point, experimental architectural prototypes allow concrete measurements to be be made under a range of different situations—situations that will be defined in terms of quality attribute scenarios that identify major performance concerns. As an example the performance scenario outlined earlier form a concrete metric for measuring whether a given prototype will respond within the two second latency limit.

For systems where performance is not deemed the most important quality, architectural prototypes may even be the only tool for measurement, serving as a litmus test that the system has no architectural bottlenecks that result in poor performance.

There are important considerations, however. Architectural prototypes are typically the skeletal architecture without any end-user or business oriented behaviour. Thus if the functional components contain costly processing it may invalidate any conclusion on performance. Of course functional components may be simulated in the architectural prototypes—stubs that simply "burn CPU cycles"—but the conclusions are no better than the estimates made for the simulated workload.

Availability. The availability of a system is the probability that it is operational when it is needed. High availability is largely a matter of detecting and masking/correcting faults and failures to reduce the time where the system is non-operational. Architectural measures to ensure high availability are well known and presented in software engineering handbooks. Bass et al. present many of these in the form of *tactics*. Examples of tactics are 'ping/echo', 'heartbeat', 'active and passive redudancy', and 'shadow operation'.

Architectural prototypes may again serve as tests for concrete availability scenarios in the same vein as argued for performance: once the prototype is operational, direct measurements can be made under the failure situations that are considered important.

The Contingency architectural prototype sketched in Section 2.3 is still in a very first draft, but we have already been able to explore how distributed cooperating services and assemblies can listen to error messages from each other across VMs and use this to adapt their own behavior. This is done using the publish-subscribe communication mechanism in the PRE-VM. This enables an assembly to be notified in case of failure of one of its services. Hence, if for example one

GPS service is failing the assembly can try to discover and switch to another GPS service nearby. This contingency management architectural prototype also helped us explore and experiment with architectural tactics for sustaining availability, like heartbeat and ping/echo mechanisms. Hence, we are able to send 'alive messages across the publish-subscribe protocols.

Just as for performance, the availability quality may be invalidated by "architecturally significant defects" introduced into the business logic that the prototype does not embody. For instance, an architectural prototype with high availability cannot in general guarantee that developers will not introduce deadlocks in the implementation of the business logic.

Testability. Testability is the probability, assuming the software system has at least one fault, that the next test execution will lead to a detected failure. Testability is, like modifiability, a quality attribute that is not observable at run-time and indeed we find that most of the argumentation made for the modifiability quality applies for testability as well. The team of architects that wish to ensure testability through architectural means may use the architectural prototypes as the lab where different techniques and tactics may be explored. Of course, care must be taken, as always, that the techniques settled on carry correctly into the production phase.

Usability. Usability measures among others the ease with which a user may accommmplish a desired task.

In general, *usability* has deeper connections to software architecture than just separation of user interface and domain functionality as in Model-View-Controller (MVC; [20]) or Presentation-Abstraction-Control (PAC; [3]). An example of this it that it may have significant architectural implications to provide a cancel operation in a distributed system [18]. On the other hand, usability is intimately coupled to user interface design which is largely non-architectural in nature.

Again, architectural prototypes are instruments to explore architectural decisions. What is the best way to support undo and redo of operations? How can the architecture reflect upon itself and provide the user with a system model that aids learning and comprehension? The Assembly architectural prototype tried to deal with some of these issues.

3.3 Business Quality Attributes

Bass et al. list a number of quality attributes relating to business: Time to market, Cost/benefit, Projected lifetime of the system, Targeted market, Rollout schedule, and Integration with legacy systems. Obviously, these qualities have a large impact on the system qualities.

Architectural prototypes serve in an indirect sense to estimation and evaluation of these qualities. Architectural prototyping is intimately related to the issue of *risk*. An executing system implementing the architecture on the choosen platform is a major step towards minimizing risk associated in architectural design. Thus, architectural prototypes serve to shed light over issues and risks relating to time to market, rollout schedule, legacy integration, and cost/benefit. Of course, the cost of the architectural prototype itself must be counted.

4 ISO/IEC 9126

The ISO/IEC 9126 standard [14] defines a quality model for software product quality in software engineering. The standard defines quality as “the totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs” [14, p. 20]. In the case of ISO/IEC 9126, the entity is a software product.

The quality model of ISO/IEC 9126 has a number of potential uses ranging from requirements specification over testing objectives to acceptance criteria for a software product. In relation to software architecture, we are generally interested in specifying architecture design criteria and evaluating architecture designs in terms of measurements of quality characteristics.

The standard is, however, generally applicable to software products and thus needs to be mapped or specialized to software architectures if it is to be useful in this context. Thus, after briefly introducing the ISO/IEC 9126 (Section 4.1), we discuss how the ISO/IEC 9126 may be applicable to software architecture (Section 4.2). In doing so, we relate to architectural prototyping in particular.

4.1 Introduction to ISO/IEC 9126

The quality model defined in the standard is a two-part model, the two parts being “internal and external quality” and “quality in use”. Internal quality is concerned with attributes of the software product itself [16], external quality is concerned with the software product’s use under certain conditions [15], and quality in use is concerned with use by specified users in a given use context [17].

For each part of the quality model, characteristics of software quality are given. For internal and external quality, these are *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability*. The characteristics are further introduced and discussed in Section 4.2. For quality in use, the characteristics are *effectiveness*, *productivity*, *safety*, and *satisfaction*.

For each of these characteristics, a number of sub-characteristics are defined. Examples for maintainability include “analysability”, “changeability”, and “stability”. For sub-characteristics, *metrics* are used to measure them. An example of an external changeability metric is “Modification complexity” and an example of an internal changeability metric is “Change recordability”. Metrics may be used to assign values to attributes of (sub-)characteristics through measurements. To measure “Modification complexity”, e.g., a maintainer of the software product who is trying to change it should be observed. The “Modification complexity” is then defined as $T = (\sum \frac{A}{B})/N$, where A is time spent on a modification, B is the size of the modification, and N is the total number of modifications [15, p. 56].

4.2 An Architectural View of ISO/IEC 9126

If we consider software architecture designs in the form of descriptions (e.g., using the UML) as a software product, internal metrics may potentially be applied to the description. If we consider software architecture designs in the form of architectural prototypes, internal metrics may be applied to architectural prototypes seen as source code and external metrics may be applied to architectural prototypes seen as executing systems. In both cases, quality in use is not relevant since there is no actual, direct use involved in neither architectural description nor architectural prototypes. This section is then concerned with external and internal quality for architectural prototypes.

For internal and external quality, we may then look at their characteristics and sub-characteristics and for each of these decide whether they are relevant in an architectural context. (Sub-)characteristics are relevant if they are architecturally significant, i.e., if they affect or are affected by software architecture. An example of a characteristic which is not per se architecturally significant is functionality. This is further discussed below.

For architecturally significant (sub-)characteristics it is then of interest to try to map metrics to software architectures and, in the context of this paper, to architectural prototypes. This is also further discussed below.

Characteristics and Software Architectures. For each of the internal/external quality characteristics, there is a sub-characteristic pertaining to compliance with standards, conventions, regulations, or similar (such as *efficiency compliance*. This may or may not be architectural in nature depending on the actual elements that the software product needs to be compliant with. An example of an architectural prototype that would be used to explore architectural issues in this would be an architectural prototype that investigated redundant hardware architectures as, e.g., required at the higher Safety Integrity Levels of the IEC 61508 standard for functional safety [12,11].

With respect to the *functionality* characteristic of internal and external quality, software architecture is most often defined as being structure rather than function of a software system[2,25]. Thus one would expect the sub-characteristics of functionality in ISO/IEC 9126 to be architecturally insignificant. This is, however, not the case for all: *suitability* (providing appropriate functions for specified tasks) and *accuracy* (providing results with needed precision) are arguably non-architectural whereas *interoperability* and *security* may be seen as architectural (e.g., [2]). Interoperability may, e.g., be explored by building architectural prototypes which provide proof-of-concept integration among systems which are to interoperate.

Reliability has as sub-characteristics *maturity* (avoiding failure as a result of faults), *fault tolerance*, and *recoverability* (re-establishing a specified level of performance in the case of failure). These are clearly architecturally significant and may be enabled by architectural tactics such as introducing redundancy, transactions, and rollbacks. The tactics may be used as the basis for architectural prototypes.

The sub-characteristics of *usability* in ISO/IEC 9126 are *understandability* (enabling the user to understand whether and how a software product can be used for a specific task), *learnability*, *operability*, and *attractiveness*. Here, learnability (supported, e.g., through exploration by undo/redo and cancel) and operability (supported, e.g., through sufficient reliability) affect software architecture, cf. Section 3.2.

Efficiency in terms of *time behaviour* (providing appropriate response times) and *resource utilisation* (using appropriate resources when performing functions) is to a high extent related to software architecture and may also be explored via architectural prototypes.

In ISO/IEC 9126, *maintainability* is concerned with *analysability* (ability for software to be diagnosed, e.g., for failure causes), *changeability*, *stability* (avoiding unexpected effects of modifications), and *testability*. Arguably, analysability is not architectural since analysis of failures need to take its outset in the actual functionality of the software product. Changeability is very much related to the "modifiability" quality attributes of Bass et al. [2] (Section 3.2). The Assembly architectural prototype among other investigates aspects of stability by the use of the assembly construction for coordinating dynamically changing services.

Lastly, *portability* has as sub-characteristics *adaptability* (ability for the software producted to be adapted to a new environment without being changed), *installability*, *co-existence* (ability to co-exist with other, independent software products), and *replaceability*. These are all architecturally significant.

Metrics and Software Architectures. We may distinguish between a number of different uses of metrics in relation to software architectures:

- As a *checklist* for exploratory use of architectural prototypes
- As a basis for *absolute measurements* in experimental use of an architectural prototype
- As a basis for *relative measurements* in comparisons of architectural prototypes

Consider as an example the *available co-existence* metric for the external co-existence subcharacteristic of portability [15, p. 65]. When used as a point in a checklist, the architect may, e.g., want explore how a software product works in a distributed setting with respect to co-existence. An example of this from the PalCom project is in the Assembly architecture prototype in which the use of publish/subscribe for distributed communication was investigated; with this communication paradigm, any number of subscribers may receive a published event, thus potentially causing failures or interference of subscribers.

In ISO/IEC 9126, the method of application of the metric is to "Use evaluated software concurrently with other software that the user often uses" and to count the number of constraints or unexpected failures that the user encounters over a certain period of time. The literal application of the metric is not possible for a prototype which does not implement a significant part of the facilities for user interaction. However, if the end user (and user interaction) is not part of the definition of a metric, an absolute measurement based on this application is relevant. A relative measurement would also be possible, e.g., comparing a publish/subscribe mechanism with a subject/observer mechanism for communication in the Assembly case.

All of the *external metrics* (as in the example above) can in principle be applied to architectural prototypes, but doing so raises issues:

– Are the resulting measurements valid in relation to a resulting product?
– If resulting measurements are valid, can they be produced with reasonable effort?

The first issue is, e.g., problematic in relation to maturity metrics in which residual latent faults are estimated; it is very hard to argue that a result for an architectural prototype should hold for a final system since this is very much dependent on implementation of functionality. The second issue is, e.g., problematic in relation to usability metrics (e.g., *demonstration effectiveness* and *function understandability*) in that they require extensive user test and thus a realistic form of user interaction.

An example of an external metric that may reasonably be used directly on an architectural prototype is the *response time* measure of the time behaviour sub-characteristic of the effiency characteristic.

Considering *internal metrics*, many of these are related to functionality implementation (e.g., the *function inclusiveness* metric for replaceability which counts number of functions in a software system that are similar to those in a replacing software system). Still others are related functional requirements and thus not applicable to software architectures.

5 Discussion

There are several views on the notion of quality [9,19], so it is unsurprising that no consensus on a definition exists. Still, much work have gone into developing suitable frameworks for quality. This is evident in both the SAIP and ISO frameworks we have focused on in this paper. Though incomplete [29] they have been selected because both represent obvious choices in an industrial setting, both are operational as quality yardsticks, and they represent two different approaches: one based on scenarios, and the other on various types of metrics.

Relating architectural prototypes to both the SAIP and the ISO framework, the analysis above convince us that architectural prototypes may evaluate most types of quality attributes or architecturally significant sub-characteristics. As argued above architectural prototypes will be able to serve to analyse, learn about, and assess qualities of an architecture. This does not mean that architectural prototypes are always first choice or should always be built. The point is that architectural prototypes are a viable tool that should be considered along with or instead of other evaluation techniques.

Many existing approaches to achieving quality software are applicable only in the later stages of a software project. This includes source code analysis [4,10], directed testing [22], and code review [7,23]. The use of architectural prototypes for evaluating quality is *preventative*, because it as part of the development process helps make better choices about the architecture. A *curative* approach, as outlined by Dromey [5], focus on testing and use by users to find defects which are subsequently fixed. While there have been some discussion about which is better [21], Dromey argues convincingly that both are needed.

The use of architectural prototypes to evaluate software quality is justified because it is positioned between the techniques that can be applied in the first stages of a development project and those requiring a large part of the system being implemented. Expert judgement [27,28], for instance, is a technique that can be applied in the early stages. The methods by which to conduct it may disclose points of disagreement as to what architecture is preferable. In that case, or if more certainty is required, an architectural prototype provide firmer ground for reasoning about the quality of the system.

The economic viability of architectural prototypes is particularly important in the early stages of development, but also in general since “the actual levels of quality achieved in practice are dictated by quality assessment tools and technologies and willingness to pay for applying them.” [23].

It is important to note that the assessment aspect is somewhat different from most other evaluation techniques. Architectural prototyping is about *building* the architecture, not reasoning about it, formalizing it, or simulating it. In this way architectural prototyping is especially well suited to explore and experiment with architectural tactics [2]. Tactics are the concrete techniques, patterns, and ideas that may enforce the presence of a required quality attribute in the architecture. Architectural prototypes in essense explore the validity of a given tactic (or experiment with a set of them) to actually provide the needed quality.

A major challenge for any architecture is the proper balancing of opposing quality attributes: high modifiability usually costs in terms of performance, high availability is usually done by redundancy which costs in terms of security, etc. Any tactic will enhance one quality but sacrifice others. A principle and intrinsic property of architectural prototypes is that they do not consider any single quality attribute in isolation: If you *do* introduce a given tactic to make some aspect more modifiable, then the architectural prototype *will* execute slower. This point has several consequences. First and foremost, it is important to see as many quality attributes and their interplay in action as possible to gain the overall picture of the quality of the architecture. Second, working with the real substance of the final system carries a bigger chance of discovering interference between qualities than techniques focusing on only one quality: a rate-monotonic analysis of performance, e.g., will not allow the architect to assess implications on modifiability.

For the qualities discussed by the two quality frameworks, it is notable that experimental architectural prototypes may serve as the “system” where measurements are made. As has already been pointed out, such prototypes may rule out architectures that will never perform with respect to some identified set of qualities, but precaution should be made if one conclude that the chosen architecture will perform. The prototype is not the final system and a bad implementation of a business component may, e.g., introduce deadlocks, contain code vulnerable for attacks, starve other processes. This said, however, we are no worse of using architectural prototypes than if we made the same analysis based on analysis artifacts only such as UML models. Indeed, the architectural prototypes may be the vehicle for transferring knowledge to business logic developers about the architectural vision that provide a higher probability of these components not invalidating architectural constraints.

Comparing the quality models of Bass et al. and ISO/IEC 9126 there are major differences: Bass et al. is by construction relevant for software architecture, they differ in attributes/sub-characteristics involved (e.g., availability can be assessed by a combination of assessment of maturity, fault tolerance, and recoverability [14, p. 9]). Used as background for exploratory architectural prototyping both have the same use: with outset in attributes/sub-characteristics, architectural prototypes may be constructed that explores these. Following Bass et al., the construction of an architectural prototype supporting a certain attribute is potentially straightforward: the architectural tactics presented by Bass et al. are all related to one or more attribute.

Looking at experimental architectural prototyping on the other hand, ISO/IEC 9126 appears to be more supportive of this, since quality assessment is directly based on metrics. Bass et al. base their assessment on quality attribute scenarios (Section 3), leading to a need for metrics in order to decide whether a response measure is as specified. It should be noted, though, that using architectural prototypes experimentally in relation to qualities has not been the focus of our work so far and thus not of this paper.

6 Conclusion

In this paper we set out to analyze the use of architectural prototypes—executable skeletal architecture implementations—as direct means to evaluate desired and required qualities of an architecture. We have based this analysis and discussion on our experience with building, analyzing, and evaluating architectural prototypes and have shortly outlined three concrete examples as a stage for our discussion. We have outlined two different quality frameworks, the ISO/IEC 9126 standard and the framework from "Software Architecture in Practice" and discussed architectural qualities from these frameworks with respect to architectural prototypes, both exploratory as well as experimental.

Our main conclusion on this analysis is that architectural prototypes are viable tools to learn about, analyse, and with some care also direct measure an support of an architecture for desired quality attributes. An important characteristics of architectural prototypes that sets them apart from most other evaluation techniques that either formalize or simulate the architecture, is that they are coarse, actual implementations of the architecture. They therefore represent an opportunity to observe conflicting qualities as it is not possible to implement code that only expose a single quality. This stresses architectural prototypes as learning vehicles for the architecture team.

Architectural prototypes has been treated primarily as an evaluation technique within this paper. However many aspects make them valuable tools over and above just their evaluative nature. They are learning vehicles for architects when envisioning and learning new architectural constructs, patterns, and ideas and force them to be grounded in the reality of concrete implementation. They help to avoid overlooking important architectural implications, and thus reduce risks, due to their closeness to implementation.

Our goal is not to propose architectural prototypes as a replacement for other techniques, rather our goal is to present the technique as a valuable assessment tool to complement others and that the technique should become part of any software architect's toolbox.

Acknowledgements

The research presented in this paper has been partly funded by the ISIS Katrinebjerg competency centre, Aarhus, Denmark (`http://www.isis.alexandra.dk`), and the 6th Framework Programme, Information Society Technologies, Disappearing Computer II, project 002057 'PalCom: Palpable Computing – A new perspective on Ambient Computing' (`http://www.ist-palcom.org`).

References

1. J. E. Bardram, H. B. Christensen, and K. M. Hansen. Architectural Prototyping: An Approach for Grounding Architectural Design and Learning. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, pages 15–24, Oslo, Norway, 2004.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Ed.* Addison-Wesley, 2003.
3. J. Coutaz. PAC, an object-oriented model for dialog design. In *Proceedings of IFIP INTERACT'87*, pages 431–436, 1987.
4. R. G. Dromey. Cornering the chimera. *IEEE Software*, 13(1):33–43, 1996.
5. R. G. Dromey. Software quality—prevention versus cure? *Software Quality Journal*, 11:197–210, 2003.
6. P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
7. M. E. Fagan. Design and code inspections to reduce errors in program developement. *IBM Systems Journal*, 15(3), 1976.
8. C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer Verlag, 1984.
9. D. A. Garvin. What does "product quality" really mean? *Sloan Management Review*, 26(1):25–43, 1984.
10. R. G. Gromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, Feb. 1995.
11. K. Hansen, L. Wells, and T. Maier. HAZOP Analysis of UML-Based Software Architecture Descriptions of Safety-Critical Systems. In *Proceedings of NWUML 2004*, 2004.

12. IEC. *Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety-Related Systems*, 1st edition, 1998-2000. International Standard IEC 61508, Parts 1-7.
13. M. Ingstrup and K. M. Hansen. Palpable Assemblies: Dynamic Service Composition in Ubiquitous Computing. To appear in Proceedings of Software Engineering and Knowledge Engingeering (SEKE) 2005, 2005.
14. ISO/IEC. *Software engineering – Product quality – Part 1: Quality model*, 2001. ISO/IEC 9126-1:2001.
15. ISO/IEC. *Software engineering – Product quality – Part 2: External metrics*, 2001. ISO/IEC 9126-2:2001.
16. ISO/IEC. *Software engineering – Product quality – Part 3: Internal metrics*, 2001. ISO/IEC 9126-3:2001.
17. ISO/IEC. *Software engineering – Product quality – Part 3: Quality in use metrics*, 2001. ISO/IEC 9126-3:2001.
18. B. E. John and L. Bass. Usability and software architecture. *Behavior & Information Technology*, 20(5):329–338, 2001.
19. B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, pages 12–21, Jan. 1996.
20. G. Krasner and S. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
21. S. Lauesen and H. Younessi. Is software quality visible in code? *IEEE Software*, pages 69–63, July/August 1998.
22. C. C. Michael and J. Voas. The ability of directed tests to predict software quality. *Annals of Software Engineering*, 4:31–64, 1997.
23. L. Osterweil, L. A. Clarke, R. A. DeMillo, S. I. Feldman, B. McKeeman, E. F. Miller, and J. Salasin. Strategic directions in software quality. *ACM Computing Surveys*, 28(4):738–750, Dec. 1996.
24. 6th Framework Programme, Information Society Technologies, Disappearing Computer II, project 002057 'PalCom: Palpable Computing – A new perspective on Ambient Computing'. `http://www.ist-palcom.org`.
25. D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
26. Rational Software. Rational Unified Process: Best Practices for Software Development Teams. `http://www.rational.com/media/whitepapers/rup_bestpractices.pdf`, 1998.
27. T. Rosqvist, M. Koskela, and H. Harju. Software Quality Evaluation Based on Expert Judgement. *Software Quality Journal*, 11:39–55, 2003.
28. M. Svahnberg and C. Wohlin. An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes. *Empirical Software Engineering*, 10:149–181, 2005.
29. B. Wong and R. Jeffrey. A Framework for Software Quality Evaluation. In *Proceedings of PROFES 2002*, number 2529 in LNCS, pages 103–118. Springer-Verlag, 2002.

On the Estimation of Software Reliability of Component-Based Dependable Distributed Systems

Aleksandar Dimov [1] and Sasikumar Punnekkat [2]

[1] Department of Information Technologies,
Faculty of Mathematics and Informatics, University of Sofia,
5 James Bourchier Blvd, 1164 Sofia, Bulgaria
aldi@fmi.uni-sofia.bg
[2] Department of Computer Science and Electronics,
Mälardalen University, Box 883,
7220, Västerås, Sweden
sasikumar.punnekkat@mdh.se

Abstract. Component based development, which had been successful in enterprise computing, shows promises to be a good development model for automotive systems. This is possible if several dependability concerns of the embedded systems can be properly addressed by the models, frameworks and integration platforms. SaveCCM is a component model for automotive systems developed by employing component based system design. Our ongoing research is related to estimation of software reliability for this model. In this paper, we provide a survey of the state of the art on research techniques for the estimation of system reliability based on component reliabilities and architectures. We further discuss their pros and cons with reference to our architectural model and discuss some practical considerations. Based on this we also present the basics of our approach to reliability modeling of SaveCCM architectures.

1 Introduction

In the realm of enterprise computing, component based software systems (CBSS) had made tremendous inroads in cost reduction and had been successful to a great extent. Despite the advantages in terms of system structuring, ease of verification and cost effectiveness of reuse, still the adoption of component-based technologies for the development of real-time and embedded systems had been significantly slower. Major reasons are that embedded systems are often safety-critical and must satisfy requirements of timeliness, quality-of-service, predictability, and have severely constrained resources (memory, processing power, communication). For smaller-size embedded systems, it is important that a system composed of components can be optimized for speed and memory consumption, which is still missing in most of the current component technologies. The increased use of embedded processors in all domains of human life brings forth the need for high quality products to ensure dependable services.

An important quality factor is dependability, which is characterized by several attributes [1], such as reliability, availability, integrity, safety, confidentiality and maintainability. All these attributes are run-time properties except maintainability. Components have specific individual requirements that can be violated when composed and deployed with other components and techniques are needed to ensure that these requirements do not interfere with each other. This is of prime importance in case of components with hard timing requirements. Many important properties of components in embedded systems, such as timing and performance, depend on characteristics of the underlying hardware platform. Traditionally dependability had been a concern only for developers of mission and safety critical systems, who had the potential to spend huge amounts of money and efforts to guarantee that they meet the demanded ultra reliability requirements. Over the years, the line of separation between critical and non-critical systems has become vague and thinner which makes it all the more relevant to have 'aerospace quality at automotive price' rather than a 'quality at any cost' approach. This instantaneously provide a strong cause for building component technologies specifically suited for embedded systems and for providing methods for quantitative evaluation of their dependability attributes and specifically reliability.

1.1 SaveCCM Component Model

Several efforts are ongoing to define and adapt a suitable component model and framework for different embedded application domains [11,12,23] and SAVE project (Component Based Design of Safety Critical Vehicular Systems), [19] has been addressing the challenges in systematic development of component-based software for safety critical embedded systems, particularly vehicular systems. The goal of SAVE project was to provide:

- Specification and development of a component model, called SAVEComp Component Model (SaveCCM) [8] that can be used for development of resource constrained real-time embedded systems;
- Techniques for analysis and verification of functional correctness, real-time behaviour, and dependability (in particular reliability and safety) of systems built from SaveCCM components;
- Architectural framework that enables separation of software from hardware components;
- Run-time and configuration support, including support for assembling components into systems, run-time monitoring, and evaluation of alternative configurations.

SaveCCM is a simple component model that includes specifications about components, interaction between components, component framework, and composition and execution model. Components are port-based execution blocks with control-flow (pipe & filter) type of interaction. A component includes a "require" interface and a "provide" interface. These interfaces consist of data and triggering ports. Data ports enable reading and writing of data, and triggering ports are used for controlling the activation of components. Ports that combine data and triggering are also specified. SaveCCM includes a special type of components (with the role of a connector) – switches that enable connections between components depending on particular port values, determined at composition or run-time. Execution model is simple, which enables a better analyzability of different properties: When triggered a component reads data from input ports, executes and provides the output to the output ports. Components are mapped to an execution model using algorithms for optimized resource usage.

In the context of SAVE project, we tried to consider the reliability estimation challenge, analyzing some of the recently proposed techniques and formulated some directions, which are outlined in this paper.

This paper is structured as follows: Section 2 presents our motivation. Section 3 makes a survey on component-oriented software reliability models. Section 4 makes a brief analysis of these models with respect to their practical applicability. Section 5 presents our view about the reliability model for SaveCCM and finally section 6 concludes the paper.

2 Motivation

It is known that the quality of CBSS is not just an addition of the qualities of its constituent components. Reliability as a part of system quality does not neglect this statement. In recent years, there appeared a trend to adjust software reliability techniques in order to be able to model and reason about the internal structure of software system, but not to treat it as a single whole [6]. Efforts were aimed to answer how does different components affect system reliability. These models are good start for solid scientific research, as they result from adaptation of well-matured physical systems reliability methods to the area of software engineering. However, some of the approaches are not likely to be applied in a straightforward way into software-industry. This is because they take into account some assumptions that do not always hold for the real software in contrast with academic examples. For instance, in the automotive domain, there exists distribution of components among different electronic control units that are connected by a network, which leads to additional requirements on reliability. One of the goals that we try to accomplish in this work is to analyze different reliability models regarding their practical applicability in distributed embedded systems domain. In this respect, we are interested to answer whether the models address the following issues:

- Ability of the model to express dependence of component failures – this is often ignored in reliability modeling. As proposed in [9], with respect to reliability, there exist two kinds of component dependence – *inter-component and intra-component dependence*. Inter-component dependence occurs when one component is likely to introduce faults when executed in the same environment with another component, but do not do so if they are integrated in different systems. In other words, inter-component dependence is a function of the required context. Intra-component dependence happens when a single component is executed more than once in a use-case or is called by another component inside a loop. Then, when the number of executions tends increases, system reliability tends to zero if using traditional reliability methods, even with high-reliable components.
- Explicit modeling of connectors as first-class entities of CBSS architecture. The classic view of software architecture states that it is a collection of components (computational entities), connectors (communication entities) and configurations

between them [18]. This supposes that in every case of architecture modeling connectors should receive first class status in the system [20].
- Availability of the method in the early phases of software development – this means that the operational and hence the usage profiles for components are unknown. Such situation occurs when system designer still evaluates appropriate components and has not made the final choice. In addition, the designer may probably want to be able to do some sensitivity analysis on how the system reliability will change depending on a particular module.
- Ability to provide complete reliability estimations for the system, both for components and for their compositions. Most of the architecture-oriented models assume that the reliabilities of components constructing the system are known in advance. Indeed, individual component reliability could be obtained by testing and using some traditional techniques, such as fault injection or software reliability growth-models. However, in that case the problem of unknown operational profile may spread doubt on the consistency and accuracy of the reliability estimation with respect to the specific usage context of both the components and the system. It will be better to have means to estimate individual component's reliability with respect to the system we want to develop.

Goseva-Popstojanova and Trivedi [6] present a good survey of different reliability models, however their focus was more on the mathematical formulations, rather than on the application of the reliability models to component technologies. Our survey explores models for architectural description of reliability for CBBS with respect to their ability to solve the above stated issues and focus more on recent models. Our view is that for easier application of software reliability models into the practice of component-based software engineering (CBSE) they should be tuned towards the component models in use. One such model is SaveCCM and in this article, we analyze the applicability of surveyed reliability techniques to SaveCCM.

3 Component-Oriented Software Reliability Models - A Survey

At this section, we make a brief review of component-oriented reliability models. They are presented in ascending order depending on the year of publication. For ease of reference, we have used the name of the first author as the name of the model, even though these models are collective works of several researchers as indicated by the references. We present first a short informal description of the model before introducing its theoretical foundation.

All of the models share the same assumption about independence of failures of different elements[1] in the architecture. It makes possible to compute the reliability of a series of elements by multiplying their particular reliabilities. Models based on Markov chains also assume independence of the current state with respect to previous states. We do not mention these assumptions explicitly in every section.

[1] We suppose the term architectural element denotes both components and connectors into software architecture.

3.1 Cheung Model [2]

This is one of the oldest component oriented software reliability models, which is highly significant as most of the subsequent models are based on it. It presents the architecture of the system as a Markov Chain (i.e. a directed graph), where every state represents a component in the system and the arrows represent probabilities of control transfer from one component to another. The model calculates system reliability, using an extended transition probability matrix (TPM). It takes into account that infinite number of component executions may occur until termination of the application execution, e.g. in case of loops.

First every component is assigned a reliability value R_i, which is the probability of successful execution. Then two states C and F are added, in the initial state-machine, as terminal states, representing correct output and failure respectively. Under the assumption that a component failure leads to system failure the probability of transition from every state i to final state F is $(1\text{-}R_i)$. If the system has n components, then the probability of transition from final state n to the terminal state C is R_n. Let denote the probability of control transfer from component i to component j by p_{ij}. This way the probability of successful execution of i and then transfer of control to j is R_ip_{ij}. Now TPM P is constructed with elements R_ip_{ij} and zeros if there does not exist an arrow between i^{th} and j^{th} states. $P^n(i,l)$ is the probability that starting from i, the program execution finishes successfully in state $l \in \{C,F\}$ at n^{th} step. Let Q be the matrix constructed from P, by deleting the rows and columns, corresponding to C and F. System reliability is the probability of reaching the final state C, from the initial state. Hence, given any infinite (including loops) trace of component executions, the probability of successfully reaching state n from state 1 is: $T=I+Q^1+Q^2+\ldots=W=(I\text{-}Q)^{-1}$ and the reliability of the system is $R = S(1,n)R_n$. It can be shown that:

$$R = R_n(-1)^{n+1}\frac{|M_{n1}|}{|W|} \tag{1}$$

where $|W|$ is the determinant of W, and M_{n1} is the determinant of the remaining matrix, excluding the n^{th} row and the first column of W.

Using (1) it is possible to make analysis on sensitivity of system reliability depending on particular components.

3.2 Krishnamurthy Model [9]

The authors refer their model as Component-Based Reliability Estimation (CBRE). It considers system architecture as a component-call graph, in which states represent constituent components and the arrows – possible transitions between them. It estimates the reliability of a system by considering different sequences of component executions in that graph, called path traces. First reliabilities of single path traces are calculated and then system reliability is calculated as the average of the reliabilities of all considered path traces.

Let denote each path trace by *t*. Then the reliability of the overall CBSS is:

$$R_c = \frac{\sum_{\forall t \in T} R_t}{|T|} \tag{2}$$

The reliability of a single trace R_t is calculated by:

$$R_t = \prod_{\forall m \in M(t)} R_m \tag{3}$$

Where M(t) is the set of components executed when the system is executed against *t* and *T* is the set of all traces used to calculate the reliability estimation of the system.

The paper also discusses the question of the so-called *intra-component dependence* by introducing a positive integer parameter called *Degree of Independence (DOI).* The idea is to "collapse" the number of component executions to DOI, so that total independence is denoted by $DOI=\infty$ and total dependence by $DOI=1$. If a path contains *n* executions of the component, then total dependence is consequently $DOI=n$. Authors also the article discus the effectiveness of the DOI parameter in the analysis section of their paper.

3.3 Wang Model [24]

This model is an extension of Cheung's model and focuses on different architectural styles, comprising the system.

Authors use the computational framework, presented by Cheung and give means for construction of the transition probability matrix for four basic architectural styles. These are batch-sequential/pipeline, parallel/pipe-filters, call-and-return and fault tolerance. Note that this model takes the same assumptions as in Cheung's model.

3.4 Hamlet Model [7]

Authors of this model try to address the issue of unavailability of component's usage profile in early system development phases. To do so they do not assume fixed numeric values for reliability but provide model mappings from particular input profile to reliability parameters. Different input profiles are represented by dividing the input domain of the component to subdomains and assigning a probability for each subdomain to occur. This model does not consider explicitly the architecture of the system. Instead, it calculates the output profile of a component, which actually is the input for the next component and is used to calculate latter reliability.

The model takes into account the following two basic assumptions:

- Actual input profile that will come in use in a real system is unknown during component development.
- It is possible to divide the input domain *D* of the component into *n* disjoint subdomains S_i, $i=1..n$.

Operational profile of each subdomain is denoted P_i (i.e. the probability density $P_i : S_i \rightarrow [0,1]$) and has probability of occurrence h_i respectively. Further, the probability of failure for each subdomain is expressed by:

$$f_i = \sum_{x \in D_f \cap S_i} P_i(x) \tag{4}$$

where D_f is the subset of D on which component fails. The distribution of P_i, within S_i is assumed uniform and hence P_i is equal to $1/|S_i|$. Thus, the following holds for f_i:

$$f_i = \frac{|D_f \cap S_i|}{|S_i|} \tag{5}$$

Actually the real practice is to calculate estimates for f_i by random testing of the component with uniform profile in subdomain S_i. Hence, if N tests are performed without failure, then f_i is roughly below $1/N$. Finally, reliability mapping that takes an input profile vector to reliability value R is given by:

$$R = \sum_{i=1}^{n} h_i (1 - f_i) \tag{6}$$

In order to be able to calculate the reliability of component configurations the model defines the *basic composition construction equation,* which maps the input profile D to the output profile Q of the component. Let Q be again divided into m subdomains U_j. In the interconnection of two components C_1 and C_2, the output profile of C_1 becomes input profile for C_2. Thus, the reliability of C_2, if connected with C_1 can be calculated using (6) if the probabilities (denoted by k_j) for each U_j are known. These probabilities can be calculated by the following equation:

$$k_j = \sum_{i=1}^{m} h_i \frac{|\{z \in S_i \mid c(z) \in U_j\}|}{|S_i|} \tag{7}$$

where c is the function computed by C_1.

The authors also state some rules for calculating system's reliability, based on different configurations of components – sequences of components; conditional composition and indefinite loop.

3.5 Singh Model [3, 21]

This model uses the UML notation to represent software architecture and annotates UML diagrams with data necessary to calculate system reliability. System reliability is calculated with respect to the probability of execution of a particular scenario in the system and the relative involvement of the components in that scenario. The model also takes into account the reliabilities of the connectors between physically separated sites in a distributed system. In order to make it appropriate for early development phases in a software development project, a Bayesian analysis framework is provided.

This work tries to develop a reliability analysis method, applicable at the early design phases of development and integration of the system. This was achieved by annotating three types of UML diagrams (use-case, sequence and deployment diagrams) with reliability data.

Use case diagrams are annotated with two types of parameters:

- Probabilities q_i for occurrence of particular types of users or groups of users;
- Probabilities P_{ij} that user *i* requests the functionality in the j^{th} use case;

Hence the probability of executing use-case *j*, given that there are *m* distinct users, is:

$$P(j) = \sum_{i=1}^{m} q_i P_{ij} \tag{8}$$

Certainly not all of the sequence diagrams within a set, corresponding to the same use-case follow uniform probability distribution. So for the k^{th} sequence diagram referring to j^{th} use-case holds:

$$P(k_j) = P(j) f_j(k) \tag{9}$$

where $f_j(k)$ is the sequence diagram frequency. Sequence diagrams are used to obtain the number of "busy periods" of a component, i.e. the time interval between the start of execution of a component's method and the corresponding "return" event. Let this number for the i^{th} component is denoted by bp_i and $_i$ is the estimate of probability of failure in scenario *j*. This way the probability of failure of i^{th} component in j^{th} scenario is:

$$\theta_{ij} = 1 - (1 - \theta_i)^{bp_{ij}} \tag{10}$$

However, the above formula assumes *regularity*, which means that a component introduces the same failure rate, each time when invoked. In other words, its reliability does not consider the usage profile. One way to overcome this constraint is to replace the unique θ_i with the set of probabilities $\theta_{i,l}$ of failure for every specific method l_k, being executed during the k^{th} busy period:

$$\theta_{ij} = 1 - \prod_{l=l_1}^{l_{bp_{ij}}} (1 - \theta_{i,l}) \tag{11}$$

Deployment diagrams are annotated with reliability data for connectors between components. In fact, based on this a failure probability can be assigned to each interaction of a sequence diagram. Let the probability of failure of communication between components *l* and *m* over the connector *i* is ψ_i. Additionally, let $I_{l,m,j}$ be the number of interactions within sequence diagram *j*. Then the reliability of communication between the components *l* and *m* over connector *i* in case of *j* is:

$$\psi_{lmj} = (1 - \psi_i)^{I_{l,m,j}} \tag{12}$$

Finally, the combination of equations (10) and (12) gives the probability of failure of the whole system (assuming regularity of component failures):

$$\theta_S = 1 - \sum_{j=1}^{K} p_j \left(\prod_{i=1}^{N} (1-\theta_i)^{bp_{ij}} \prod_{l,i} (1-\psi_{lij})^{I_{l,i,j}}\right) \tag{13}$$

where p_j are the probabilities of occurrence of a scenario, given by formula (9).

Based on the above equations a Bayesian algorithm can be applied to estimate the reliability of the system. Let us assume that each $_i$ and $_{lij}$ in equation (13) are random variables, with beta distributions as prior distribution. Beta family of distributions are flexible and can describe a wide range of other distributions, especially in the interval [0,1]. Then an estimate for system probability of failure may be obtained by (13) with a number of simulations. A normalized histogram is built using the simulation data and the Beta distribution is fitted to this histogram, which gives a priori estimations for $_i$ and $_{lij}$. Using the results of the prior distribution of the system's failure probability, posterior reliability estimates may be obtained.

3.6 Reussner Model [17]

The model of Reussner et al uses Markov chains to model system architecture. This method uses the idea to express component reliability not as an absolute value but as function of the input profile of the component. Markov chains are constructed in hierarchical manner and states include calls to component services in addition to usual component executions. Services may invoke different methods, which may be either internal or external for the component. The reliability of the component is calculated by the reliabilities of the methods, which it uses, and they depend on the operational profile.

This method addresses the problems with missing data for the operational profile and context environment for the components[2] in the early development stages. Parameterized contracts [10] are used, which makes possible the representation of component reliabilities as functions of the input profile, instead as absolute values.

The model also takes into account that *software components represent closed complex subsystems and open interoperating parts of a larger subsystem at the same time*. Therefore, component's reliability should be calculated in correspondence to dependence with other components. Smaller parts of the components are denoted as methods and services. Every service may call one or more methods to accomplish its task. The estimation of the reliability of a method within a component depends on the reliabilities of the other methods it calls and is calculated by the following equation:

$$r_m = r_{cr} r_{body} r_e \tag{14}$$

where r_{cr} is the reliability of the connection between methods (including hardware reliability), r_{body} is the reliability of the body of the method and r_e is the reliability of the external methods called.

Markov chain is used to model the connection between usage profiles and external component reliabilities. Usage profiles are presented by probability value $u_{i,m}$ for *call to a provided service (method) m* when in state *i*. The latter represent method execu-

[2] Authors denote component as *ken* in their work.

tions in the final implementation. Every method is assigned a preliminary known reliability r_m. Hence, the probability for transition from state i to state j is given by:

$$s_{ij} = \sum_{k=1}^{l} u_{i,m_k} r_{m_k} \tag{15}$$

where l is the number of possible transitions between states. The probability of successful transition between the initial (I) and the final (J) state may be found in a way similar to Cheung [2]:

$$r_{system} = (I - S)^{-1}{}_{(I,J)} \tag{16}$$

For systems with multiple final states, a new *super final state* is introduced that has a reliability of 1.0 and is reachable from any of the original final states.

Further, component reliability is calculated from the reliabilities of its methods and the context-specific usage profile after service reliabilities have been calculated. Reliability of composite components may be predicted similar to the reliability of basic ones. The only difference is that the reliability r_s of one of its services s is:

$$r_s = r_m r_{inner} \tag{17}$$

where r_m is the mapping reliability between the similar ports[3] of the assembly and the inner component, and s is mapped to its service s_{inner}. This equation may be applied recursively for nested composite components.

3.7 Gokhale Model [5]

This model uses time-dependent failure intensity to model failure behaviour of components. It assumes that failures follow the Enhanced Non-Homogeneous Poisson Process (ENHPP) model, which relates the failure intensity to the expected number of faults initially present in the component and the test coverage of component code. Further, it presents the system architecture as Markov chain, where states represent executions of components and arrows – control flow between them. Authors present an experimental framework for estimation of the coverage function, initial number of faults and transition probabilities between components.

Let the system has a total number of n states and m of them to be the absorbing (terminal) states. Then the expected number of executions of component j, denoted by V_j is given by:

$$V_j = q_j + \sum_{k=1}^{n-m} p_{kj} V_k \tag{18}$$

where q_j is the probability that the application starts in module j. This model assumes that the execution time t_j of a component into the application is constant and a priori known. Then the overall time during which the application executes in component j is $\gamma_j = V_j t_j$.

[3] Component ports are referred as gates by Reussner et al.

Failure behaviour of components is supposed to follow the ENHPP[4], which according to the authors is a generalization of the Poisson process models family. This way a time-dependent failure intensity function is $\lambda(t)=ac'(t)$, where a is the total initial number of faults, expected to be present in the component and $c'(t)$ is the first derivative of the so-called *test coverage*. Test coverage is a time-dependent function and is defined as *the ratio of the number of potential fault-sites sensitized by the test divided by the total number of potential fault-sites under consideration* [4].

Reliability of component j is given by:

$$R_j \approx \exp(-\int_0^{\gamma_j} \lambda_j(\theta)d\theta) = \exp(-a_j c_j(\gamma_j)) \tag{19}$$

Then the reliability of the whole component-based system, assuming independent component failures is given by:

$$R \approx \prod_{i=1}^{n-m} \exp(-a_i c_i(\gamma_i)) \tag{20}$$

3.8 Roshandel Model [15, 16]

This model presents an approach for the estimation of individual components reliability when both the implementation and the data for the operational profile are unknown. Component architecture is modeled with a variation of Markov chain, called Hidden Markov Model (HMM) [13]. Transition probabilities of HMM are estimated by an iterative algorithm, which upon convergence provides optimal component reliabilities.

Modeling process is divided into three steps. First is to analyze the architectural specification in respect to a representation of the component semantics and syntax, called the *Quartet*. This is a result from the authors' previous work [14] and models four views of component architecture: interfaces, static and dynamic behaviour and interaction protocols. Information from the dynamic behavior view of a component, which is modeled in a state-based notation, is used in the second step to construct the Hidden Markov Model. It is an extension of the classical Markov model and presents uncertainty of a transition between states, i.e. the unknown usage profile. Uncertainty appears when component's state may remain unchanged during some time interval or when an event, requesting actions for service invocation is issued, but the transition to another state does not occur (erroneous behaviour). Moreover, a single event may lead to different transitions in the HMM, i.e., one event may lead to different states, depending on the context of a program. To represent uncertainty in transition between states, authors use the notions of events and (re)actions. Different events may trigger different (re)actions leading to different component states. The probability for transition between states S_i and S_j via any of the possible pairs of events E_m and actions F_k, given P_{iE_m} (probability of observation of the event E_m) and P_{iF_k} (probability of reaction F_k)is:

$$T_{ij} = \sum_{m=1}^{M}\sum_{k=1}^{K} P_{iE_m} P_{iF_k} = \sum_{m=1}^{M}\sum_{k=1}^{K} P_{ijE_mF_k} \tag{21}$$

For each state, the following equation holds:

$$\sum_{m=1}^{M}\sum_{k=1}^{K}\sum_{j=1}^{N} P_{ijE_mF_k} = 1 \tag{22}$$

Further assuming that the usage profile is unknown initial (prior) values for the transition probabilities are assigned. Then the actual values are estimated by simulation, using the Baum-Welch algorithm [13], given a sequence of interface invocations. This is an iterative process that recalculates component reliability at every step. The final convergence of the estimations is optimized regarding maximization of the component reliability.

The third step is to calculate the overall component's reliability, based on the well-known Cheung's model [2].

Possible option to compute system's reliability is to build an augmented Markov model using series of concurrent component state machines. It may also include event/action pairs, in order to model connectors between components.

4 Analysis of the Models

In this section, we provide an analysis of the models, namely their advantages and disadvantages, in relation to the issues elaborated in section 2.

Cheung model do not consider component dependencies. It also assumes that reliability and architecture data are available in advance. This makes it inappropriate for early development phases and inapplicable without modifications. However, this model is a foundation for the majority of the subsequent models. As an advantage of this model, which is also an advantage of all models based on Markov chains, we may emphasize the ability of straightforward inclusion of connector reliabilities. One disadvantage all models that model software architecture with classical Markov Chains is that they assume independence of the current state of the system from it previous states.

Krishnamurthy model uses existing set of test cases, which may be the system test case or a regression test suite and does not rely on operational profile or randomly generated test inputs to obtain reliability of a software system. This is advantageous for the speed of computation of reliability values, but makes the model little applicable to early software development phases. To overcome this, the authors propose a *risk parameter*, i.e. credibility of the reliability estimation. However, they do not provide means for obtaining the risk parameter. This model attempts to model intra-component dependence, but this technique is not formally proved correct. It is not clear, neither if it is appropriate to collapse multiple executions into less executions, nor how to determine the suitable value of *DOI*. The authors rely on an experimental analysis to evaluate dependence models with different *DOIs*. Another weakness of the *DOI* parameter is that if its optimal value is not an integer number, then the relative error of the reliability estimation will appear unacceptably high. Nevertheless experimental analysis provided in the paper shows that this way of modeling dependence between components can raise the accuracy of the system's reliability estimation.

Wang model is oriented towards the analysis of large-scale CBSS. The idea of modeling system architecture with respect to constituent architectural styles is useful.

However, this is too complex with respect to SaveCCM, because it uses only pipe-and-filter style. In this case, Wang model is actually reduced to the classical Cheung model and inherits all the disadvantages mentioned above. An interesting issue here is that modeling of the call-and-return style may appear useful for expressing inter-component dependence.

Hamlet model do not express reliabilities as exact values but as dependencies from component input profile, which is the main advantage of this model. The need for component function to calculate system reliability renders the method difficult to apply in the early development phases. Even this model does not concern software architecture explicitly, it is appropriate for pipe-and filter architectural styles, which is the case with SaveCCM. One disadvantage here is that, in case of components supplied from different vendors, the assumption for disjoint input subdomains may not hold, which makes the application of the model difficult. Another problem is that obtaining component reliability by testing with uniform input profile may lead to some incorrect results, which again brings up the question for assessment of the reliability parameter with credibility value.

Singh model provides means for estimation of system reliability early in the design development phases when neither component reliabilities nor the usage profile are known and this is the main advantage here. Authors also express connector reliabilities explicitly. These two features make the model interesting for experimentation with SaveCCM. The model does not consider component dependence, but our view is that if not assuming regularity it may become possible to model at least inter-component dependence.

Reussner model is capable of taking into account component dependencies. It is based on a Markov chain, which means that it is possible to include connector reliabilities, but also assumes independence between calls of services or executions of component's methods. A big advantage here is the usage of parameterized contracts. In this way, reliabilities depend on the real context and usage profile of the component and are not fixed values, which makes this method applicable at the early design development phases. However, the model does not take into account the problem of obtaining individual reliabilities of the basic methods.

Gokhale model takes a slightly different approach than other models. It calculates system reliability, based on the time-dependent failure intensity of constituent components. This makes possible the modeling of intra-component dependence. Estimations of failure rate are obtained assuming the ENHPP process and using a test coverage function. An advantage is that authors provide an experimental framework for estimation of all necessary data to calculate system reliability. A disadvantage of their approach is that it assumes existence of data about the internal structure of the component. A possible option, mentioned by the authors is to observe events at component interfaces to obtain coverage data, but this is not really discussed. This issue makes the model hardly applicable for early development phases.

One of the recent works in the area, viz., **Roshandel model** is not yet well documented and seems incomplete. Nevertheless, it proposes a promising and simple approach for estimation of system reliability when the operational profile is unknown, which makes it applicable early in the development process. Originally, the model provides means for individual components reliability estimation, based on specification, but this technique should appear meaningful for component-based systems as well. A disadvantage of the model, which needs to be addressed is that is does not consider component dependencies.

Authors claim that their model addresses the following issues that constrain other similar models, as they do not assume them:

- Ignorance of the effect of component internal behaviour on its reliability
- Availability of a running system to obtain probability data for component service invocations
- Correspondence of each state of the underlining Markov model to an observable event

One disadvantage that is common for all models except for the Wang model is that they do not provide discussion if there are specifics in modeling of fault-tolerant systems. In this way, properties such as diversity and redundancy seem to remain unaddressed. We believe that this is very important in the case of distributed embedded systems.

5 An Approach to SaveCCM Reliability Model

In this section, we present the current directions of our research on arriving at a suitable reliability model for SaveCCM based systems. We should aim to address the following features:

- Description of component reliability
- Description of connection reliabilities
- Description of assembly reliabilities

Reliability is being regarded as a component attribute in the terms of SaveCCM syntax. We denote it as C_r and it may be expressed as a tuple $CR=<R^C,,C^C>$, where R^C is the reliability in terms of the classical definitions – *this is the probability of successful execution of the module, given some specified time period.* The parameter C^C in the reliability expression is the credibility value in SaveCCM, i.e. the risk parameter, associated with R^C.

Reliability R^C may be decomposed as $R^C = R^i R^e R^o$ (similar to [17]), where R^i is the probability of successful reading of significant and consistent data from the input port, R^e is the probability of correct execution of the actual function provided by the component and R^o is the probability for successful writing of the data to the output ports. As SaveCCM uses the pipe-and-filter architectural style, this decomposition of R^C is a promising approach to model component dependence.

Currently none of the known reliability models assumes explicitly the presence of the risk parameter in the formal expression of system reliability. We argue that it will be valuable to introduce means for assessing the reliability of the overall software system based on the credibility of the constituent components reliabilities and our ongoing research is focused on that.

Connection reliability is the probability of successful operation of the connection between the following artifacts: component-component and/or assembly-component. This parameter is worthy at least in the case of remote connection over a network.

Switches represent dynamic behaviour in SaveCCM and we plan to incorporate this dynamism into our reliability model. Moreover, switches are not assigned attribute values in their syntax specification in contrast with the case of components. Probability of successful execution of the switch should be expressed by the connection reliability even in the case of internal (i.e. not distributed) connections. Naturally, it will be unity, if there exist only a binding from the output of one component to the input port of the other, i.e. without switches or data passing over a network.

Reliability *AR* of assemblies are expressed in an analogous way to the reliability of components, e.g. $AR = <R^A, C^A>$, where the parameters R^A and C^A have similar meaning as for components. However, description of assembly reliability attributes should receive special attention, as assemblies have different semantic meaning from components. In SaveCCM, components have finite, but possibly variable execution time. Moreover, we assign weight coefficients on component failure probabilities, depending on the actual importance of the critical components for the reliability of the assembly.

Additionally, our framework plans to include reliability trade-off analysis to enable selection from given component and architectural choices.

6 Conclusions

Industry trends point to increased need for adoption of component-based software technologies to embedded systems domain. Although this is seen as a key to success in achieving time-to-market, other issues such as increasing safety concerns and certification requirements will essentially call for making the component based approach more dependable and predictable. What are needed are sound techniques to estimate and evaluate CBSS reliability in the distributed embedded systems domain.

The main contributions of this paper are the presentation of the current state of the art in architecture-based modeling for CBSS and the analysis of the models with respect to their applicability in the domain of distributed embedded systems. In addition, we briefly present our ongoing research on estimation of software reliability of SaveCCM based systems. Though our focus is on SaveCCM, due to the generic nature of the issues addressed, we envisage that our model to be applicable other domains of distributed systems as well.

Our analysis shows that for adoption of the surveyed reliability models to a specific component model such as SaveCCM, they need some further refinement. Finally, we plan to extend and combine their advantages. Main point should be to preserve the applicability in the early development phases, when exact data of the operational profile or component reliability data is missing. This data should be refined later with the usage of a common framework as and when finer details of component reliabilities are available.

Possible directions for further work are to include into the formal model also a prediction of the effect of software aging on assembly reliability. As recognized in [22] this is a wide-open research topic. Of particular interest are such general questions as:

- How to estimate individual component reliabilities – more specifically - the values R^i, R^e, and R^o

- How to estimate the value of the credibility parameter. There already exists some works in this area, but this question need extensive research in terms of the SaveCCM model
- Modeling of component dependencies inside the assembly

Acknowledgements

The research related to this work is partly funded by The Swedish Foundation for strategic research (SSF) under SAVE Project. The authors are also grateful to Ivica Crnkovic, Sylvia Ilieva, and the anonymous referees for the support and useful comments on earlier version.

References

1. Avižienis, A., Laprie, J-C., Randell, B.: Basic concepts and Taxonomy of dependable and secure computing, IEEE Trans on Dependable and Secure computing, Vol. 1, Issue 1, Jan - March 2004
2. Cheung, R. C.: A user-oriented software reliability model, IEEE Transactions on Software Engineering, 6(2): 118-125, 1980
3. Cortellessa, V., Singh, H., Cukic, B.: Early reliability assessment of UML based software models, In Proc. of Third International Workshop on Software and Performance (WOSP), Rome, Italy, pp. 302-309, July 24-26, 2002
4. Gokhale, S., Philip, T., Marinos, P., Trivedi, K.S.: Unification of finite failure non-homogeneous Poisson process models through test coverage, In Proc. of Seventh International Symposium on Software Reliability Engineering, pp. 299-307, 1996.
5. Gokhale, S.,Wong, W., Horgan, J., Trivedi, K.S.: An analytical approach to architecture-based software performance and reliability prediction, Performance Evaluation, Vol. 58(4), pp. 391-412, 2004
6. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture Based Approach to Reliability Assessment of Software Systems, Performance Evaluation, Vol.45/2-3, June 2001
7. Hamlet, D., Mason, D., Woit, D.: Theory of Software Reliability Based on Components, In Proc. of International Conference on Software Engineering (ICSE), Toronto, Canada, pp. 361-370, May, 2001.
8. Hansson, H., Åkerholm, M., Crnkovic, I., Törngren, M.: SaveCCM – A Component Model for Safety-Critical Real-time Systems, In Proc. of Euromicro Conference, Special Session Component Models for Dependable Systems, IEEE, Rennes, France, September, 2004
9. Krishnamurthy, S., Mathur, A.: On the Estimation of Reliability of a Software System Using Reliability of its Components, In Proc. of the 8th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp.146 – 155, Nov. 1997
10. Krämer, B., Reussner, R., Schmidt, H.-W.: Predicting Properties of Component Based Software Architectures through Parameterized Contracts, In M. Wirsing, S. Balsamo, editors, Radical Innovations of Software and Systems Engineering in the Future, TR CS 2002-10, Universitá Cá Foscari di Venezia, 2002
11. Müller, P., Stich, C., Zeidler, C.: Components@Work: Component Technology for Embedded Systems, In Proc of 27th International Workshop on Component-Based Software Engineering, EUROMICRO 2001

12. Ommering, R., Linden, F., Kramer, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer, 33(3):78–85, March, 2000
13. Rabiner, L.: A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, In Proceedings of the IEEE, Vol. 77(2), pp 257-286, 1989
14. Roshandel, R., Medvidovic, N.: Modeling Multiple Aspects of Software Components, In Proc. of workshop on Specification and Verification of Component-Based Systems, ESEC-FSE03, Helsinki, Finland, September 2003
15. Roshandel, R., Medvidovic, N.: Toward Architecture-based Reliability Estimation, In Proc. of the Workshop on Architecting Dependable Systems, International Conference on Software Engineering (ICSE 26), Edinburgh, UK, May 2004
16. Roshandel, R: Calculating Architectural Reliability via Modelling and Analysis, In Proc. of the Doctoral Symposium at the International Conference on Software Engineering (ICSE26), Edinburgh, UK, May 2004.
17. Reussner, R., Schmidt, H., Poernomo, I.: Reliability prediction, for Component-based Software Architectures, In Journal of Systems and Software, 66(3), Elsevier Science Inc, 2003
18. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17(4), 1992
19. SAVE Project, http://www.mrtc.mdh.se/SAVE/
20. Shaw, M.: Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status, In: Proceedings of the Workshop on Studies of Software Design, May 1993.
21. Singh, H., Cortellessa, V., Cukic, B., Gunel, E., Bharadwaj, V.: A Bayesian approach to reliability prediction and assessment of component based systems. In Proc. of 12th International Symposium on Software Reliability Engineering (ISSRE'01), 2001
22. Stafford, J., McGregor, J.: Issues in Predicting the Reliability of Composed Components, In Proceedings of 5th workshop on component based software engineering, 2002
23. Wallnau, K.: A Technology for Predictable Assembly from Certifiable Components, Technical report CMU/SEI-2003-TR-009, 2003
24. Wang, W., Wu , Y., Chen, M.: An Architecture-Based Software Reliability Model, In Proc. of Pacific Rim International Symposium on Dependable Computing, 1999

Empirical Evaluation of Model-Based Performance Prediction Methods in Software Development

Heiko Koziolek and Viktoria Firus

Graduate School Trustsoft[⋆],
Software Engineering Group,
University of Oldenburg, Germany
(Fax ++49 441 798 2196)
{heiko.koziolek, viktoria.firus}@informatik.uni-oldenburg.de

Abstract. Predicting the performance of software architectures during early design stages is an active field of research in software engineering. It is expected that accurate predictions minimize the risk of performance problems in software systems by a great extent. This would improve quality and save development time and costs of subsequent code fixings. Although a lot of different methods have been proposed, none of them have gained widespread application in practice. In this paper we describe the evaluation and comparison of three approaches for early performance predictions (Software Performance Engineering (SPE), Capacity Planning (CP) and umlPSI). We conducted an experiment with 31 computer science students. Our results show that SPE and CP are suited for supporting performance design decisions in our scenario. CP is also able to validate performance goals as stated in requirement documents under certain conditions. We found that SPE and CP are matured, yet lack the proper tool support that would ease their application in practice.

1 Introduction

One of the most important non-functional quality characteristics of a software system is its performance and related to that its scalability. In a software engineering context, performance is defined as the timing behaviour of a system expressed through quality attributes like response time, throughput or reaction time.

To achieve high performance, an information system not only has to rely on sufficient hardware (like fast processors, large memory and fast network connections), it also has to be implemented with efficient software. On a low abstraction level the program code must be optimized using fast algorithms, which utilise the hardware effectively. On a higher abstraction level the software architecture has to be designed carefully to avoid performance-limiting bottlenecks.

However a thorough architectural evaluation of performance properties is still underestimated and often neglected in the software industry today. Developers usually follow a ”fix-it-later”-approach [14] when it comes to performance and shift the performance analysis to late development stages. This approach ignores the fact that a large

⋆ This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

number of performance problems are simply the result of a badly designed software architecture. If such problems occur in an implemented system, most often it will not be sufficient to fix small code areas. Instead, the whole architecture has to be redesigned, which might be very expensive if possible at all.

Countering the ”fix-it-later”-approach, the performance prediction for software architectures is an active research area today. Over the course of the last decade a large number of methods has been developed to analyse the performance during early development cycles of a software system [2]. Most of the methods try to exploit design documents of software systems (e.g. described in the UML), let the performance analyst add performance relevant information (like expected calculation times, number of users) and automatically construct performance models. These models are able to produce performance predictions without the need for an actual implementation of the architecture.

The goal of the methods is not to predict exact response times and throughput figures of an architecture, which is hardly possible during early design stages when lots of details are still unknown. Instead, performance goals negotiated with the customers of a system, which are stated in requirements documents, shall be validated. Such goals can be maximum response times for specific use cases or minimal throughput numbers for the overall system. The methods only have to ensure that these upper or lower bounds are met, they do not have to predict precise response times and throughput numbers. Furthermore the methods shall support design decisions. If developers identify several different design alternatives with equal functionality during the design phase, they can apply the methods on these alternatives to determine the differences between them regarding the performance. The exact absolute values for each alternative are not needed to support the decision for one alternative, only the relative differences of the alternatives are relevant and can be predicted by the methods.

None of the performance prediction methods has gained widespread application in the software industry. Most of the methods have not been tested on large software systems but only on small examples often created by the authors of the methods themselves. So their practical usefulness is still unknown.

On this account we conducted an empirical experiment, in which we analysed and compared three performance prediction methods for software architectures. Similar work has been done by Balsamo et. al. [3], who compare an analytical and a simulation-based method in a case study. Yet they do not compare the prediction of the methods with measurements from an implementation as we have done in our experiment. Such a comparison of prediction and measurement has been done by Gorton et. al. [6] for a software architecture based on Enterprise Java Beans, but only with a single method.

We selected three of the more matured performance prediction methods and let 31 computer science students apply the methods on the design documents of an experimental web server. The students had no code to analyse, but were given five design alternatives from which they should favour one to implement. After the predictions had been completed we implemented the web server, measured its actual performance and compared predictions and measurements.

With this experiment we tried to answer the following questions:

– How precisely can quantitative performance characteristics be predicted with the methods?
– Do the methods really support the decision for the design alternative with the fastest implementation?
– How does the application of the methods compare to an unmethodical approach to performance prediction?
– What are the properties of the methods and where is room for improvement?

While the first three questions are suited to create and test hypothesises, the last question has a more explorative character. Because we could not gather enough data points for a hypothesis test, the results are shown as descriptive statistics.

The contribution of this paper is an evaluation and comparison of three different performance prediction methods within an empirical study. The usefulness and the deficits of these methods are analysed for the purpose of creating a tighter integration of performance analysis and software engineering. Additionally, a generic method for the evaluation of performance prediction methods has been developed and can possibly be applied to other methods.

The paper is organised as follows. In the next section we shortly describe the three methods under analysis and reason why we have selected them. Section 3 explains our methodical approach and how we conducted the experiment with the computer science students. The results and the answers to our research questions are presented in section 4, while section 5 discusses the validity and limitations of our experiment.

2 Analysed Methods

The following three performance prediction methods were selected for their maturity, their use of standard notations (like UML and Queueing Networks) and the availability of software tools. The Software Performance Engineering (SPE) method has been developed by Smith and Williams and recently extended to include UML-models [14]. Capacity Planning (CP) as described by Menasce and Almeida [11,10] is an approach for the proper sizing of information systems and can also be applied during early development cycles. To include a simulation-based method we added one recent approach by Marzolla and Balsamo [8]. They have developed a tool called UML Performance Simulator (umlPSI), which allows the simulation of performance annotated UML models.

SPE includes three steps for performance modelling and analysis. First, the software architect models the system based on the requirements with UML. In the second step performance-critical scenarios of the UML-model are selected and a so-called software execution model is created for each scenario. These models include the control flow for one scenario and are annotated with the scenario's performance characteristics (e.g. expected duration of a step, number of loop iterations, probabilities for alternative ways through the control flow etc.). In a third step the so-called system execution model is created, which contains information about the underlying hardware resources (e.g. processors, network connections, etc.). In the software tool SPE-ED [13] the developer can define the request arrival rate for one scenario and let the tool analyse the scenario. For the system execution model the tool creates queueing networks and connects them with the software execution model. The calculated performance attributes for a scenario are response times, throughput and resource utilisation and are reinserted into the control diagrams of the software execution model.

CP is usually used for the performance analysis and prediction for already implemented systems. This method is based to a large extent on the queueing network theory. A developer starts with an informal description of the system's architecture, and tries to collect as much performance information as possible through system monitoring and testing. Afterwards a queueing network for the system is constructed and its input values are determined with the performance information collected before. With various software tools [9] the queueing network can be solved and the results may be validated. By changing the input parameters, predictions can be made for the system, e.g. when the number of users increases or other hardware resources are used. Capacity Planning yields response times, throughput, resource utilisation and queue length.

Performance-annotated UML diagrams in the XMI file format are the input for the **umlPSI** tool. The annotations are made according to the UML Profile for Schedulability, Performance and Time [12], and can be entered in regular UML modelling tools. They consist of arrival rates of requests, duration times for activities, the speed of hardware resources etc. umlPSI automatically generates an event-based simulation from these UML-diagrams and executes it. The results of the simulation, like response times for scenarios or utilisation of certain hardware devices, are reported back into special result attributes of the UML profile contained in the XMI-files. The developer may then re-import the diagrams into his UML-tool and directly identify performance problems within his architecture.

3 Conduction of the Empirical Evaluation

3.1 Methodical Approach and Participants

To direct our evaluation we applied the Goal-Question-Metric approach [4]. First we defined our goal (the evaluation of performance prediction methods) and then we raised questions (see introduction) to reach this goal. We defined metrics for the answers to our questions, which will be explained in section four.

To collect data for our metrics we favoured the conduction of a controlled experiment, which results are most reliable of all empirical methods. Other methods are case studies, field studies, meta studies or surveys [17], but their expected results are usually not as reliable as in an controlled experiment. In a controlled experiment a researcher tries to control every factor that might influence the result, except for the factor he wants to study. In our study the only varying factor is the performance prediction method itself. This means that all other data like the qualification of the testers, the input for the methods, the hardware/software environment etc. have to be constant when applying each method. To control the qualification of the testers we decided to let a group of computer science students apply the methods, such that the results of our study are more representative for all developers and not bound to some individual qualification. We also tried to keep the inputs constant for every method and controlled the environment in which the students did their predictions.

The participants were 31 computer science students, who had all passed their intermediate diploma and had at least two years of experience in the field of computer science. All of them had participated in a software project lasting one semester and were not new to software engineering and UML models. The number of participants was too low to perform hypothesis testing, because the amount of data points was too low for certain statistical methods. Nevertheless, by having multiple students apply the methods, we were able to reduce distortions to results due to exceptional individual performances.

The actual study consisted of three steps. First we trained the students in the methods, then we conducted the experiment, and finally we implemented the web server and measured its performance.

3.2 Training Session

Each method was introduced in a two-hour session to 24 students with help of the respective literature. Seven students did not participate in the training sessions and formed an untrained control group. After each session, a paper exercise was given to the students, who had to return the solutions within one week. By solving the paper exercise, the students had to experiment with the methods and tools on their own and were prepared for the later experiment.

From this pretest we could also retrieve some information on how to design the tasks of the later experiment. After the sessions the 24 trained students were separated into three groups. The participants of each group had to apply one of the methods. The disposition of the students into groups was based on the results of the paper exercises making it possible to have three homogeneous groups with a comparable qualification range.

3.3 Conduction of the Experiment

For the experiment we prepared UML-diagrams (component, sequence and deployment) of an experimental web server, which has been developed in our research group. This multi-threaded server is written in C#, can be accessed with the HTTP-protocol, and is also able to generate HTML-pages from the contents of a database.

To simulate the typical application of the methods, we also provided the students with five different, performance-optimizing design alternatives. Out of these alternatives the one with the best performance should be selected. The alternatives consisted of

- Alternative 1a: a cache for static HTML-pages
- Alternative 1b: a cache for dynamically generated HTML-pages
- Alternative 2: a single-threaded version of the server
- Alternative 3: application of a compression-algorithm before sending HTML-pages
- Alternative 4: clustering of the server on two independent computers

Further Details can be found in [7].

A common usage profile (arrival rate of requests, file sizes of retrieved documents, distributions of the type of requests) was provided to the students. Additionally, a hardware environment was specified including the bandwidth of the network connections, the response time for database accesses and the speed for certain calculations. The students were asked to predict the response time for a given scenario for each design alternative. The scenario was modelled as an UML sequence diagram, showing the control flow through the components of the web server when the user requested a HTML-page that was generated with data from a database connected with the web server. In addition to the end-to-end response time as perceived by the user the students should also predict the utilisation of the involved resources like processors and network connections. Based on these results the students made their recommendation for one of the alternatives.

For the CP-method a workload trace from a prototype of the web server was handed out to the students, so that they were able to perform the necessary calculations. However, none of the groups was provided with any code of the server, preventing the possibility of running tests during the experiment.

The experiment was carried out in a computer lab at the university. Each student performed his/her predictions within two hours. It was assured that each student applied the methods single-handedly and did not influence other students.

3.4 Implementation and Measurement

After the experiment the web server was implemented for each of the five design alternatives. The hardware environment of the experiment's task was reproduced and the proposed usage profile was simulated via testing tools. We measured the response time of each design alternative and monitored the resource utilisation with tools from the operating system. To exclude side effects during the testing session, we shut down background processes and repeated the requests on the web server over a 5-minute time interval for each alternative respectively. Finally, we used the averaged results of these measurements for the comparison with the predictions that is reported below.

4 Results

As result of the performance analysis for each approach, the response times for the single alternatives were noted. By the response time we refer to the time needed by the system for handling a user request from entering the system up to leaving the system. In the following we define metrics to answer the questions posed and to interpret the results regarding the goal of the case study. The formally defined metrics can be used in comparable studies.

4.1 How Precisely Can Quantitative Performance Characteristics Be Predicted with the Methods?

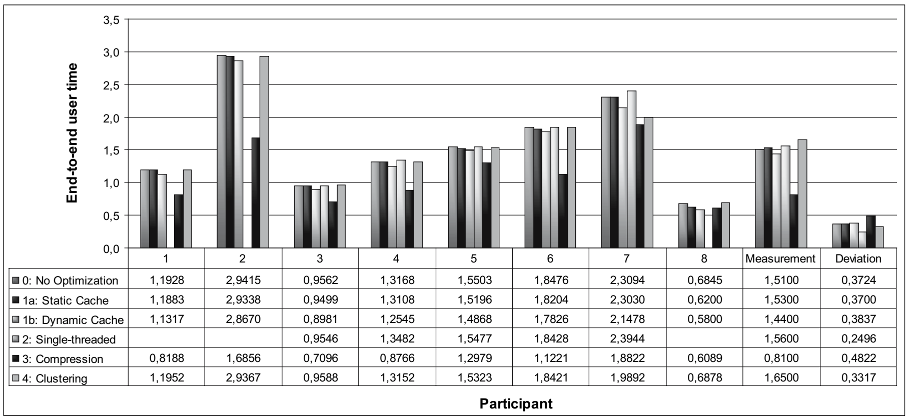
To answer the question we need a metric indicating the degree of discrepancy between the forecasted and measured values. For this reason the mean deviation between the predicted and measured response times in percent and not the Euclidean distance is determined among all alternatives and among all participants who predicted with a certain approach. Let $\mathcal{A}$ be the set of all design alternatives under consideration, n the number of participants applying the method, $rt_{i,j}$ the predicted response time from student i for the alternative j and $rt_{meas,j}$ the measured response time of the implementation of the alternative j. Then *Metric 1* can be specified as follows:

$$m1 := \frac{1}{|\mathcal{A}|} \sum_{j=1}^{|\mathcal{A}|} \frac{1}{n} \sum_{i=1}^{n} \frac{|rt_{i,j} - rt_{meas,j}|}{rt_{meas,j}}$$

Small values for *Metric 1* show a good suitability of the method for the evaluation of quantitative performance requirements.

In the following we present the predicted response times for each approach. The times are sorted according to design alternatives and participants, and are shown in a table using bar charts. In the second last column the actual measurements of the web server can be found, which enables the comparison with the predictions. The last column contains the mean deviation of the forecasts from the measurements in percent for each alternative determined among all participants. The bar chart of the last column was generated automatically and does not mean the absolute response times.

In figure 1 the response times forecasted with the *SPE-approach* are presented. With this approach several participants did not succeed in computing a response time for design alternative 2 (single-threaded version of the server) since the SPE-approach does not support modelling multiple threads. The deviations in the forecasts of individual participants are based on the fact that the SPE method needs input values, which are not present during the design phase of the system development. These values must be gathered from comparable systems or prototypes or have to be estimated on the basis of developers' experiences. In this experiment the input values were estimated by the participants. This explains the large deviations of the predicted values from the measurements. Nevertheless, significant patterns concerning the performance behaviour of the single design alternatives are observable.The *Metric 1* computed for the SPE approach

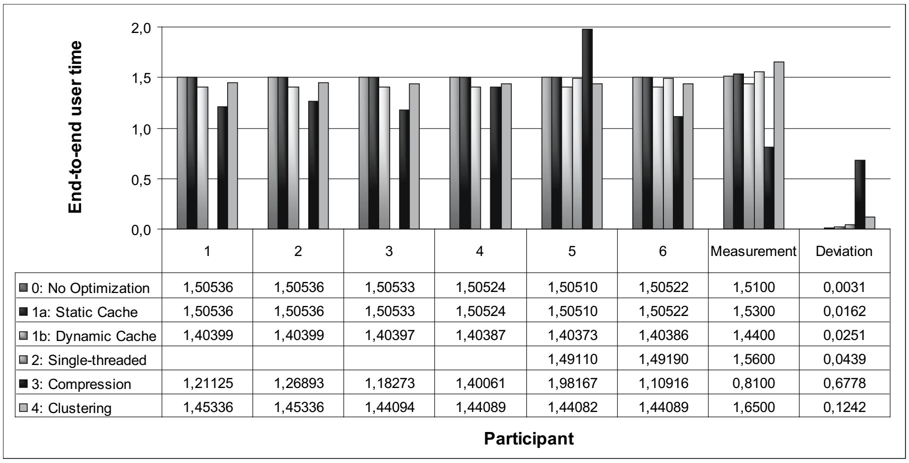


	1	2	3	4	5	6	7	8	Measurement	Deviation
▨ 0: No Optimization	1,1928	2,9415	0,9562	1,3168	1,5503	1,8476	2,3094	0,6845	1,5100	0,3724
■ 1a: Static Cache	1,1883	2,9338	0,9499	1,3108	1,5196	1,8204	2,3030	0,6200	1,5300	0,3700
▨ 1b: Dynamic Cache	1,1317	2,8670	0,8981	1,2545	1,4868	1,7826	2,1478	0,5800	1,4400	0,3837
▨ 2: Single-threaded			0,9546	1,3482	1,5477	1,8428	2,3944		1,5600	0,2496
■ 3: Compression	0,8188	1,6856	0,7096	0,8766	1,2979	1,1221	1,8822	0,6089	0,8100	0,4822
▨ 4: Clustering	1,1952	2,9367	0,9588	1,3152	1,5323	1,8421	1,9892	0,6878	1,6500	0,3317

Fig. 1. Predicted end-to-end user times, SPE method

amounts to 0,3649. That is, the forecasts deviate on the average over all participants and design alternatives by 36% from the measurements.

For the *CP-approach* the results of only 6 participants were gathered (figure 2) as two of the students were not able to finish their predictions within the available time. Just as with the SPE method, most participants did not succeed in modelling



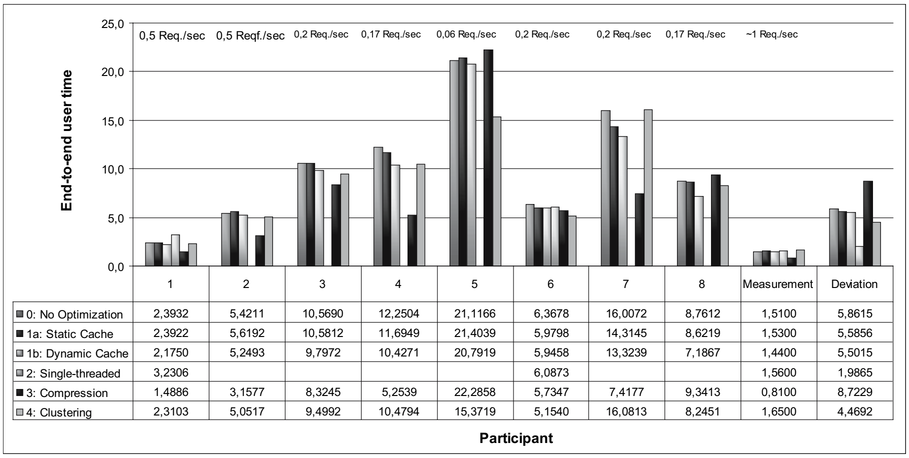
	1	2	3	4	5	6	Measurement	Deviation
▨ 0: No Optimization	1,50536	1,50536	1,50533	1,50524	1,50510	1,50522	1,5100	0,0031
■ 1a: Static Cache	1,50536	1,50536	1,50533	1,50524	1,50510	1,50522	1,5300	0,0162
▨ 1b: Dynamic Cache	1,40399	1,40399	1,40397	1,40387	1,40373	1,40386	1,4400	0,0251
▨ 2: Single-threaded					1,49110	1,49190	1,5600	0,0439
■ 3: Compression	1,21125	1,26893	1,18273	1,40061	1,98167	1,10916	0,8100	0,6778
▨ 4: Clustering	1,45336	1,45336	1,44094	1,44089	1,44082	1,44089	1,6500	0,1242

Fig. 2. Predicted end-to-end user times, CP method

alternative 2 and provided no results. With the CP-approach the forecasts of the participants, except alternative 3, differ very slightly from the measurement of the implementations. That is because of the fact that the method uses data measured on an available system. With alternative 3 a compression rate for delivered files had to be estimated, therefore noticeable deviations in the values arise. *Metric 1* computed for the CP-approach amounts to 0,1484. By about 15% deviation of prediction from measurements the CP method suites well for predicting future performance for existing systems.

Response times predicted with the *umlPSI-approach* are presented in figure 3. Apart from the infeasibility of modelling threads, the relatively large variation of obtained results is remarkable. With this approach, input data based on estimation is not separated into software and hardware resources, as it is the case with the SPE approach. In addition, the umlPSI approach relies on less input data, which makes the forecasts more inaccurate. The mean deviation between the predicted and measured response times computed via *Metric 1* is here over 500%.

To answer question 1 it can be stated that the CP approach is suitable for validation of performance requirements because of its comparatively high accuracy with the forecast of the absolute response times. However, the CP method requires measured response times of an existing system, therefore it can be only used for new developments in later phases of the software development.



	1	2	3	4	5	6	7	8	Measurement	Deviation
▩ 0: No Optimization	2,3932	5,4211	10,5690	12,2504	21,1166	6,3678	16,0072	8,7612	1,5100	5,8615
■ 1a: Static Cache	2,3922	5,6192	10,5812	11,6949	21,4039	5,9798	14,3145	8,6219	1,5300	5,5856
□ 1b: Dynamic Cache	2,1750	5,2493	9,7972	10,4271	20,7919	5,9458	13,3239	7,1867	1,4400	5,5015
□ 2: Single-threaded	3,2306					6,0873			1,5600	1,9865
■ 3: Compression	1,4886	3,1577	8,3245	5,2539	22,2858	5,7347	7,4177	9,3413	0,8100	8,7229
□ 4: Clustering	2,3103	5,0517	9,4992	10,4794	15,3719	5,1540	16,0813	8,2451	1,6500	4,4692

Fig. 3. Predicted end-to-end user times, umlPSI method

4.2 Do the Methods Really Support the Decision for the Design Alternative with the Fastest Implementation?

On the basis of the predicted response times the design alternatives can be ordered. The question is, how authentic this order is, if compared with performance of the implementation of alternatives. We need a metric indicating the number of wrong decisions suggested by a certain method.

For this reason first a ranking for the design alternatives is set up on the basis of the *measured* response times. Let $\mathcal{A}$ be the set of all design alternatives under consideration. We define a mapping

$$Pos_{meas} : \mathcal{A} \longrightarrow \{1, ..., |\mathcal{A}|\}$$

with $Pos(AlternativeA) \leq Pos(AlternativeB)$ for the response time of the alternative B being not less than the one of the alternative A. In the same manner the design alternatives are arranged for each participant into a ranking according to their *predicted* response times. The mapping Pos_{pred_i} arranges similarly to each alternative its place in the ranking according to the response times predicted by participant i.

In the next step we want to calculate, how many positions the measurement-based ranking differs from the prediction-based ranking. Since we do not want to take into account place permutations in cases of insignificant differences in response times we divide the alternatives according their *measured* times into classes applying a distance-based clustering algorithm [1]. In this way alternatives with very close response times are regarded equivalent and belong to the same class. The monotonically increasing mapping $Class : Pos_{meas}(\mathcal{A}) \longrightarrow \{\infty, ..., |\mathcal{C}\updownarrow\dashv\int\int\rceil\int|\}$ assigns each position of the measurement-based ranking to the class of the associated design alternative.

Subsequently we define

$$Displacement_i(Alt) := |Class(Pos_{pred_i}(Alt)) - Class(Pos_{meas}(Alt))|.$$

This formula indicates for the participant i the number of place discrepancies between his predicted ranking of design alternative Alt and the measurement-based ranking of this alternative if the rank positions belong to different clusters. *Metric 2* is defined as follows:

$$m2 := \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{|\mathcal{A}|} Displacement_i(Alt_j),$$

with n being the number of participants predicting with a certain method. The smaller the value of the metric $m2$ the less false placements were obtained by ordering the alternatives according to timing behaviour.

Figure 4 shows the comparison of the measured and the predicted rank lists of our experiment. On the basis of measurements the alternative 3 was assigned to the class 1

			SPE								CP						umlPSI							
	Classes	Measurement	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	Participant 7	Participant 8	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	Participant 1	Participant 2	Participant 3	Participant 4	Participant 5	Participant 6	Participant 7	Participant 8
Ranking Design Alternatives:																								
1. (fastest)	Class 1	3	3	3	3	3	3	3	3	1b	3	3	3	3	1b	3	3	3	3	3	4	4	3	4
2.	Class 2	1b	1b	1b	1b	1b	1b	1b	4	3	1b	1b	1b	1b	4	1b	1b	4	4	1b	1b	3	1b	1b
3.	Class 2	1a	1a	1a	1a	1a	1a	1a	1b	1a	4	4	4	4	2	4	4	1b	1b	4	1a	1b	1a	1a
4.	Class 2	2	4	4	2	4	4	4	1a	4	1a	1a	1a	1a	1a	1a	1a	1a	1a	1a	3	1a	4	3
5. (slowest)	Class 2	4	2	2	4	2	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2
Metric 2:			**0,25**								**0,33**						**0,75**							

Fig. 4. Ranking of design alternatives

and other alternatives to the class 2, because the response time of alternative 3 clearly differs from others. With the SPE and CP approaches there are only two "‘incorrect"’ placements in the prediction-based ranking of one participant in each case. Other place permutations among the alternatives 1a, 1b, 2 and 4 occur within a cluster and therefore do not taken into account. With the umlPSI approach more often an alternative of the "‘slower"’ class was forecasted being the fastest, which results in the notably higher value for *Metric 2*.

As answer to question 2 it can be stated that both the SPE and CP approach are well suited for supporting the selection of the best design alternative concerning timing behaviour.

4.3 How Does the Application of the Methods Compare to an Unmethodical Approach to Performance Prediction?

The tasks of the experiment were additionally given to seven members of the control group, who did not take part in the tutorials. Doing so we wanted to find out whether the intuitive evaluation of design alternatives leads to different decisions than applying one of the methods. The members of the control group did no ranking of the alternatives but selected the best one on their own opinion. Four of the seven members of the control group preferred the alternative 1b (dynamic cache), only three participants picked alternative 3 (compression). Thus, the larger part of the control group took an alternative, which had the best response times in only 12% of the method-supported predictions. Although the sample of seven participants is very small, this result at least implies that analysing the design documents does not lead necessarily to the decision for design alternative 3 and that our experiment's setup was valid.

4.4 What Are the Properties of the Methods and Where is Room for Improvement?

We grouped the properties of the methods into four different subsections: expressiveness, tool support, domain compatibility and process integration, and report our experiences with them during the experiment.

Expressiveness. The SPE method has a quite intuitive separation of a software and a system model, which allows changing hardware and software independently for a quick analysis of different designs. The underlying queueing networks are encapsulated by the method's tool and are not of the developers' concern. One scenario may be modelled over multiple levels of control flow making it easy to model complex situations. Multiple scenarios on one resource may be evaluated via simulation directly within the tool. Although it is possible to model quite complex systems, the students were not able to model multiple threads within one process. An advanced system model for different queueing disciplines and the inclusion of passive resources has been proposed [15], yet is not implemented in the tool. The SPE method allowed the students to make predictions on the performance of the architecture with the fewest input data of all three analysed methods. Thus it is suitable for application during early development cycles when details of the complete architecture are still unknown.

Other than the SPE method, in CP only a system model is used for the analysis. No explicit software model is included. The workload of a system is characterised by traces of requests and the output of monitoring tools. It is necessary to have at least a prototype of the new architecture to collect this data. In our experiment we had to include a workload-trace of a prototype of the web server into the design documents, so that the students could apply the method. Many different queueing networks may be modelled, for example with multiple queues before one server or the blocking of resources. This allows a lot of real-life situations to be analysed. Yet software tools are missing for some of these networks and the analysis has to be done manually in this case. Like in SPE the students had difficulties modelling multiple threads. A hierarchical structuring of complex queueing networks is not provided. Because this method relies on measured performance data, it is better suited for performance planning when extending existing systems.

The expressiveness of the umlPSI approach is as powerful as UML diagrams (use-case-, activity- and deployment-diagrams) and the UML profile used. Using UML was intuitive for the students, who had learned this modelling language in a course about software engineering. Some students complained that the modelling of hardware resources with deployment diagrams was not expressive enough because the UML Performance Profile only allows the definition of a speed-factor to differentiate resources. Multiple threads could be modelled through a passive resource with a pool of threads.

The generated simulation is more powerful than the analysis models of the previous two methods. For example, it allows arbitrary distributions of incoming requests, while queueing networks are bound to an exponential distribution.

Tool support. The SPE-method comes with the tool SPE-ED (Software Performance Engineering EDitor) for Windows, which can be downloaded in a demo version from the authors' website [13]. The full version for commercial use has to be purchased, yet a price is not provided. SPE-ED has been developed several years ago and the user interface is not up to the latest standards. For example, there is no drag and drop and students complained that the graphical modelling is unintuitive and quite different from similar tools. There is no integration with other modelling tools, although an export function to a XML-based file format has been proposed, but is not implemented yet [16]. Having manually entered software execution graphs the performance modeller is able to do a quick analysis with the tool highlighting expected bottlenecks of his architecture. The tool is stable and the students hardly reported any crashes. A problem of the tool is that it is not possible to feed the performance results directly back into the software model (e.g. in UML) and that software model and performance model have to be maintained separately. Overall, the tool appears useful for serious performance modelling and is the most matured of the analysed tools.

Although several tools exist to solve queueing networks, we decided for the CP-method to use the spreadsheets provided by the authors, which can be downloaded free of charge from their website [9]. There are several versions of the spreadsheets, each one for a different type of queueing network. Spreadsheets are also available for workload clustering as described in the book. The spreadsheets for queueing networks let the user put in the type and number of resources, the expected calculation time on each resource and the arrival rate of incoming requests. They output response times, resource utilizations, queue length and throughput, but do not visualise the results. A graphical visualisation would have been useful because the output may consist of a large amount of numbers, which have to be carefully analysed. Some students reported that they had trouble interpreting the results. Because the queueing networks do not contain an explicit software model and are resource-oriented, reporting back the results into the software model is impossible. The spreadsheets are appropriate for their actual purpose, which is capacity planning for existing information systems. However their usefulness is limited when analysing large software architectures.

As mentioned earlier, an UML modelling tool is necessary before using the free umlPSI Linux-tool, because umlPSI only generates and executes simulations out of XMI-files. Like the authors we used Poseidon for modelling and annotating UML-diagrams in our experiment. umlPSI in its current implementation is bound to a specific version of Poseidon, since it can only process the XMI-dialect generated by this version properly. The installation of this tool was difficult because it requires some libraries which are not available on every Linux distribution. Most of the students did not manage to install and run umlPSI and only submitted their annotated UML-models exported from Poseidon. Generation and execution of the simulation was also difficult because the tool produced incomprehensible error messages if the XMI-files contained syntax errors, which occurred several times during our testing. After the simulation is executed properly the tool outputs a result table and also reported the results back into the XMI-files, which could be reimported to Poseidon. There the students were able to inspect bottlenecks and make adjustments to their architectures. It would be practical if the tool was available as a plugin for a modelling tool and had not to be started externally. Overall umlPSI is intended as a proof-of-concept implementation for the method and cannot been seen as a tool ready for industrial use.

Domain applicability. The application domain of the authors of the SPE-method are distributed systems and especially web applications. Their documentation contains several examples of systems providing services to the web. They also documented the application of the SPE-method to embedded real-time systems. The SPE approach is very flexible and can be applied to lots of different contexts.

Like SPE, CP is also developed with the focus of web applications and client/server systems as analysis subjects. The authors have written five different books about the topic with varying emphasises (e.g. Web Server, E-Business, Client/Server).

The umlPSI tool so far is able to analyse use-case-, activity-, and deployment-diagrams and is therefore bound to the modelling possibilities of these three diagram types. For example the inclusion of component-diagrams is not possible. The authors tested this method with examples of small web applications.

Process integration. For the SPE method a well-documented process consisting of several steps is available. Although it is originally designed for the application during the design phase of a waterfall model during software development, the authors also show, that the method can be easily embedded into iterative models like the spiral model or the unified process.

CP contains a process model, but it has to be adopted individually depending on the system under analysis. It is best suited to be applied during implementation, testing or evolution stages of a software system.

The umlPSI tool can be used every time the developers are concerned with UML modelling, mostly during early stages of the development. It is not bound to a specific process model.

5 Validity of the Experiment

Several limitations affect the validity of our results. We discuss the internal and external validity of our experiment.

The *internal validity* is the degree, to which every relevant interfering variable could be controlled. This includes the degree to which the results of the experiment are indeed attributed to the properties of the performance prediction methods and not bound to mistakes in our experimental set-up.

One interfering variable we could not control was the input to the methods. For the SPE-approach and the umlPSI-approach the students had to base their predictions on estimated values. Instead, for the CP-approach we included a workload trace with measured values into the experimental task, because this is common when applying this approach.

The disposition of the students into each group was based on the results of their exercise papers. This way we controlled the possibility of inhomogeneous groups with deviating qualifications.

The *external validity* is the degree, to which the results of the experiment are valid in other situations than the one analysed here. This especially concerns the participants and the analysed subject.

The participants in our experiment were students and we could not include practitioners with more experience. However, due to the training session, every student had a similar knowledge about the performance prediction methods. This way, past experiences of the participants did not distort the results, which would have been possible if practitioners had been included into the experiment.

The number of students was limited to eight for each method, so a statistical hypothesis test was not possible due to the limited available data points. However, individual exceptions in student performance could be detected in principle. By this the influence of individual performance was controllable.

The students analysed a small architecture, the scalability of the methods to larger architectures is unknown. Our implementation of the web server is only one possible implementation, the performance measurements of other implementations of the same design might differ.

We analysed only three methods, although more than 20 approaches are known [3]. The validity of our experimental set-up applied to other methods is unknown.

6 Conclusions

In this paper we compared three different methods for the early performance analysis of software systems. A group of students applied the methods on design documents and made performance predictions for the architecture of a web server. After implementing the web server we were able to compare predictions with actual measurements.

The SPE method was appropriate to support design decisions. Yet the validation of performance goals was hardly possible because the predictions relied on students' estimations and a satisfying precision could not be achieved. The method is especially suited for early, explorative predictions because it is able to compute results with only few inputs. A better integration of this method into common modelling tools would be of practical interest.

CP delivered the most precise predictions, but relied on a workload trace of the web server taken from a prototype. Thus, the method is well suited for extending existing system and also able to validate performance goals. But the use for early performance predictions of new architectures is limited.

umlPSI might be most convenient for developers because it relies on UML models and does an automatic transformation of software models to performance simulations. However the simulation was more time-consuming than the analysis of the other methods and the tool proved to be error-prone. The umlPSI method can be used during early development cycles, yet the precision of the performance results was the worst in our experiment.

Future work in this area includes the conduction of a field study in an industrial environment possibly with a large software architecture. Other methods, especially those for component based systems [5], will also be evaluated.

Acknowledgements. We would like to thank Yvette Teiken for implementing the web server, members of the Palladio research group for fruitful discussions and all students, who volunteered to participate in our experiment.

References

1. M.R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.
2. S. Balsamo, A. DiMarco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
3. S. Balsamo, M. Marzolla, A. DiMarco, and P. Inverardi. Experimenting different software architectures performance techniques: A case study. In *Proceedings of the Fourth International Workshop on Software and Performance*, pages 115–119. ACM Press, 2004.
4. V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. *Encyclopedia of Software Engineering - 2 Volume Set*, pages 528–532, 1994.
5. A. Bertolino and R. Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, volume 3054 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2004.
6. I. Gorton and A. Liu. Performance evaluation of alternative component architectures for enterprise javabean applications. *IEEE Internet Computing*, 7(3):18–23, 2003.
7. H. Koziolek. Empirische bewertung von performance-analyseverfahren fr software-architekturen. Diploma thesis, University of Oldenburg, Faculty II, Department of Computing Science, Okt. 2004.
8. M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universit‘a Ca Foscari di Venezia, 2004.
9. D. A. Menasc, V. A. F. Almeida, and L. W. Dowdy. Download files for the book: Performance by design: computer capacity planning by example. `http://cs.gmu.edu/~menasce/perfbyd/efiles.html`, 2004.
10. D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.
11. D. A. Menasce and V. A.F. Almeida. *Capacity Planning for Web Services*. Prentice-Hall, 2002.
12. Object Management Group OMG. Uml profile for schedulability, performance and time. `http://www.omg.org/cgi-bin/doc?formal/2003-09-01`, 2003.
13. C. U. Smith. Speed: The software performance engineering (spe) tool. `http://www.perfeng.com/sped.htm`, Jan 2000.
14. C. U. Smith. *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*. Addison-Wesley, 2002.
15. C. U. Smith. Spe-ed user guide. `http://www.perfeng.com/papers/manual.zip`, 2003.
16. C. U. Smith and C. M. Llad. Performance model interchange format (pmif 2.0): Xml definition and implementation. Technical report, Performance Engineering Services, Universitat Illes Balears, 2004.
17. C. Wohling, P. Runeson, M. Hst, M.C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering – An Introduction*. Kluwer Academic Publishers, 2000.

Automatic Test Generation for N-Way Combinatorial Testing [*]

Changhai Nie[1], Baowen Xu[1,2], Liang Shi[1], and Guowei Dong[1]

[1] Dept. of Computer Sci. & Eng., Southeast University, Nanjing 210096, China
[2] State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China
{changhainie, bwxu, shiliang_ada}@seu.edu.cn

Abstract. *n*-way combinatorial testing is a specification-based testing criterion, which requires that for a system consisting of a few parameters, every combination of valid values of arbitrary n ($n \geq 2$) parameters be covered by at least one test. In this paper, we propose two different tests generation algorithms based on the combinatorial design for the *n*-way combination testing. We show that the produced tests can cover all the combinations of parameters to the greatest degree with the small quantity. We implemented the automatic test generators based on the algorithms and obtained some valuable empirical results.

Keywords: software testing, combinatorial testing, test generation.

1 Introduction

Automatic test generation is an important research subject in software testing, which selects a small number of tests from a large test suite to cover all test aspects to the greatest degree. The combinatorial design approach for testing has been studied and applied widely for its efficiency and effectiveness with a quite small test suite.

R. Mandl defined the concept of pair-wise testing for the first time in 1985[1], which requires that every combination of valid values of arbitrary two system parameters be covered by at least one test. His method utilized the orthogonal Latin squares to generate tests, which was applied in the Ada compiler testing.

D. Cohen, et al. provided a strategy that used combinatorial design approaches to generate tests that could cover the pair-wise, triple, or *n*-way combinations of a system's parameters[2,3]. They implemented the strategy in the AETG system, which has been applied in a variety of applications for unit, system and interoperability testing. Y. Lei, et al. also offered a test generation method for pair-wise testing in parameter order[4,5], and implemented it in the PAIRWISE TEST system. We proposed an algorithm for pair-wise testing based on the network model[6], and developed a tool to generate tests automatically. Toshiaki Shiba et al. also suggested two different algo-

* This work was supported in part by the National Natural Science Foundation of China (60425206, 60373066, 60403016), National Grand Fundamental Research 973 Program of China (2002CB312000) and the Natural Science Foundation of Jiangsu Province(BK2005060).
Correspondence to: Baowen Xu, Department of Computer Science and Engineering, Southeast Univsity, 210096 Nanjing, China. Email: bwxu@seu.edu.cn

rithms GA and ACA for n-way testing. Because the problem of generating a minimum test suit for combinatorial testing is *NP-complete*[4], none of these methods can assure that the generated test suites are the best. In most cases, the sizes of the test suites generated by them are approximately equal.

D.R. Kuhn et al. investigated the applicability of the design of experiments to software testing in 2001[8]. In their experiments, about 20~40% software failures were caused by a certain value of one parameter, and about 20~40% software failures were caused by a certain combination of two parameters, and about 20% software failures were caused by a certain combination of three parameters. Thus they believe that the combinatorial design approach for automatic test generation is quite effective in many cases.

While pair-wise testing is fit for most general applications, in some safety-critical situations, to assure the software stability, testers may require that for a system consisting of a few parameters, every combination of valid values of arbitrary n ($n \geq 2$) parameters be covered by at least one test. In this paper, we propose two different test generation algorithms based on the combinatorial design for the n-way testing.

The rest of the paper is organized as follows. In section 2, some basic definitions and theorems are introduced. In section 3, two different algorithms are presented and analyzed. Finally, some empirical results are shown in section 4.

2 Basic Concepts of *N*-Way Testing

Suppose a system has k parameters P_1, P_2… P_k, where parameter P_i has t_i values, $1 \leq i \leq k$. For covering all the combinations of all the parameters, $t_1 \times t_2 \times \ldots \times t_k$ test cases are needed. Since it is too expensive to test all the combinations in most conditions, a more practical method is to choose tests based on some testing criterions.

There are many factors that could cause a software failure: a certain value of one certain parameter, a certain combination of certain two parameters… a certain combination of certain k-1 parameters, or a certain combination of all the k parameters. Consequently it is profitable to generate a small test suite to cover all the possible combinations to the greatest degree. It is evident that if a test suit can cover all the combinations of certain l ($1 \leq l \leq k$) parameters, it can do so to all the l-1, l-2… 1 parameters in the l parameters.

Definition 1. Suppose A is an $m \times k$ matrix, and column j represents parameter P_j, whose element is from the finite symbol set T_j (j = 1, 2… k). If column i and column j are arbitrary two columns in A which satisfy that all the combinations of symbols between set T_i and set T_j appear in the ordered pairs formed by column i and column j, A is called a ***pair-wise test table***, which satisfies the ***pair-wise coverage criterion***. Every row of A is a piece of test case, and m is the number of tests.

Definition 2. Suppose A is an $m \times k$ matrix, and column j represents parameter P_j, whose element is from the finite symbol set T_j (j = 1, 2… k). If columns i_1, i_2… i_n are arbitrary n ($2 \leq n \leq k$) columns in A that satisfy: all the combinations of symbols among set T_1, set T_2… set T_n appear in the ordered combinations formed by column i_1, column i_2... column i_n, A is called an ***n-way test table***, which satisfies the ***n-way coverage criterion***. Every row of A is a piece of test case, and m is the number of tests.

It can be known from the definitions that an n-way test table is a pair-wise test table when n equals 2. Thus, algorithms for the n-way combination testing can also generate tests for the pair-wise testing.

Theorem 1. Suppose a system S has k parameters $P_1, P_2 \ldots P_k$, and every parameter P_i has t_i (i =1, 2, ..., k) values. Then the system's n-way test table A has r rows, where

$$r \geq \max(t_{i_1} \times t_{i_2} \times \cdots \times t_{i_n}),\ (1 \leq i_1 \neq i_2 \neq \ldots \neq i_n \leq k).$$

Proof. Because the number of combinations of n parameters is $t_{i_1} \times t_{i_2} \times \cdots \times t_{i_n}$, the n-way test table A must have at least $\max(t_{i_1} \times t_{i_2} \times \cdots \times t_{i_n})$ rows to cover all the combinations of the n parameters.

Theorem 2. Suppose a system has k parameters, and every parameter P_i has t_i (i = 1, 2... k) values, where $t_1 \geq t_2 \geq \ldots \geq t_k$. Then the system's n-way test table A has coverage p to all the combinations of arbitrary m ($n \leq m \leq k$) parameters:

$$p \geq (t_1 \times t_2 \times \ldots \times t_n) / (t_1 \times t_2 \times \ldots \times t_m).$$

Proof. To m parameters $P_{i_1}, P_{i_2}, \cdots, P_{i_m}$, assume that they have the number of values $t_{i_1} \geq t_{i_2} \geq \cdots \geq t_{i_m}$ respectively. Based on Theorem 1, the system has an n-way test table A whose number of rows is not less than $t_{i_1} \times t_{i_2} \times \cdots \times t_{i_n}$. Since the combination number of the m parameters is $t_{i_1} \times t_{i_2} \times \cdots \times t_{i_m}$, the table's coverage p to the m parameters satisfies

$$p \geq \frac{t_{i_1} \times t_{i_2} \times \ldots t_{i_n}}{t_{i_1} \times t_{i_2} \times \ldots t_{i_m}} = \frac{1}{t_{i_{(n+1)}} \times t_{i_{(n+2)}} \times \cdots \times t_{i_m}} \geq \frac{1}{t_{n+1} \times t_{n+2} \times \cdots \times t_m} = \frac{t_1 \times t_2 \times \ldots \times t_n}{t_1 \times t_2 \times \cdots \times t_m}.$$

Altogether, the n-way test table is more suitable for software testing because it covers all the factors to the greatest degree with a small test suite. We will introduce how to generate this kind of table in the next section.

3 *N*-Way Testing Algorithms

Suppose a system has k parameters P_1, P_2... P_k, where parameter P_i has a suite of discrete value T_i, $1 \leq i \leq k$. Let $t_i = |T_i|$, where $t_1 \geq t_2 \geq \ldots \geq t_k$. The variable n ($2 \leq n \leq$ k) is assigned by testers.

3.1 Generate Tests Using Greedy Algorithm

D. Cohen, et al. provided an algorithm using combinatorial design approaches to generate the pair-wise test tables[1], and mentioned that their AETG could also generate n-way test tables. However, they did not offer an algorithm for n-way test tables in detail. This paper proposes a test generate method for n-way testing based on extending their strategy for pair-wise testing.

The algorithm GA-N (showed in Fig.1) is a greedy algorithm, which generates tests piece by piece. Firstly, construct a set *Uncover* that contains all the combinations of valid values of arbitrary *n* parameters. Secondly, generate one test such that it can cover most combinations in *Uncover*. Add the test into the *n*-way test table *M*, and delete the combinations covered by it from *Uncover*. Finally, generate tests continuously in the same way until *Uncover* is empty.

```
Input: $T_1, T_2, \ldots, T_k$
Output: $n$-way test table $M$
begin
  $Uncover := \{(v_{i_1}, v_{i_2}, \cdots, v_{i_n}) \mid v_{i_1} \in T_{i_1}, v_{i_2} \in T_{i_2}, \ldots, v_{i_n} \in T_{i_n} \wedge i_1 \neq i_2 \neq \ldots \neq i_n \wedge 1 \leq i_1, i_2, \ldots, i_n \leq k\}$;
   while ( Uncover is not empty )
     select a combination $(p_{i_1}, p_{i_2}, \cdots, p_{i_{n-1}})$ of parameters $P_{i_1}, P_{i_2}, \cdots, P_{i_{n-1}}$ that is covered by most elements in Uncover;
     $P_1 := P_{i_1}$, $P_2 := P_{i_2}$, …, $P_{n-1} := P_{i_{n-1}}$, and choose a random order for the remaining parameters;
     $v_1 := p_{i_1}$, $v_2 := p_{i_2}$, …, $v_{n-1} := p_{i_{n-1}}$; /*This statement and the for-statement below generate a new test $(v_1, v_2, \ldots, v_k)$*/
      for i := n to k do //(1)
         for $x$ := 1 to $t_i$ do //(2)
          construct $count_x$ for parameter $P_i$'s value $y_x (1 \leq x \leq t_i)$, and initialize them by 0;
            select $(n-1)$ values from $v_1, v_2, \ldots, v_{i-1}$ to form $n$-way combinations with $y_x$, and the number of the combinations is $C_{i-1}^{n-1}$;
            check each combination one by one to determine whether it is an element of Uncover, and use $count_x$ to record the number of combinations that are members of Uncover;
         end for
         Select $count_m = \max\{ count_x \mid 1 \leq x \leq t_i \}$;
         $v_i := y_m$;
      end for
      add test $(v_1, v_2, \ldots, v_k)$ to $M$;
      delete all the combinations covered by $(v_1, v_2, \ldots, v_k)$ from Uncover;
   end while
end
```

Fig. 1. Greedy algorithm for *n*-way test table generation (GA-N)

To illustrate GA-N (n=2), consider a system with 3 parameters: parameter A has values A_1 and A_2, and parameter B has values B_1 and B_2, and parameter C has values C_1 and C_2. Firstly, set *Uncover* $=\{(A_1,B_1), (A_1,B_2), (A_2,B_1), (A_2,B_2), (A_1,C_1), (A_1,C_2), (A_2,C_1), (A_2,C_2), (B_1,C_1), (B_1,C_2), (B_2,C_1), (B_2,C_2)\}$. Because none of the pairs have been covered, add test (A_1,B_1,C_1) to the test table M and set *Uncover* $=\{(A_1,B_2), (A_2,B_1), (A_2,B_2), (A_1,C_2), (A_2,C_1), (A_2,C_2), (B_1,C_2), (B_2,C_1), (B_2,C_2)\}$. For there is 2 A_1, 4 A_2, 2 B_1, 4 B_2, 2 C_1 and 4 C_2 in *Uncover* now, we choose A_2 to generate test $(A_2,\text{-},\text{-})$. Since *Uncover* contains both (A_2,B_1) and (A_2,B_2), we can choose B_1 to obtain test $(A_2,B_1,\text{-})$. If we Add C_1 for $(A_2,B_1,\text{-})$, the extended test (A_2,B_1,C_1) covers (A_2,C_1) in

Uncover. If we add C_2 for $(A_2,B_1,-)$, the extended test (A_2,B_1,C_2) covers (A_2,C_2) and (B_1,C_2) in *Uncover*. Therefore we choose C_2 for $(A_2,B1,-)$, add (A_2,B_1,C_2) to M, and set $Uncover = \{(A_1,B_2), (A_2,B_2), (A_1,C_2), (A_2,C_1), (B_2,C_1), (B_2,C_2)\}$. Then generate tests continuously in the same way until *Uncover* is empty.

```
Input: $T_1, T_2, \ldots, T_k$
Output: $n$-way test table $M$
begin
  //Construct M for the first n parameters
  $M := \{(v_1, v_2, \ldots, v_n) \mid v_1 \in T_1, v_2 \in T_2, \ldots, v_n \in T_n\}$;
  for $i := n+1$ to $k$ do //(1) Extend M by other parameters
    $\pi := \{ (p_{m_1}, p_{m_2}, \cdots, p_{m_{n-1}}, p_{m_n}) \mid p_{m_1} \in T_{m_1}, p_{m_2} \in T_{m_2}, \ldots, p_{m_{n-1}} \in T_{m_{n-1}}$
        $\wedge\ m_1 \neq m_2 \neq \ldots \neq m_{n-1} \wedge 1 \leq m_1, m_2, \ldots, m_{n-1} \leq i-1 \ \wedge\ p_{m_n} \in T_i \}$;
    for $j := 1$ to $|M|$ do //(2) Horizontal growth
      if ($\pi$ is not empty ) then
        $\pi' := \{\}$;
        for each $v$ in $T_i$ do //(3)
          $\pi'' := \{z \mid z \in \pi \wedge z$ is covered by the extended test $(v_1, v_2, \ldots, v_{i-1}, v_i)\}$;
            //(4)
          if ($|\pi''| \geq |\pi'|$) then
            $\pi' := \pi''$; $v' := v$;
          end if
        end for
        extend the $j$-th test in $M$ by $v'$: $(v_1, v_2, \ldots, v_{i-1})$ is extended to $(v_1, v_2, \ldots, v_{i-1}, v')$;
        $\pi := \pi - \pi'$; //(5)
      else
        select $v \in T_i$ arbitrarily, and extend the $j$-th test in $M$ by $v$;
      end if
    end for
    while ( $\pi$ is not empty ) do //(6) Vertical growth
      generate one test $t$ such that it can cover most combinations in $\pi$;
      add $t$ into $M$, and delete all the combinations covered by $t$ from $\pi$;
    end while
  end for
end
```

Fig. 2. *N*-way test table generation In Parameter Order (IPO-N)

In the following analysis of the runtime complexity of algorithm GA-N, let $d = \max\{t_i \mid 1 \leq i \leq k\}$. The initial size of *Uncover* is $C_k^n \times d^n$ at most. Since GA-N can account $count_x$ $(1 \leq x \leq t_i)$ by scanning *Uncover* once, the statement (2) takes O(|*Uncover*|) time in an iteration cycle. Since the statement (2) iterates at most d times, it takes O(|*Uncover*| $\times$ d) time totally. For the statement (1) iterates k-n+1 times, it takes O(|*Uncover*|$\times d \times k$) time in all. In the worst condition, the while-statement iterates |*Uncover*| times, so the overall runtime complexity of GA-N is O(|*Uncover*|$^2 \times d \times k$) = $O((C_k^n)^2 \times d^{2n+1} \times k)$.

In some cases, the test table is required to be constructed based on some appointed tests. The requirement can be satisfied by GA-N. Firstly, construct *Uncover* that contains all the combinations of n parameters. Secondly, add the appointed tests into M, and delete the combinations covered by them from *Uncover*. Then generate tests continuously until *Uncover* is empty.

3.2 Generate Tests in Parameter Order

Y. Lei et al. offered a test generation algorithm for pair-wise testing in parameter order [3]. This paper extends it to construct the n-way test table.

The algorithm IPO-N (showed in Fig. 2) is also a greedy algorithm, which generates tests in parameter order. Firstly, construct n-way test table M for the first n parameters. Secondly, extend M by another parameter to assure that every combination of valid values of n parameters in M is covered by at least one test. Then extend M continuously, until it contains all the parameters. The extension for the addition of a new parameter includes the following two steps: (1) horizontal growth, which extends each existing test by one value of the new parameter, and (2) vertical growth, which adds new tests to satisfy the n-way coverage criterion.

To illustrate IPO-N, consider a system with 3 parameters: parameter A has values A_1 and A_2, parameter B has values B_1 and B_2, and parameter C has values C_1, C_2 and C_3. Firstly, construct test table $M = \{(A_1,B_1), (A_1,B_2), (A_2,B_1), (A_2,B_2)\}$for A and B. Secondly, extend tests in M by the values of C and obtain three extended tests (A_1,B_1,C_1), (A_1,B_2,C_2) and (A_2,B_1,C_3). Now $\pi = \{(A_2,C_1), (B_2,C_1), (A_2,C_2), (B_1,C_2), (A_1,C_3), (B_2,C_3)\}$, and we need choose a value of C to extend (A_2, B_2). If we add C_1 for (A_2,B_2), the extended test (A_2,B_2,C_1) covers (A_2,C_1) and (B_2,C_1) in π. If we add C_2 for (A_2,B_2), the extended test (A_2,B_2,C_2) covers (A_2,C_2) in π. If we Add C_3 for (A_2,B_2), the extended test (A_2,B_2,C_3) covers (B_2,C_3) in π. Therefore we can choose C_1 to extend (A_2,B_2), add (A_2,B_2,C_1) to M, and delete (A_2,C_1) and (B_2,C_1) form π. Finally apply a method similar to the algorithm shown in Fig.1 to generate tests to cover remaining combinations in π.

In the following analysis of the runtime complexity of algorithm IPO-N, let $d = \max\{t_i \mid 1 \le i \le k\}$. The initial size of π is $d^n \times C_{i-1}^{n-1}$ at the most. Since IPO-N can determine whether z is covered by scanning π once, the statement (4) takes $O(|\pi|)$ time. Thus the statement (3) takes $O(|\pi| \times d)$ time. Because IPO-N can delete elements of π' from π by scanning π once, the statement (5) takes $O(|\pi|)$ time. The statement (2) iterates $|\pi|$ times at the most, so it takes $O(|\pi|^2 \times d)$ time in all. And, the while statement (6) iterates at most π times, so it takes O(π) time totally. The statement (1) iterates k-n times, so the overall runtime complexity of IPO-N is $O((|\pi|^2 \times d + \pi) \times k)$ = $O((C_{k-1}^{n-1})^2 \times d^{2n+1} \times k)$.

4 Effectiveness Analysis

To evaluate the efficiency of the algorithms, we implemented two automatic test generators based on them. We experimented with a computer consisting of a PIII processor of 733Mhz, and obtained some empirical results. In our experiments, $P^{(Q)}$ represents a system consisting of Q parameters, each of which has P values. For example, $4^{(5)}\times 3^{(12)}$ represents a system of 17 parameters: 5 parameters have 4 values, and the other 12 parameters have 3 values.

Table 1 displays the number of tests generated by the two algorithms for different systems and the time consumed for the generation. When *n* equals 2, the tests generated by GA-N are usually a few more than IPO-N, and the consumed time is also a little more, and the difference is not distinct. When *n* equals 3, the tests generated by GA-N are much more than IPO-N, and the consumed time is also more in evidence. Some other experiment results also show that IPO-N exceeds GA-N at the test number and the consumed time, when *n* is above 3.

Table 1. Comparison between GA-N and IPO-N(*n*=2 or 3)

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
GA-N	29	58	12	50	264	31	56	66
(*n*=2)	1”	1”	1”	10”	6”	1”	4”	7”
IPO-N	22	55	10	21	275	33	47	36
(*n*=2)	1”	1”	1”	3”	1”	1”	2”	2”
GA-N	103	375	22	305	5424	172	390	418
(*n*=3)	3’’	19”	1”	2539”	17767”	2”	1534”	1638”
IPO-N	86	288	19	128	2774	153	244	216
(*n*=3)	1”	1”	1”	859”	904”	1”	296”	592”

S_1: $3^{(13)}$; S_2: $5^{(10)}$; S_3: $2^{(10)}$; S_4: $2^{(100)}$; S_5:$10^{(20)}$;
S_6: $5^{(3)}\times 4^{(3)}\times 3^{(1)}\times 2^{(2)}$; S_7: $4^{(15)}\times 3^{(17)}\times 2^{(29)}$; S_8: $4^{(1)}\times 3^{(39)}\times 2^{(35)}$;

Table 2. Comparison with existing algorithms (*n*=3)

	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
AETG	38	77	194	330	1473	218	114	377
GA	33	64	125	331	1501	218	108	360
ACA	33	64	125	330	1496	218	106	361
GA-N	52	85	223	389	1769	366	120	373
IPO-N	47	64	173	271	1102	199	113	368

X_1: $3^{(6)}$; X_2: $4^{(6)}$; X_3: $5^{(6)}$; X_4: $6^{(6)}$; X_5:$10^{(6)}$; X6: $5^{(7)}$;
X_7: $5^{(2)}\times 4^{(2)}\times 3^{(2)}$; X_8: $10^{(1)}\times 6^{(2)}\times 4^{(3)}\times 3^{(1)}$.

Although GA-N can generate the *n*-way coverage test table from a special test suit appointed by testers, IPO-N can achieve the same goal by adding the appointed tests into the generated ones. These results suggest that IPO-N may be a more practical *n*-way combination-testing algorithm, when *n* is above 3.

Table 1 also shows that combinatorial testing can reduce the cost of software testing significantly. 1594323(3^{13}) tests are needed for the full coverage of S_1. When n equals 2, IPO-N generates 22 tests for the pair-wise coverage criterion, so that 99.999% tests are reduced. When n equals 3, IPO-N generates 86 tests for the triple (3-way) coverage criterion, so that 99.995% tests are reduced. In most conditions, the tests generated by IPO-N are much less than those required for the full coverage. It is usually unpractical to test all the combinations of all the parameters, and it is aimless and risky to select tests randomly. The results from Table 1 suggest that it may be feasible to select a proper n and generate the n-way test table to reduce the risk and cost of software testing.

We also compared our algorithms with some existing n-way test suite generation algorithms: AETG[2,3], GA and ACA[7]. When n equals 3, the number of tests generated by each algorithm is showed in Table 2, where the data of AETG, GA and ACA is collected from [7]. The experimental results also suggest that GA-N need to be optimized carefully, since its generated tests are always more than other algorithms in many cases. As shown in Table 2, IPO-N can generate the smallest test suites for some systems such as X_2, X_4, X_5 and X_6, and in other cases such as X_7 and X_8, the sizes of test suites generated by IPO-N are only a bit more than the smallest ones. These results seem to indicate that IPO-N is an effective algorithm, which performs well with respect to the size of generated test suites.

5 Conclusions

Much attention has by far been paid to the pair-wise testing, however in some safety-critical applications, testers may require the test suits satisfy the n-way coverage criterion. In this paper, we propose two different test generation algorithms based on combinatorial design approaches for the n-way combination testing, and suggest the algorithm IPO-N may be more feasible based on the empirical results.

As shown in Table 1, with the increase of n, the number of tests increases rapidly, and the time consumed for the test generation grows sharply. If the automatic test generators could not finish its work in acceptable time, testers would abandon the combinatorial design approaches. Therefore the algorithm's efficiency plays an important role in practice. We are searching novel algorithms for the n-way combination testing to improve the performance of test generators.

References

1. R. Mandl: Orthogonal Latin Squares: An Application of Experimental Design to Compiler. Testing Communications of the ACM. Oct. 1985, 28(10), 1054-1058.
2. D.M. Cohen, S.R. Dalal, M.L. Fredman: The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Trans. on Software Engineering. July 1997, 23(7)437-444.
3. D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton: The Combinatorial Design Approach to Automatic Test Generation. IEEE Software. Sep. 1996, 83-87.

4. Y. Lei, K.C. Tai: In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. Technical Report TR-2001-03. Dept. of Computer Science. North Carolina State Univ, Raleigh, North Carolina. Mar. 2001.
5. K.C. Tai, Y. Lei: A Test Generation Strategy for Pairwise Testing. IEEE Trans. on Software Engineering. Jan 2002, 28(1): 109-111.
6. Baowen Xu, Changhai Nie, Qunfeng Shi, Hong Lu: An Algorithm for Automatically Generating Black-box Test Cases. Journal of Electronics. Jan 2003, 20(1):74-78.
7. S. Toshiaki, T. Tatsuhiro, K. Tohru: Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing. 28th Computer Software and Conference, Hong Kong, China. Sep. 2004.
8. D. R. Kuhn and A. M. Gallo: Software Fault Interactions and Implications for Software Testing. IEEE Trans. On Software Engineering. June 2004, 30(6): 418-421

Automated Generation and Evaluation of Dataflow-Based Test Data for Object-Oriented Software

Norbert Oster

Friedrich-Alexander-University Erlangen-Nuremberg,
Department of Software Engineering,
Erlangen, Germany
`http://www11.informatik.uni-erlangen.de/`
`Norbert.Oster@informatik.uni-erlangen.de`

Abstract. In this research paper, an approach to fully automating the generation of test data for object-oriented programs fulfilling dataflow-based testing criteria and the subsequent evaluation of its fault-detection capability are presented. The underlying aim of the generation is twofold: to achieve a given dataflow coverage measure and to minimize the effort to reach this goal in terms of the number of test cases required. In order to solve the inherent conflict of this task, hybrid self-adaptive and multi-objective evolutionary algorithms are adopted. Our approach comprises the following steps: a preliminary activity provides support for the automatic instrumentation of source code in order to record the relevant dataflow information. Based on the insight gained hereby, test data sets are continuously enhanced towards the goals mentioned above. Afterwards, the generated test set is evaluated by means of mutation testing. Progress achieved so far in our ongoing project will be described in this paper.

Keywords: testing, dataflow, evolutionary algorithms, automated test data generation, object-oriented software, mutation testing.

1 Introduction

Today's software systems have reached a complexity where powerful software engineering tools are necessary in order to reasonably cope with their demands. One of the most time consuming activities during the development of software, especially in the case of safety critical systems, is testing - in particular when done according to control- or dataflow-oriented strategies. The main reason why testing requires a substantial amount of time and money is the fact that a considerable work has to be carried out by hand. Among those tasks is the identification of test cases for the purpose of fulfilling a given testing criterion. When this is done, the tests must be carried out. After completing the test case execution, the tester has to analyze the results of the runs twofold: first, one has to determine whether the desired criterion is sufficiently fulfilled and second, one must check whether the software behaves in compliance with the specification.

It is seldom possible to fully automate the latter job, unless a "golden version" is available against which the testing results can be compared, because their assessment requires rational human judgment. Running the defined test cases is facilitated by all major software engineering tools - that is what is usually called *test automation* today. Determining whether a test set fulfils a given testing criterion is also supported by most testing tools for different programming languages, but only very few of them provide support for criteria beyond basic control flow (typically statement and branch) coverage.

Assuming that there is such support for the criterion under consideration, the most ungrateful task is to enhance the test set if it does not yet achieve the desired coverage. This has still to be done by hand and only very little support is given here by any testing tool. Random test case generators might be helpful here at first glance, but they usually spawn a lot of superfluous data. This is due to the fact that they create many similar test cases covering the same paths many times while possibly not executing others at all. The problem that arises here is that all those test cases have to be validated individually. Therefore, the smaller the set of test cases fulfilling the testing goal, the lesser the effort spent on validation.

During the last few years, evolutionary algorithms have become popular in this context and several papers and PhD theses about software testing applications have been published (e.g. [1,2,3,4,5]). Our technique is new to our knowledge in different senses: our aim is to provide a means for *fully automating* the generation and evaluation of test cases for *object-oriented* programs, allowing for *complete* Java without any restrictions, focusing on *dataflow-oriented* testing strategies, such that they satisfy at least the following *two main objectives at the same time*: maximise the coverage achieved by the test set and minimise the number of necessary test cases.

The approach described in this work is based on distributed, hybrid, self-adaptive and multi-objective evolutionary algorithms, where the problem of test set generation is considered as an optimisation task. We implemented a tool that automatically instruments Java source code, such that all dataflow information relevant for evaluating a test case or a set of test cases (denoted as *test set* in the remainder of this paper) can be reconstructed from a trace generated during the run of each test case [6]. The information thus gained is used to determine the coverage achieved in terms of the actual number of covered entities, as required by the first objective; on the other hand, the second objective simply requires to evaluate the size of the test set. On the whole, those two values constitute the so-called fitness of a test set.

2 Dataflow-Based Testing

Dataflow-oriented testing is not really a new strategy and although a lot of progress has been made in this area, it is still not adopted in industrial software development practice due to the arguments given in section 1. In this work, only a short overview of the original criteria defined in [7] is given.

Dataflow-oriented testing is concerned with how information is processed during the execution of the software, more precisely how variables are accessed throughout the run of the program. First, accesses to variables are distinguished into *definitions* (so-called *def*), where variables are assigned a new value and *uses*, where the value of a variable is read. It is useful to further distinguish two classes of uses. In computational uses (*c-use*) the value of a variable is read in order to perform a computation. For example in "`x = y + z;`", there is a c-use of the variables `y` and `z` followed by a def of the variable `x`. If the value of a variable is used in order to come to a decision, then the variable is in a predicative use (*p-use*). The statement "`if (a > 7)`" contains a p-use of the variable `a`.

The goal of dataflow testing is to execute certain paths between each definition of each variable in the code and certain uses of that variable reached by that definition. The whole testing process is thus based on the control flow of the program that is annotated with the relevant dataflow information. Since a predicate decides upon the flow of control (i.e. which instruction to execute next), p-uses are associated to the edges starting from the control flow node to which the predicate is associated.

Before giving a short overview of the main criteria, some basic notions are introduced. A *sub-path* is a sequence of consecutive nodes in the control flow graph, each node representing a block of sequential statements. A *path* is thus a special sub-path starting with the entry node of the control flow graph of the system under test and ending with the exit node. A sub-path is said to be *def-clear* with respect to a certain variable, iff no definition of this variable occurs along the sub-path. A sub-path p starting with a def of a variable v is called *du-path* w.r.t. v if p is def-clear w.r.t. v except for the first node and v encounters either a c-use in the last node or a p-use along the last edge of p – originally[1], a du-path has been required to be loop-free or at most a simple loop[2]

Which classes of program paths have to be executed depends on the dataflow coverage criterion chosen. The weakest one in the family of dataflow-oriented coverage criteria, **all-defs**, requires to execute *at least one* def-clear sub-path between *every* definition of every variable and *at least one* reachable use of that variable.

The **all-p-uses**-criterion requires to execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable p-use of that variable. Since p-uses are associated to edges, this criterion subsumes the branch coverage criterion based on control flow. In analogy to all-p-uses, **all-c-uses** requires to execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable c-use of the respective variable.

As a definition of a variable must not necessarily reach any of its c-uses, a stronger criterion named **all-c-uses/some-p-uses** requires in such cases the

[1] Requirement has been relaxed, since some du-paths might be infeasible.

[2] All nodes are pairwise distinct, except for the first and the last node.

reaching of at least one p-use. Analogous considerations lead to the symmetrical criterion **all-p-uses/some-c-uses**.

The comprehensive **all-uses**-criterion requires to execute *at least one* def-clear sub-path from *every* definition of every variable to *every* reachable use of that variable. This criterion obviously subsumes all of those mentioned above.

All criteria described so far require *at least one* sub-path between defs and uses. Requiring *all* possible sub-paths would be too demanding in case that they contain loops that can be iterated a number of times not known in advance. Therefore, a less ambitious criterion, called **all-du-paths** was defined in [7] by restricting the above requirement from all sub-paths to all du-paths (i.e. loop-free or simple-loop). Unfortunately, this criterion does not necessarily subsume any of the above-mentioned criteria, e.g. in case that there exist executable sub-path(s), but no executable du-path between a definition and a corresponding use. The subsumption of all-uses by all-du-paths can be restored by relaxing the requirement of loop-freeness if no loop-free du-path can be executed, as proposed by [8].

3 Evolutionary Algorithms

The main functionality of evolutionary algorithms is based on the principles of Darwinian evolution and the theory of survival of the fittest. Invented by John H. Holland and Ingo Rechenberg in the mid 70's, they are well known powerful search and optimisation techniques and as such applied in many different areas [9,10]. The progress of evolutionary engines in optimisation is achieved performing the following operations:

1. First, a random initial population is generated, where each of the individuals of this population represents a possible solution to the optimisation problem.
2. *Evaluation:* Each individual is evaluated by means of a fitness function, where its value is an indicator of how good (fit) this individual is with respect to its ability to solve the given problem. Usually, the fitness of the global optimum is not known in advance, but the relative comparison of fitness values among the individuals is sufficient to apply this optimisation technique.
3. A new generation is constructed by repeatedly applying the following two steps:
   - *Selection:* Two individuals are selected from the old generation according to their fitness, i.e. the higher the fitness, the higher the probability that these individuals are selected. Different selection criteria exist [9].
   - *Crossover:* The individuals thus selected are merged forming two new individuals – again, different crossover strategies can be applied, depending on the encoding of each individual and the specific problem. The new "children" are inserted into the population of the new generation.
4. *Mutation:* Each individual of the new population is mutated, i.e. slightly modified with a certain probability. The goal of this step is to introduce

new values into the genetic material predefined during the setup of the first population. Otherwise, the search would be limited to combinations of those initial values only.
5. Now the next generation passes through all the steps above starting with the evaluation phase.
6. The whole optimisation process can be stopped as soon as either the optimal solution is found or a predetermined number of generations has been evaluated, thus always giving at least a sub-optimal solution.

This kind of optimisation is superior to classical search/optimisation algorithms since it is scanning a broader part of the search space at the same time, allowing inferior solutions to be reconsidered later on as well, thus not becoming stuck in local optima, as e.g. hillclimbers.

Evolutionary algorithms were enhanced in different ways. Since they decisively depend on several parameters, e.g. the probability for crossover or the selection strategy, a self-adaptation during the optimisation can guide the search in reasonable ways. One possibility for self-adaptation is to define in advance certain modifications of the parameters, e.g. increasing the mutation probability by a fixed increment, and apply them after completing the creation of each new generation. Since this requires very detailed knowledge of the specific problem and of the optimisation technique, an easier way to integrate self-adaptation is to let the algorithm choose the best parameters itself. This is done by including the parameters into the genetic material of the solution thus being evolved as part of the population. The idea is that better individuals were developed using better parameters. Since the generation of the new population is also subject to randomness, it is not guaranteed that good solutions are kept until the end of the process. To avoid this problem, elitism can be applied, i.e. a certain number of the best individuals of the current population is saved without modification to the new generation.

As mentioned earlier, the generation of optimal test sets belongs to the class of multiobjective problems with conflicting goals (maximal coverage with minimal effort). In order to apply evolutionary techniques to this field, the fitness measure can be evaluated by aggregating the different objective measures weighted according to their importance into one overall fitness value. Because this approach suffers from crucial deficits (e.g. how to define the weights), other more powerful variants have been developed, as e.g. the *Vector Evaluated Genetic Algorithm (VEGA)*, the *Niched Pareto Genetic Algorithm* and the *Non-dominated Sorting Genetic Algorithm (NSGA)*, as described and compared in [11]. On the basis of this comparison, the latter one was used in the approach presented here in addition to the aggregation technique.

4 Test Set Generation with Evolutionary Algorithms

In order to generate test sets by means of evolutionary algorithms, we have to choose a performant encoding of the population and a good fitness measure first.

In the approach of global optimisation, the individuals of a population are taken to represent test sets, each consisting of a list of test cases, where each test case is a sequence of primitive data (the "arguments"). The length of the sequence and the type and range of each argument can be determined by the tester in order to adapt the generation to different systems under test. For the execution, the test cases are either directly passed to the system under test (e.g. in the case of complete Java-applications) or to a driver, decoding the data as required and invoking the module under test.

As already mentioned, the multiobjective fitness measure is evaluated on the basis of both the testing coverage achieved and the inverse size of the test set. The former value is determined as the number of distinct def/use-pairs actually covered. The latter value is the reciprocal of the number of test cases. In order to avoid one value dominating the other, both of them are subjected to a preliminary normalisation.

4.1 Test Set Generation

The first step of the evolutionary system is the generation of an initial population of a given size, chosen independently of the problem but influencing the speed and quality of the optimisation. The population consists of many different test sets, where each test set is built up with a randomly chosen number of test cases. The latter ones are created by random according to their specification such that they represent legal inputs for the system under test or its driver respectively.

After having completed the first generation, each test set is individually evaluated with respect to the two objectives coverage/size. The number of distinct and relevant dataflow pairs covered by the whole test set is calculated by dynamic analysis of test case execution [6], as described in section 4.2.

According to both fitness values two test sets are selected for reproduction and "mated", i.e. crossed over, thus yielding the children for the next generation. Mating is done by "exchanging" test cases between the parent test sets, such that each child contains test cases from at least one parent test set, preferably from both. Since the crossover is also done by random, it might happen that a child is identical with one of the parents, although this is very unlikely. This "exchange" is done in a way such that the parents are not modified, but the children gather the complete information of each test case, i.e. not only the test data but also the coverage known from the last test case execution. This is necessary in order to avoid executing test cases over and over again although they were not modified, thus saving a lot of execution time. The process of selection and crossover is repeated in order to build the next generation.

After having created a new test set, it may be subjected to a random mutation. This is done by either adding a new test case to the set, removing one from the set or modifying one of the existing test cases. Again, since the whole population is sharing data as much as possible, each mutation of a test case is done by creating a copy of the old one and applying some mutation operator to the copy only.

As a matter of course, mutated test cases must be executed in order to gather their coverage whereas non-modified test cases need not be re-executed. Because mutations are usually done with low probability (this was heuristically identified as better than high mutation probabilities), the number of newly generated or modified test cases is low compared to the size of the population. Therefore, the fitness evaluation of the next generation takes evidently less time than for the initial population, where all test cases had to be executed.

4.2 Dynamic Analysis of a Test Case Execution

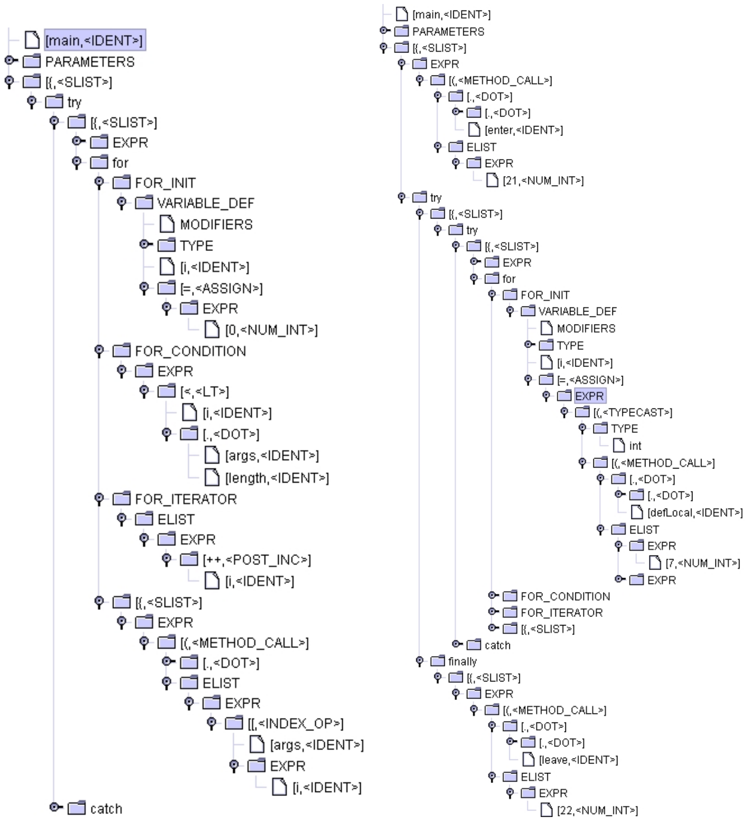
The most important fitness contribution to the generation of test sets is the number of dataflow pairs covered by each of their test cases. In order to determine this fitness value, a dynamic test case execution analyzer was implemented based on the ANTLR parser generator and translator [12]. By means of this tool, an abstract syntax tree (AST) is built and then processed by the instrumentation.

As a first step the system under test (in our case Java source code) is instrumented by inserting calls to a corresponding external logging system at all relevant sites in the program. During the instrumentation phase, the static information concerning dataflow and controlflow is recorded into a so called instrumentation log. This log contains an identifier for each relevant event that might occur during execution and additional information specific to the event under consideration. The use of a local variable occurring within a method e.g. is recorded as an event annotated with the type of the variable and with the exact location containing class name, method name, source file and position within the original source file. Because dataflow is based on controlflow, the information required to reconstruct the flow of control during the execution is also logged: e.g. whenever a method is invoked, the calling site as well as the entering and leaving of the method are also instrumented for logging. The same holds for conditions, the evaluation of which is wrapped by a start of predicate event and an end of predicate event explicitly, since the predicate might have a more complicated substructure, e.g. it might contain calls to other methods.

```
class OutputParameters {
 public static void main(String[] args) {
  try {
   System.out.println("Parameter:");
   for (int i=0; i < args.length; i++) {
    System.out.println(args[i]);
   }
  } catch (Exception e) {
  }
 }
}
```

Fig. 1. Original sample program

Consider the program example in figure 1. The main sections of interest of the corresponding abstract syntax trees are shown in figure 2. Figure 2(a) represents the AST as generated from the original source code given in figure 1, while figure 2(b) shows the AST of the instrumented code.



(a) original source code (b) instrumented source code

Fig. 2. Abstract Syntax Trees (AST)

The fully instrumented code as resulting from the automatic tool is shown pretty-printed in figure 3. In order to trace dataflow correctly in different instances of a class, each instrumented class contains an instance identifier, since each object field access belongs unambiguously to one object and *not* to the variable(s) referencing the object. By means of reference lists, a unique identifier is also assigned at runtime to instances of classes that were used but not instrumented.

The corresponding instrumentation log contains among others the entries given in table 1. According to it, an event with id 5 is a use of the static variable

```
class OutputParameters implements rt.InstanceId{
 public final int ___instanceId=rt.DULog.getNewInstanceId(0);
 public final int ___getInstanceId(){return ___instanceId;}
 {rt.DULog.enterInit(1);
  try{___doZeroInit();}
  finally{rt.DULog.leaveInit(2);}
 }
 public static void main(String[] args){
  rt.DULog.enter(21);
  try{
   try{
    ((java.io.PrintStream)rt.DULog.useStatic(5,System.out))
     .println((java.lang.String)rt.DULog.cp(6,"Parameter:"));

    for(int i=(int)rt.DULog.defLocal(7,0);
      rt.DULog.predResult(12,rt.DULog.newPredicate(11),(int)rt.DULog.useLocal(8,i)
       < rt.DULog.useArrayLength(10,(java.lang.String[])rt.DULog.useLocal(9,args)));
      rt.DULog.useDefLocal(13,i++)){(
     (java.io.PrintStream)rt.DULog.useStatic(14,System.out))
      .println((java.lang.String)rt.DULog.cp(18,(java.lang.String)
       rt.DULog.useArray(17,(java.lang.String[])
        rt.DULog.useLocal(15,args),
        rt.DULog.useLocal(16,i))));
    }
   } catch(Exception e) {
    rt.DULog.exceptHandlerCall(20);
    rt.DULog.defLocal(19);
   }
  } finally {
   rt.DULog.leave(22);
  }
 }
 protected void ___doZeroInit() {}
 {rt.DULog.initCompleted(2);}
}
```

Fig. 3. Instrumented sample program

`System.out` in class `OutputParameters` in line `4` at column `31`. The use with a subsequent def of the loop counter `i` is denoted by id `13`. Whenever id `18` has been logged at runtime, the call of the method `System.out.println` in line `6` occurred. Figure 2(b) shows explicitly the calls to the logging system to log the entry (node *method call* with method name *enter* and identifier `21`) of the `main` method.

Executing the instrumented software with a test case provides a compact log of data and control flow information for the specific test case. This information represents the trace of the executed path as well as relevant dataflow events covered by the test case. The trace is minimal in the sense that only the event identifier stored during the instrumentation (e.g. `21` for entering method `main`, see table 1 and figure 2(b)) is logged together with additional data not available until execution (e.g. in the case of a predicate evaluation, the dynamic data represents the result of the predicate). Access to non-static variables is specified by means of the instance identifier in order to distinguish potentially multiple instances of a variable.

Matching the instrumentation log and the execution log provides the information about witch def/use-pairs were covered along which sub-paths connecting them. The acquired data is sufficient for a fine-grained analysis of the dataflow covered by an executed test case. It can be simplified by folding equivalent

Table 1. Excerpt from the instrumentation log

ID	event	event specification	row	column
5	useStatic	public static final java.io.PrintStream java.lang.System.out	4	31
7	defLocal	int OutputParameters.main([Ljava.lang.String;).i	5	0
13	useDefLocal	int OutputParameters.main([Ljava.lang.String;).i	5	56
17	useArray	[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args	6	52
18	cp	public void java.io.PrintStream.println(java.lang.String)	6	51
21	enter	public static void OutputParameters.main(java.lang.String[])	2	0

def/use-pairs, e.g. concerning different instances of a variable, thus giving the dataflow information at any desired granularity.

The coverage of a whole test set is determined on the basis of the coverage of each test case. Care must be taken in order not to count the same def/use-pair covered by different test cases of the same test set more than once, otherwise the fitness measure will be blurred, thus degrading the performance of the optimisation.

5 Experimental Results

Implemented in Java and run on simple PCs, the tool outperformed our expectations with regard to speed and quality of the generation and optimisation of test cases as well as memory usage. In general, the test case execution and the matching of static and dynamic dataflow information is the most resource consuming part of the optimisation. For this reason, the tool was implemented such as to run the instrumented system under test in parallel on a number of networked PCs. As the execution was performed in the background, while the PCs were in normal use, the observed execution times are insignificant.

Table 2 shows the results of applying the multiobjective aggregation strategy with self-adaptation of genetic parameters. Hereby, the weights used for aggre-

Table 2. Results of multiobjective aggregation strategy

ID	Project	Size LOC / Classes	DU-pairs found	DU-pairs covered	Size of test set	Number of generations
1	Hanoi	38 / 1	42	42	2	9.2
2	Dijkstra	159 / 2	213	213	2	29
3	JDK sort	82 / 1	315	315	2	117
4	Huffman	318 / 2	368	367.6	3.6	155.8

gation of fitness values were chosen in favour of coverage maximisation. The column "DU-pairs found" shows the maximum number of pairs identified within any run without selection pressure on the number of test cases. All other values are averaged over 5 runs each, where "Number of generations" represents the earliest generation where the coverage/test set size - ratio stabilised. This kind of optimisation does not strictly require a static analysis of the dataflow, which is part of ongoing work and might be used to provide the ideal stopping rule. Instead, each optimization was terminated after 1000 generations.

In addition to multiobjective aggregation, which offers a compromise between coverage and validation effort, the implemented tool is also able to generate and optimise with respect to the following relation (so-called *domination*): a test set t_1 dominates another test set t_2 w.r.t. both coverage c and test set size s iff $c(t_1) \geq c(t_2)$ and $s(t_1) \leq s(t_2)$ and either $c(t_1) > c(t_2)$ or $s(t_1) < s(t_2)$ holds. The set of all optimal (i.e. dominating but not dominated) test sets represents the so-called Pareto-front. It can be generated by the *Nondominated Sorting Genetic Algorithm*, one of the most promising algorithms [11] of this kind, which was adopted for our tool and applied to a variant of the JDK math package with additional functionality (553 LOC in 3 classes).

Figure 4 shows the Pareto-front for the all-uses-criterion as generated by our tool. Each dot represents one test set in the population, most of them already lined up along the front. The tool could identify 1.373 def/use-pairs and cover them with one test set consisting of 54 test cases within 1000 generations. These

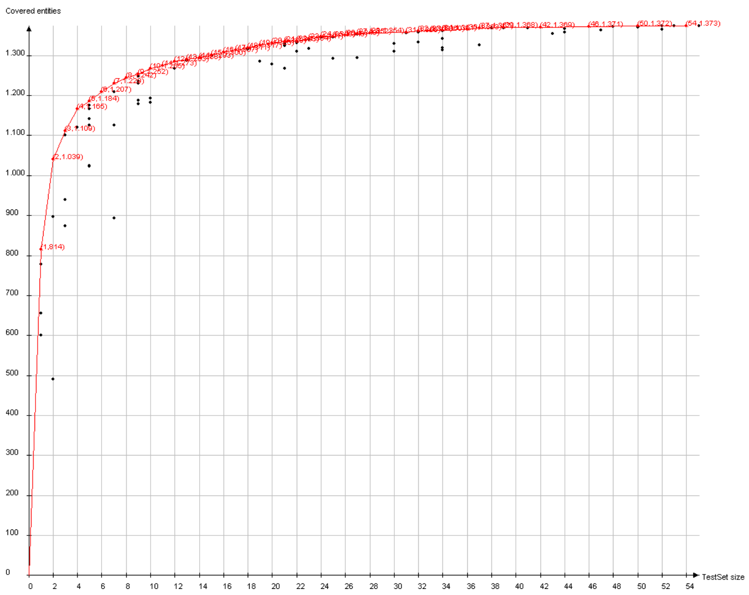


Fig. 4. Pareto-optimal front for all-uses

results were counter-checked by two additional runs, where the first one was dedicated to covering all c-uses and the second one for all p-uses. The tool proposed 18 test cases in order to cover 588 c-uses and 44 test cases for 799 p-uses. Remember that each predicative use of a variable e.g. within an "if"-statement is counted for each possible outcome of the decision in part, such that the all-p-uses criterion subsumes the branch coverage criterion. Applied to satisfy the all-defs-criterion only, the tool is able to stabilise the population quite fast to the front described by table 3.

Table 3. Pareto-front for all-defs

test set size	1	2	3	4
covered defs	191	198	201	202

Astonishing results were achieved for the same system under test with respect to a slightly modified all-uses-criterion, requiring – in addition to the classical criterion used above – to test all possible combinations of predicate outcomes in case of nested predicates. Within 850 generations, the Pareto-front showed 23.724 DU-pairs covered by 70 test cases. This astonishingly high number of def/use-pairs is due to the fact that the example code contains reciprocal recursion and a deeply nested controlflow structure. This implies a particularly high number of p-uses, because in case of the modified all-uses-criterion a p-use is characterized by the complete stack of predicates rather than by the innermost active condition as required by the classical all-uses-criterion.

In order to evaluate the fault-detecting capability of the generated test sets, we extended our tool with a variant of mutation testing [13]. The key idea is to insert small modifications (e.g. replacing "$\leq$" by "$<$") into the original system under test, thus generating a set of slightly different *mutant*s. Then, each mutant is run with each generated test case. If during this execution, the mutant behaves differently for the same test case than the original program, the mutant is said to be *killed.* The ratio of killed mutants, the so-called *mutation score* is an indicator for the fault detection capability (and thus the "quality") of the test set under consideration.

By means of this mutation analysis, we compared this indicator for test sets generated to satisfy the branch coverage criterion with the score for test sets generated by enhancing the corresponding branch-covering test set with additional test cases necessary to cover as many def/use-pairs as possible. Applied to our projects[3], we observed the results presented in table 4.

Although the mutation scores for dataflow-based test sets do not seem to be significantly higher than those for branch-coverage, it must be considered that e.g. in the case of the `BigFloat`-project 63 additional mutants, denoting 63 potential software faults detected, could be killed with only 32 additional dataflow-related test cases achieved at the same cost as for branch coverage, due

[3] Slightly modified compared to table 2.

Table 4. Evaluation of generated test cases by mutation analyis

ID	Project	TNM	NTC-B	C-B	MS-B	NTC-AU	C-AU	MS-AU
1	BigFloat	1528	5	144	69.18	37	1384	73.30
2	Dijkstra	220	1	26	70.45	3	168	71.82
3	Hanoi	227	2	4	74.89	3	42	77.53
4	Huffman	623	3	61	74.32	4	353	84.27
5	JDK sort	852	1	37	61.50	3	315	64.79

TNM:	total number of generated mutants (including possibly equivalent ones)
NTC-B:	number of test cases generated for 100% branch coverage
C-B:	number of branches covered
MS-B:	mutation score (in %) achieved by test cases generated for branch coverage
NTC-AU:	number of test cases generated for all-uses coverage
C-AU:	number of def/use-pairs covered
MS-AU:	mutation score (in %) achieved by test cases generated for all-uses

to their fully automated generation. This benefit is even more obvious for the `Huffman`-example with 62 mutants killed by one additional dataflow-specific test case.

The mutations scores of table 4 are in a sense pessimistic, in that the total number of mutants also includes those that are semantically equivalent, i.e. the modified software behaves the same as the original for any possible input. In the case of the `Dijkstra`-project, only 4 of the mutants remaining unkilled by the dataflow-based test set turned out to be truly undiscovered while 58 are equivalent to the original. Thus, with respect to the evaluation of the fault detecting capability of the generated test sets, an automated support for distinguishing equivalent and non-equivalent mutants would be of great benefit, as the number of automatically generated mutants can be significantly high. As part of the ongoing research, we estimate this ratio by means of statistical testing.

6 Conclusion and Outlook

This article has presented an approach supporting the automatic generation of test data for object-oriented programs with respect to dataflow-based criteria. The technique proposed is based on evolutionary algorithms aiming at maximizing dataflow coverage and minimizing the size of the test set, and thus the effort for validation. Different search and optimization methods have been implemented for several dataflow criteria and demonstrated by means of practical examples. The preliminary results achieved confirm the efficiency and ease of use of the approach adopted.

Nevertheless, both results and performance can be enhanced by extending the evolutionary algorithms presented. A possible *hybridisation* is to interleave the global optimisation phase with a local test case generation. During the global optimisation phase the population is represented by test sets and the technique aims at finding the test set with the lowest number of test cases achieving the desired coverage. This might lead to some def/use-pairs remaining uncovered because either the trade-off between test set size and coverage prevented the necessary test cases to be accepted in a new generation, or the paths to be executed in order to cover those pairs were too hard to find by the quasi-random optimisation technique. Therefore, a local optimisation phase may be applied to generate test cases dedicated to covering the missing dataflow pairs. In the ongoing research, techniques allowing to identify the missed paths and to generate appropriate test cases are under development.

Reviewing the mutants remaining unkilled by the dataflow-based test set, it turned out that most of them can only be incidentally killed by dataflow testing - as those criteria consider the flow of data through the program rather than the values actually processed. In order to augment the quality of such optimal test sets, as generated by our technique, we propose to enhance the test sets by additional test cases, e.g. generated with respect to equivalence class partitioning coverage - a topic that is as well part of our ongoing research.

References

1. O'Sullivan, M., Vössner, S., Wegener, J.: Testing temporal correctness of real-time systems - a new approach using genetic algorithms and cluster analysis. In: EuroSTAR98 Software Testing Analysis & Review. Number 6 in EuroSTAR, Munich Park Hilton (1998) 397–418
2. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering. Number 16 in ICSE, IEEE (1994) 191–200
3. Michael, C.C., McGraw, G.: Automated software test data generation for complex programs. In: Automated Software Engineering. Thirteenth IEEE Conference on Automated Software Engineering, IEEE (1998) 136–146
4. Baresel, A.: Automatisierung von Strukturtests mit evolutionären Algorithmen. Diplomarbeit, Lehr- und Forschungsgebiet Softwaretechnik, Humboldt-Universität Berlin, Berlin (2000)
5. Harman, M., Hu, L., Hierons, R., Baresel, A., Sthamer, H.: Improving evolutionary testing by flag removal. In: Genetic and Evolutionary Computation Conference (GECCO 2002). (2002)
6. Oster, N., Dorn, R.D.: A data flow approach to testing object-oriented java-programs. In Spitzer, C., Schmocker, U., Dang, V.N., eds.: Probabilistic Safety Assessment and Management (PSAM7/ESREL'04). Volume 2., Berlin, Springer-Verlag London (2004) 1114–1119
7. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **SE-11** (1985) 367–375
8. Horgan, J.R., London, S.: Data flow coverage and the C language. In: Proceedings of the symposium on Testing, Analysis and Verification, ACM, ACM Press (1991) 87–97
9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989)
10. Wilke, P., Gröbner, M., Oster, N.: A hybrid genetic algorithm for school timetabling. In McKay, B., Slaney, J., eds.: AI 2002: Advances in Artificial Intelligence. Volume 2557 / 2002 of Lecture Notes in Computer Science., Heidelberg, Springer-Verlag (2002) 455–464

11. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms – a comparative case study. Technical report, Swiss Federal Institute of Technology Zurich, Computer Engineering and Communication Networks Laboratory (TIK), Gloriastrasse 35, CH-8092 Zurich, Switzerland (1998)
12. Parr, T.: ANTLR, ANother Tool for Language Recognition. (http://www.antlr.org/).
13. Offutt, J., Ma, Y., Kwon, Y.: An experimental mutation system for java. Proceedings of the Workshop on Empirical Research in Software Testing / ACM SIGSOFT Software Engineering Notes **29** (2004)

Automated Model-Based Testing of χ Simulation Models with TorX [⋆]

Michiel van Osch

Eindhoven University of Technology,
Department of Mathematics and Computer Science,
5600 MB, Eindhoven, The Netherlands
`m.p.w.j.van.osch@tue.nl`

Abstract. Simulation models are used for performance optimization and validation of embedded systems. However, these models are usually not validated in a structural, formal, way. This paper describes a method for testing a χ-model using the model-based test-tool TorX. The method is explained by using a simple example. After that, we describe the results of a case study performed on a simulation model of an industrial system.

1 Introduction

In the embedded systems industry (e.g. aerospace, railway, and automotive industry), software development is complex, expensive, and systems need to be of high quality. In this industry, simulation tools (e.g. MATLAB [1], Simulink and LabVIEW [2]) are often used in the early stages of system development process. A model of the system or a specific component is made in a high-level programming language and executed using the tool. With these simulations insight is gained in the behavior of components. Based on the results of the simulations the design is improved. When the real system is being implemented, the correctness of the implementation can be validated by comparing it to the behavior of the simulations again. Thus, simulation can be used for validation of the design and for validation of the implementation. However, the simulation model itself is usually not thoroughly validated. When the simulation "looks reasonable" or "looks to behave according to the documentation" the simulation is considered correct. Mistakes introduced in the simulation model influence the quality of the design and of the implementation. Mistakes in the design that remain in the simulation model remain in the final implementation also.

We propose to use model-based testing for the validation of simulation models. In model-based testing the test-cases are automatically generated from the model. These test-cases are automatically performed on the implementation. In this way much more tests can be performed compared to manual testing, and

⋆ This work has been carried out as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

this can increase the quality of the implementation. Several tools have been developed for model-based testing so far (e.g. AsmL [3], TGV [4] and TCBeans [5]). When this approach is used on simulation models it increases the quality of the simulation model. And by testing the simulation model the quality of the design and the implementation increases also.

In this paper we show how the model-based test-tool TorX [6, 7] can be used to test χ simulation-models [8, 9]. To make testing possible, a connection between the TorX test-tool and the χ-simulator is made. Furthermore, we make the χ-model itself suitable for communication with our test-tool without changing (the part of) the model we want to test. To demonstrate the benefits of model-based testing of simulation models we apply this approach within an industrial case study. In this case study we test a simulation model of a wafer-scanner machine. A wafer-scanner is used in the chip manufacturing industry. It burns an image of the chip layout on a silicon disc. Testing of a simulation model of an industrial component revealed mistakes in the model as well as a bug in the test-tool. The mistakes in the model that was tested, were fixed, as well as the bug found in the test-tool.

The rest of this paper is organized as follows: In Section 2 the basics of model-based testing are explained together with the TorX test-tool; in Section 3 the χ-language is explained taking the alternating bit protocol as an example; in Section 4 this protocol is tested using TorX; in Section 5 the results of testing a simulation model of an industrial component are explained; and in Section 6 some conclusions are presented.

2 Model-Based Testing with TorX

The TorX test tool was developed within the Côte-de-Resyste project [10], a collaboration between the University of Twente, Eindhoven University, Philips Research Laboratories and Lucent Technologies. It can be used for testing hardware and software.

Several case-studies have been performed with TorX (e.g. a highway tolling system [11] and a communication protocol [12]). TorX makes use of the *ioco* (input-output conformance) relation [6, 13]. The ioco theory states that a specification and an implementation are input-output conform if for every input in the specification the resulting output given by the implementation (after processing this input) is present in the specification also.

We illustrate the ioco theory by an example. Consider the two labelled transition systems in Fig. 1. *Model 1* specifies a system which accepts a coin as input, after which it nondeterministically returns tea or a coin. After the output of tea or a coin the automaton returns to the initial state. *Model 2* specifies a system which accepts every coin and always produces tea after a coin has been inserted. After output of tea this automaton also returns to its initial state.

In this example *Model 2* is input-output conform *Model 1*. Starting from the initial state a coin can be inserted in both transition systems. After a coin has been inserted, *Model 2* produces tea. In *Model 1* it is also possible to produce tea so *Model 2* behaves correctly.

The other way around input-output conformance does not hold: *Model 1* is not input-output conform with *Model 2*. After inserting a coin in *Model 1* the coin can be returned. According to *Model 2* this should not be possible. *Model 1* contains behavior that was not specified in *Model 2*. The intuitive reasoning is that you can only detect "unexpected" behavior of the implementation as errors.

In ioco testing, test-cases are built from the specification by recursively selecting an input, observing output, or giving a verdict pass or fail. From *Model 2* we could for example generate the following test-cases:

Note that the output action "!coin" does not occur (as transition) in *Model 2*, but still has to be considered as a possible output action of *Model 2*. The test-cases consist of applying input "?coin", then observing an output, and then

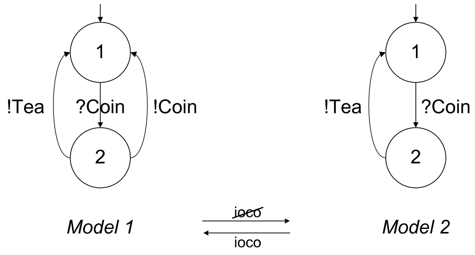


Fig. 1. Two Input-Output Transition Systems of a Tea Machine

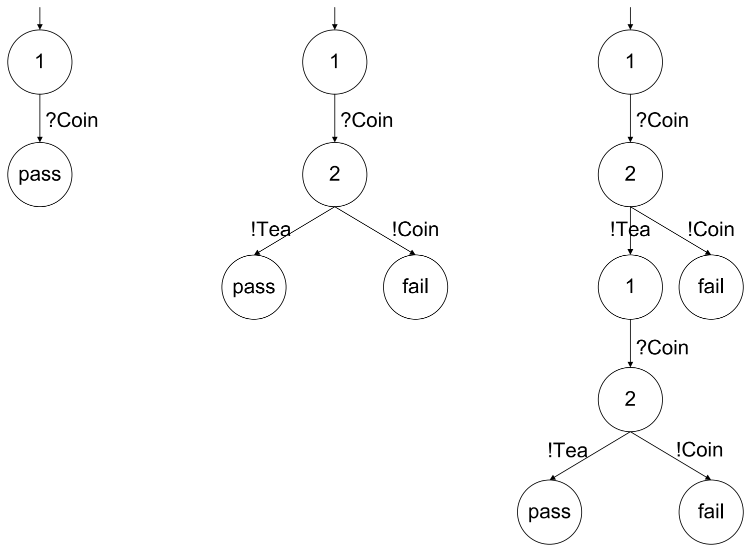


Fig. 2. Example Test-Cases

drawing a conclusion or continue with a new input. As long as we did not observe output "!coin" from an implementation that we are testing (e.g. *Model 1*) a test-case will end with a "pass" verdict. As soon as we observe output "!coin" we can stop testing, conclude verdict "fail", and non-conformance of *Model 1* with respect to *Model 2*.

The TorX tool supports testing of discrete event systems according to this conformance relation. An input generated from a specification is applied to the system under test, and the output from this system is compared to the specified output. When it is possible to perform the observed output transition in the specification, the test is passed. The architecture is depicted in Fig. 3.

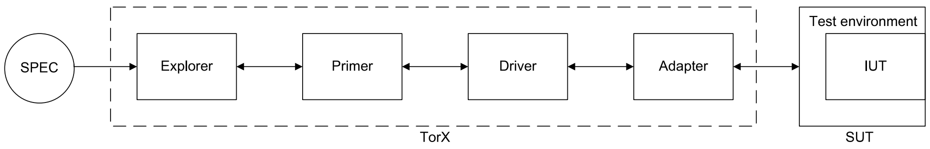


Fig. 3. Torx Architecture

SPEC: The formal model, from which the test-cases are generated. TorX is able to extract test cases from several specification langauges, e.g. TROJKA [14] (a dialect of the PROMELA language) and LOTOS. It is also possible to implement an automaton or labelled transition system in C and generate test cases from these models.

Explorer: This component offers functionality to explore the transition graph of a specification. It is language specific. For a given state it provides the set of transitions (input and output) from that state. E.g. an explorer is available that is based on the *OPEN/CAESAR* [15] interface, and one that is based on the *SPIN* [16] interface.

Primer: This component implements the test derivation algorithm. It also provides functionality to offer input stimuli to the implementation under test and to check output observations from the implementation under test. Test selection is currently done at random (with random seeds) or using test purposes [17].

Driver: This component controls the progress of the testing process. It decides whether to do an input action or to observe possible output from the implementation. It uses the primer to select an input action from the specification and the adapter to observe the actual output from the implementation.

Adapter: This component is the connection between the test-tool and the system under test. It is responsible for translating input transitions from the specification to readable inputs for the implementation, and for translating outputs received from the implementation back to output actions. These output actions are then matched to the output transitions in the specification.

SUT: The System Under Test (SUT) consists of the actual implementation under test (IUT), together with the test environment (e.g. drivers, stubs, hardware and operating system). An input from the adapter is passed on to the implementation under test. The output from the SUT is communicated back to the adapter.

The initial purpose of the tool was to test real implementations automatically. In the rest of this paper we show it can also be used to test simulation models.

3 Modelling with χ

Simulation is used for rapid prototyping, to gain insight in the correctness of the design, to build understanding between developers and customers, for throughput evaluation, or for conformance validation between design and implementation. The χ-language is used for simulation of manufacturing networks and components of embedded systems (hardware and software), controller synthesis of chemical analyzers, and scheduling problems. E.g. a wafer-scanner is modelled with a pre-defined probability of successful exposure. A failure to expose a wafer leads to a decrease in throughput. By simulations the variation in throughput can be analyzed. Numerous case studies have been performed using χ, e.g. for production systems [18, 19].

We use the alternating bit protocol as an example to illustrate how to model a system in χ. We deliberately introduce a mistake in the model presented in this section. In the next section we will explain how to test this model using TorX, and present how to fix the mistake in the model.

The alternating bit protocol allows communication over lossy communication mediums. Lossy means a message might get corrupted or lost on the medium. A transmitter sends a message over a lossy medium to a receiver. If the message arrives uncorrupted, the receiver sends an acknowledgement over a second lossy medium. If the message arrives corrupted the receiver sends an error message over the second medium. The acknowledgement or error message might get corrupted also. If the transmitter does not receive the proper acknowledgment, the original message is sent over the first medium again. This process is repeated until the transmitter receives the correct acknowledgment. After that, a new message is sent.

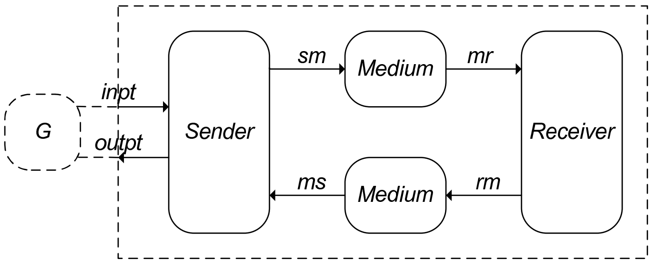


Fig. 4. The Alternating Bit Protocol

The protocol is depicted in Fig. 4. The transmitter, mediums, and receiver are modelled as processes; process *Sender*, two processes *Medium* and process *Receiver* respectively. Process G is the environment of the protocol, it generates the messages and waits for the acknowledgements of successful transmission.

Process *Sender* receives a message from an external process and as long as it does not receive an acknowledgement it keeps on retransmitting it. Information is exchanged between processes via synchronous channels. In this case input channel *inp*? and output channel *outp*!. After the declaration of local variables *msg* and *a*, the body of this process is enclosed in an infinite repetition. Within this repetition a selective repetition is used to keep on retransmitting message as long as no acknowledgement *ack*0 or *ack*1 is received.

```
proc $Sender(sout\colon !\text{string}, sin\colon ?\text{string}, inp\colon ?\text{string}, outp\colon !\text{string}) =$
$[\![\, msg : \text{string}, a : \text{string}$
$|\, *[\, \mathsf{true} \longrightarrow inp?msg$
            $; *[\, a \notin \{"ack0", "ack1"\} \longrightarrow sout!msg$
                                           $; sin?a$                    $(\chi\text{-}3.1)$
              $]$
            $; outp!a$
   $]$
$]\!]$
```

Process *Receiver* receives a message uncorrupted (*msg*0 and *msg*1) or corrupted (c). This process uses a selection statement in which depending on the message, a particular acknowledgement (*ack*0, *ack*1 or *nack*) is selected.

```
proc $Receiver(rin\colon ?\text{string}, sout\colon !\text{string}) =$
$[\![\, msg : \text{string}$
$|\, *[\, \mathsf{true} \longrightarrow rin?msg$
            $; [\, msg = "c"\qquad \longrightarrow rout!"nack"$
             $[\!]\, msg = "msg0" \longrightarrow rout!"ack0"$          $(\chi\text{-}3.2)$
             $[\!]\, msg = "msg1" \longrightarrow rout!"ack1"$
             $]$
   $]$
$]\!]$
```

Process *Medium* is the communication medium between *Sender* and *Receiver*. Messages on the *Medium* become corrupted at random, by assigning either value 0 or 1 to a variable s according to a uniform distribution.

```
proc $Medium(cin\colon ?\text{string}, cout\colon !\text{string}) =$
$[\![\, msg : \text{string}, u :\rightarrow \text{nat}, s : \text{nat}$
$|\, u := uniform(0, 2)$
$;\, *[\,\text{true} \longrightarrow cin?msg$
            $;\, s := sample(u)$
            $;\, [\, s = 0 \longrightarrow cout!msg$                    $(\chi\text{-}3.3)$
             $[\!]\, s = 1 \longrightarrow cout!\text{"}c\text{"}$
             $]$
   $]$
$]\!]$
```

Process G is used to generate messages to be sent by process *Sender* to process *Medium*. After the acknowledgement is received from the process *Sender* it generates another message (*msg1*). After the acknowledgement for the second message is received the first message is sent again. Two messages are needed to be sent between the processes because we want to simulate the possibility to send different messages by alternating between them.

```
proc $G(inp\colon !\text{string}, outp\colon ?\text{string}) =$
$[\![\, a : \text{string}$
$|\, *[\,\text{true} \longrightarrow inp!\text{"}msg0\text{"}$
            $;\, outp?a$                                        $(\chi\text{-}3.4)$
            $;\, inp!\text{"}msg1\text{"}$
            $;\, outp?a$
   $]$
$]\!]$
```

The processes need to be clustered in a network. In this network the connections between processes are specified. All processes are executed in parallel and are instantiated as an experiment.

```
clus $abp() =$
$[\![\, sm\colon -\text{string}, ms\colon -\text{string}, mr\colon -\text{string}, rm\colon -\text{string}$
$,\, inpt\colon -\text{string}, outpt\colon -\text{string}$
$|\, G(inpt, outpt) \parallel Sender(sm, ms, inpt, outpt) \parallel Medium(sm, mr)$      $(\chi\text{-}3.5)$
$\parallel Medium(rm, ms) \parallel Receiver(mr, rm)$
$]\!]$
```

```
xper() $= [\![ abp() ]\!]$                                       $(\chi\text{-}3.6)$
```

After the processes have been instantiated the χ-model can be compiled and executed. For this model the χ compiler does not find an error and an executable simulation is created. However, that does not mean the model is correct.

4 Model-Based Testing of χ-Models

The simulation model presented in the previous section contains a mistake that might not be spotted in one glance. Usually, only when the simulation model appears to run incorrectly a designer starts debugging the model. He might add extra debug code to output additional information on the state of the simulation at particular points. He might study the code itself, which is time consuming and does not guarantee success either.

We propose to use TorX for automated model-based testing of this model. To be able to test a simulation model, four steps have to be taken:

1. The model for generating test-cases needs to be built if this model does not exist already. Once this model has been made it can be re-used or adapted for any simulation model with similar behavior.
2. Both models, the simulation model and the model used for test generation need to be *open*, i.e. external channels need to be observable for the test tool.
3. The connection between the test tool and the simulator needs to be built. An adapter needs to be implemented which can be used in testing χ models. We have adapted an existing adapter, written in Python, for this purpose.
4. Within the adapter a translation scheme needs to be made for syntactic differences between the model under test and the model to generate test cases from.

TorX is able to derive test-cases from a dialect of the PROMELA language called TROJKA. The only difference between PROMELA and TROJKA is that in the TROJKA-language a channel can be defined as observable. This means that the communication over this channel is visible (e.g. for TorX to use as test-input or output). The messages over channels that are not declared observable are not used for test-case generation.

The correct behavior of the alternating bit protocol is that for every message sent, eventually the corresponding acknowledgement should be received. Only after an acknowledgment has been received the next message can be sent. The behavior of the system can be specified in TROJKA as follows:

```
mtype = {call, result, send,  m0, m1, a0, a1};

chan TDRV__channel = [0] of { mtype,mtype,mtype } OBSERVABLE;

active proctype Sender() {
      do
      :: TDRV__channel?call,send,m0;
         TDRV__channel!result,send,a0;
         TDRV__channel?call,send,m1;
         TDRV__channel!result,send,a1
      od
}
```

In this model input message `m0` is sent first, then output message `a0` is received. Then `m1` is sent after which `a1` is received. After that, `m0` is sent again and `a0` is received and so on. A simulation model is normally *closed.* There is no interaction between the simulation (processes) and "the outside world" (e.g. other tools or a user) to guide the simulation. In order to test the χ-model of the alternating bit protocol interaction with the TorX tool is needed. The TROJKA-model is open by means of the observable channel. The χ model can be made open by replacing process G (see Section 3) with a new process *IO.* Replacing process G does not affect the behavior of the alternating bit protocol itself (which is implemented by the processes *Sender*, *Medium*, and *Receiver*).

```
proc $io(inp\colon !\text{string}, outp\colon ?\text{string}) =$
$[\![\, m : \text{string}$
$|\, *[\, \text{true} \longrightarrow ?m$
            $;\, inp!m$
            $;\, outp?m$                                   ($\chi$-4.1)
            $;\, !m$
   $]$
$]\!]$
```

Instead of alternating between $msg0$ and $msg1$ inside a process, the message to be sent is received from a special purpose channel. In the χ-language this is a channel without a name and it is normally used for receiving input from a unix-terminal command line, or for printing output to the command line. For our purpose we need TorX to communicate with this special channel.

We have built an adapter to establish the connection between TorX and the simulation through this channel. It consists of a generic part and a model specific part. The generic part can be re-used; it implements a connection between TorX and χ by means of pipes. Because a pipe mechanism is used every data type used in the TROJKA-model needs to be translated to a string. The model-specific part is the translation of input and output messages between the TROJKA-model and the χ-model. PROMELA, and therefore TROJKA, allows data types name, bit, and byte. The χ language also allows data types string, boolean, and natural. To make testing possible, we need to define a mapping of messages defined in the TROJKA-model to the messages defined in the χ model, and the other way arround. In our alternating bit protocol example the message `m0` in the TROJKA-model is the message $msg0$ in the χ model. The same holds for the other message `m1` and the acknowledgments. In the adapter we define a mapping between these messages.

Our adapter presupposes that the TROJKA-model meets some mild syntactic conventions (that were already used in the TROJKA model earlier in this section):

– The channel names have to start with `TDRV__` in order to use the correct Python class in which the adapter has been implemented.
– The first part of the message has to be `call` or `result` to indicate an input to the simulation or an output from the simulation respectively.

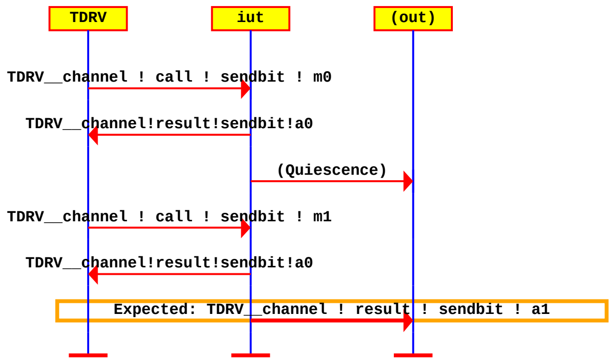

TDRV
iut
(out)
TDRV__channel ! call ! sendbit ! m0
TDRV__channel!result!sendbit!a0
(Quiescence)
TDRV__channel ! call ! sendbit ! m1
TDRV__channel!result!sendbit!a0
Expected: TDRV__channel ! result ! sendbit ! a1

Fig. 5. Error Found in Model-based Testing the Alternating Bit Protocol

– The second part of the message has to be the function call in the adapter (`send`) which sends the input to the simulation and returns the output from the simulation.

Using the adapter we can now test the χ-simulation of the alternating bit protocol with TorX. During test execution TorX generates a message sequence chart of the inputs applied to the simulation and the returned output from the simulation. The result is depicted in Fig. 5.

After providing message `m1` as input, acknowledgement `a0` is received. However, acknowledgement `a1` was expected as output. This was the acknowledgement to follow message `m1` in the TROJKA-model. Repeating this test with `m1` as first input shows that upon message `m0`, `a1` is received. In this case acknowledgement `a0` was the expected output from the simulation.

These observations lead to the conclusion a mistake could have been made in process *Sender*: When the first acknowledgement is received the variable a is assigned that value; after sending the second message a still has this value; therefore $a \in \{"ack0","ack1"\}$ and value a is returned as output. This leads to the conclusion that the second time the message is never sent over the medium. The solution for this problem is to read the first response on the current message before evaluating the guard of the repetition for the first time:

```
proc $Sender(sout: !\text{string}, sin: ?\text{string}, inp: ?\text{string}, outp: !\text{string}) =$
$[\![\, msg : \text{string}, a : \text{string}$
$|\ *[\ \mathsf{true} \longrightarrow inp?msg$
                 $;\ sout!msg$
                 $;\ sin?a$
                 $;\ *[\ a \notin \{"ack0","ack1"\} \longrightarrow sout!msg$                    ($\chi$-4.2)
                                                      $;\ sin?a$
                   $]$
                 $;\ outp!a$
   $]$
$]\!]$
```

Although this was a small example, it contained a mistake. By examining the model it can be hard to spot the mistake. With the help of model-based testing the mistake was found and the model used for validating the simulation model only contained ten lines of code.

5 Industrial Applicability

We applied our model-based testing approach to a χ-simulation model of a laser system. This system is part of a wafer-scanner machine. A wafer-scanner illuminates silicon discs with laser light so that an image of the integrated circuit is printed on it. Within the machine a controller is used to e.g. turn the laser on or off, to adjust the energy intensity of the laser beam, or for instance to adjust the frequency of the laser pulses. The machine can operate by itself or can be controlled from outside the machine through a terminal (see Fig. 6).

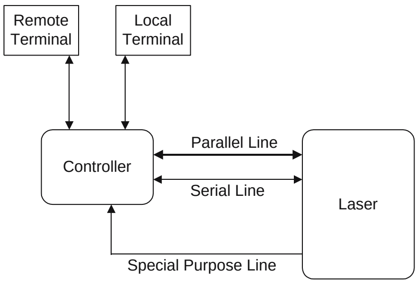


Fig. 6. Laser System

The χ-model we have been testing simulated the communication between controller and laser. Controller and laser were modelled, the two terminals were not modelled. Our initial purpose for testing this model with TorX was to find out whether the χ-simulation model was correct with respect to a TROJKA translation of this model. Later on, we also tested this χ-model against some other TROJKA-models. We did this to test the correctness of the χ-model with respect to the informal specification of the laser system.

Figure 7 shows the processes of the χ-model of the controller-laser system and the channels between them.

- Process C receives input from a process *Generator* (not depicted in Fig. 7). This input can be e.g. a signal that the laser has to fire a laser beam. In order to be able to fire a laser beam the controller sends a sequence of serial and parallel commands to process *IO*.
- Process *IO* is strictly used for directing input received from process C to underlying processes *LC* and *LS* and to communicate output received from all underlying processes via one channel back to the controller.

- Process LC models all communication except for communication related to the laser state. After receiving an input it generates the proper output response and informs process LS when the state variables of the laser need to be updated.
- Process LS keeps track of the state of the laser. When it is informed by process LC that an output of the laser state is required it generates this output command and sends it to process IO.

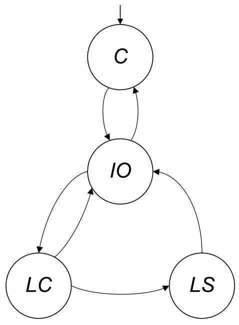


Fig. 7. Laser Model

We wanted to test the part of this model that specified the behavior of the laser. Therefore, we first removed process C. Then we made an open simulation model by adding a process which communicated with process IO. This process could communicate with TorX in the same way as presented in the alternating bit protocol example. The adapter contained the translation scheme between messages in the TROJKA-model and the χ-model. Using this adapter we were able to test the χ-model of the laser system against a one-to-one, but manual translation of this model to TROJKA. During these tests two mistakes were found:

1. We found a mistake in the χ-model: When the χ simulation received an input for which no behavior was specified it should return a specific serial command. This command, according to the informal specification, means it does not recognize the input. Instead, it returned a command indicating an error has occurred in the laser. Although the TROJKA-model was supposed to be a direct translation of the χ-model the mistake was not made in the TROJKA-model. As it turned out the mistake was removed from the TROJKA-model when this model was used to test a laser test-bench in a separate case study.
2. We found a mistake in the TROJKA-model: When the input command was given to perform a specific state change, the returned output from the simulation indicated a different state as specified in the TROJKA-model. As it turned out this mistake was introduced when the model was used for another test.

By testing the χ-model against other models two more mistakes were found:

3. Our experiments also revealed a bug in TorX: When tests where generated from one model TorX sometimes observed quiescent output, causing tests to fail. Quiescence normally indicates observing no output from the implementation. According to the specification used in the tests this should not have been possible. Always some input should have been offered or some output should have been observed. When tests were generated from another model this extra quiescence was not observed. This was strange because the same input was applied to the implementation. The cause of the bug was the use of internal communication between processes in the first specification. This generated quiescent output when that should not be possible, causing other tests to fail. TorX generated this extra quiescence because it makes use of the algorithms of the SPIN model-checker to traverse the state-space. In the meantime this bug has been fixed.
4. We also found that some test-cases failed, while both implementation and specification were correct according to their informal specification: The cause of these mistakes were that the implementation and specifications were based on documentation from different laser-system manufacturers. Because in practice only one controller is built to interact with the laser-system, using different laser-systems can be a cause for bugs.

This case study showed that model-based testing can be useful in an industrial context. By finding mistakes in a simulation ,the quality of the model was improved. The one-to-one translation of this model was used for testing in industry with the TorX tool. By improving the quality of the models in this case, the quality of testing a machine component, and therefore the quality of the machine, was improved.

6 Conclusions

We have shown that it is possible to test discrete-event χ-models with the model-based test tool TorX. To illustrate our method we tested a χ-model of the alternating bit protocol. To be able to achieve this we needed to:

- Make a specification of the model to be tested.
- Make it possible to communicate with the simulation model to be tested. (We were able to do this without making changes to the code to be tested of the χ-model.)
- Establish a connection between TorX and the χ-simulation through an adapter.
- Make a translation scheme for the messages used in the specification and the messages used in the implementation inside the adapter.

We made an adapter specifically for discrete event χ-models. Depending on the messages to be exchanged between specification and simulation, we need to make a mapping between them. We believe this method is extensible to other discrete event simulation languages and tools, however for every tool/language an adapter needs to be build to establish the interface between test-tool and simulator.

Testing experiments with a simulation model of an industrial system showed added value of this method in practice. We were able show mistakes in a simulation model, that was also used for testing in industry. By testing a simulation model we also improve the quality of the final product.

We also observed that a test failure is not always caused by the model that is being tested. When a test case fails, the mistake can be caused by the specification or by the test-tool also. Even with automated model-based testing it is necessary to thoroughly analyze every possible cause of test-case failure. Testing a model with more than one specification was helpful in that respect.

The χ simulation language also allows timed, continuous, and stochastic behavior. TorX recently allows the possibility of timed testing. We believe that we can test these timed aspects of χ-models as well. To do this we need a timed specification language and make our adapter suitable for timed testing. We are also extending the ioco test-theory and model-based testing with the possibilities to test hybrid systems (i.e. systems with both discrete and continuous behavior). The goal of this research is to test real industrial systems. This research will also be useful for testing hybrid simulation models.

Acknowledgments

Hereby I would like to thank Jan Tretmans and all members of the model-based testing project TANGRAM for their help in this research. Furthermore, I would like to thank Bas Luttik and Jos Baeten from the Computer Science department of Eindhoven University of Technology for their advice. I also would like to thank Albert Hofkamp of the Mechanical Engineering department of Eindhoven University of Technology for his help in building the connection between TorX and the χ-simulator. Rob Hendrikx, Kasia Peplowska, and Jusuf Anggono provided the χ simulation model of the laser system and the PROMELA translation of that model.

References

1. Mathworks: http://www.mathworks.com (2005)
2. Instruments, N.: http://www.ni.com (2005)
3. Stobie, K.: Model Based Testing in Practice at Microsoft. In: Proceedings of MBT 2004. (2004)
4. Fernandez, J., Jard, C., T.Jeron, Viho, C.: An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. In Groote, J., Rem, M., eds.: Special Issue of Industrially Relevant Applications fo Formal Analysis Techniques, Elsevier (1996)

5. Farchi, E., Hartman, A., Pinter, S.: Using a Model-based Test Generator to Test for Standard Conformance. Volume 41, Number 1 of IBM Systems Journal. (2002) 89–110
6. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools **17** (1996) 103–120
7. Tretmans, J., Brinksma, E.: TorX: Automated Model Based Testing. In Hartman, A., Dussa-Ziegler, K., eds.: Proceedings of the 1st European Conference on Model-Driven Software Engineering. (2003)
8. Hofkanp, A., Rooda, J.: χ Reference Manual. Mechanical Engineering Department, Systems Engineering group, Eindhoven University of Technology, Eindhoven, The Netherlands. (2002)
9. van den Mortel-Fronczak, J., Rooda, J.: Application of Concurrent Programming to Specification of Industrial Systems. In: Proceedings of INCOM'95. (1995) 421–426
10. de Resyste, C.: http://fmt.cs.utwente.nl/cdr (2005)
11. de Vries, R., Belinfante, A., Feenstra, J.: Testing in practice: The highway tolling system. In Schiefendecker, I., König, H., Wolisz, A., eds.: Testing of Communicating Systems XIV, Berlin, Germany, Kluwer academic publishers (2002) 210–234
12. Heerink, L., Feenstra, J., Tretmans, J.: Formal Test Automation: The Conference Protocol with PHACT. In Ural, H., Probert, R., Bochmann, G.v., eds.: Testing of Communicating Systems – Procs. of TestCom 2000, Kluwer Academic Publishers (2000) 211–220
13. Tretmans, J.: Testing concurrent systems: A formal approach. In Baeten, J., Mauw, S., eds.: CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory. Volume 1664 of Lecture Notes in Computer Science., Springer-Verlag (1999) 46–65
14. de Vries, R., Tretmans, J.: On-the-Fly Conformance Testing using SPIN. Software Tools for Technology Transfer **2** (2000) 382–393
15. Garavel, H.: OPEN/CAESAR: An Open Software Architecture for Verfication, Simulation and Testing. In Steffen, B., ed.: Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Volume 1384 of Lecture Notes in Computer Science., Springer-Verlag (1998) 68–84
16. Holzmann, G.: Design and Validation of Computer Protocols, Prentice Hall Inc. (1991)
17. R.G. de Vries: Towards Formal Test Purposes. In Tretmans, J., Brinksma, E., eds.: Formal Approaches to Testing of Software 2001 (FATES'01). BRICS Notes Series (NS-01-4) (2001) 61–76
18. Bos, V., Kleijn, J.: Formal specification and analysis of production systems. In: Proceedings of the 16th International Conference on Production Research. (2001)
19. van Beek, D., van der Ham, A., Rooda, J.: Modelling and Control of Process Industry Batch Production Systems. In: 15th Triennial World Congress of the International Federation of Automatic Control. (2002)

Jartege: A Tool for Random Generation of Unit Tests for Java Classes

Catherine Oriat

LSR-IMAG, Grenoble
`Catherine.Oriat@imag.fr`

Abstract. This paper presents Jartege, a tool which allows random generation of unit tests for Java classes specified in JML. JML (Java Modeling Language) is a specification language for Java which allows one to write invariants for classes, and pre- and postconditions for operations. As in the JML-JUnit tool, we use JML specifications on the one hand to eliminate irrelevant test cases, and on the other hand as a test oracle. Jartege randomly generates test cases, which consist of a sequence of constructor and method calls for the classes under test. The random aspect of the tool can be parameterized by associating weights to classes and operations, and by controlling the number of instances which are created. The practical use of Jartege is illustrated by a small case study.

1 Introduction

Main validation technique in software engineering, program testing aims at ensuring that the program is *correct*, i.e. conforms to its specifications. As the input domain of a program is usually very large or infinite, exhaustive testing, which consists in testing the program for all its possible inputs, is in general impossible. The objective of testing is thus rather to improve the software quality by finding faults in it. Testing is an important activity of software development, whose cost is usually estimated to about 40% of the total cost of software development, exceeding the cost of code writing.

A test campaign for a program requires several steps: design and development of test sets, execution and results examination (or *oracle*). Considering the cost of testing, it is interesting to automate some of these steps.

For Java programs, the JUnit framework [1,2] allows the developer to write an oracle for each test case, and to automatically execute test sets. JUnit in particular permits to automatically regression test several test sets.

If a formal specification is available, it can be translated into assertions which can be checked at runtime, and thus serve as a test oracle. For instance, the DAISTS system [3] compiles algebraic axioms of an abstract data type into consistency checks; Rosenblum's APP pre-processor allows the programmer to write assertions for C programs [4]; the Eiffel *design and contract* approach integrates assertions in the programming language [5,6].

It is also interesting to automate the *development* of tests. We can distinguish between two groups of strategies to produce test sets: random and systematic strategies. Systematic strategies, such as functional testing or structural testing, consist in decomposing the input domain of the program in several subdomains, often called “partitions”.

Many systematic strategies propose to derive test cases from a formal specification [7]. For instance, the Dick and Faivre method [8], which consists in constructing a finite state automaton from the formal specification and in selecting test cases as paths in this automaton, has been used as a basis by other approaches, in particular Casting [9] or BZTT [10]. BZTT uses B or Z specifications to generate test cases which consist in placing the system in a boundary state and calling an operation with a boundary value.

In contrast to systematic methods, random testing generally does not use the program nor the specification to produce test sets. It may use an *operational profile* of the program, which describe how the program is expected to be used. The utility of random testing is controversial in the testing community. It is usually presented as the poorest approach for selecting test data [11]. However, random testing has a few advantages which make us think that it could be a good complement to systematic testing:

– random testing is cheap and rather easy to implement. In particular, it can produce large or very large test sets;
– it can detect a substantial number of errors at a low cost [12,13,14].

Moreover, if an operational profile of the program is available,

– random testing allows early detection of the failures that are most likely to appear when using the program [15];
– it can be used to evaluate the program *reliability* [13,16], a way of quantifying its quality.

However, partition testing can be much more effective at finding failures, especially when the strategy defines some very small subdomains with a high probability to cause failures [17].

Among the best practices of *extreme programming*, or XP [18,19], are *continuous testing* and *code refactoring*. While they are writing code, developers should write corresponding unit tests, using a testing framework such as JUnit. Unit testing is often presented as a support to refactoring: it gives the developer confidence that the changes have not introduced new errors. However, code refactoring often requires to change some of the corresponding unit tests as well. Both practices can therefore be hard to conciliate, especially when the amount of code corresponding to the tests exceeds the amount of code of the class.

This paper proposes to use random generation of tests to facilitate continuous testing and code refactoring in the context of extreme programming. We presents Jartege, a tool for random generation of unit tests for Java classes specified in JML, which aims at easily producing numerous test cases, in order to detect a substantial number of errors at a low cost.

The rest of the paper is organized as follows. Section 2 introduces the approach. Section 3 presents a case study which consists in modeling bank accounts.

This case study will serve to illustrate the use of our testing tool. Section 4 presents our tool Jartege and how it can be used to test the bank account case study. Section 5 introduces more advanced features of Jartege, which allow one to parameterize its random aspect. Section 6 shows the errors which are detected by test cases generated by Jartege in the case study. Section 7 presents and compares related approaches. Section 8 discusses some points about random generation of tests and draw future work we intend to undertake around Jartege.

2 Approach

JML (Java Modeling Language) is a specification language for Java inspired by Eiffel, VDM and Larch, which was designed by Gary Leavens and his colleagues [20,21,22]. Several teams are currently still working on JML design and tools around JML [23]. JML allows one to specify various assertions in particular invariants for classes as well as pre- and postconditions for methods.

The JML compiler (jmlc) [24] translates JML specifications into assertions checked at runtime. If an assertion is violated, then a specific exception is raised. In the context of a given method call, the JML compiler makes a useful difference between an *entry precondition* which is a precondition of the given method, and an *internal precondition*, which is a precondition of an operation being called, at some level, by the given method.

The JML-JUnit tool [25] generates JUnit test cases for a Java program specified in JML, using the JML compiler to translate JML specifications into test oracles. Test cases are produced from a test fixture and parameter values supplied by the user.

Our approach is inspired by the JML-JUnit tool: we propose to generate random tests for Java programs specified in JML, using this specification as a test oracle, in the JML-JUnit way. Our aim is to produce a big number of tests at a low cost, in order to facilitate unit testing.

To implement these ideas, we started developing a prototype tool, called *Jartege* for *Java Random Test Generator*. Jartege is designed to generate *unit tests* for Java classes specified with JML. By unit tests, we here mean tests for some operations in a single class or a small cluster of classes. In our context, a *test case* is a Java method which consists of *a sequence of operation (constructor or method) calls*. As in the JML-JUnit tool, we use the JML specification to assist test generation in two ways:

1. It permits the rejection of test sequences which contain an operation call which violates the operation *entry precondition*. We consider that these test sequences are not interesting because they detect errors which correspond to a fault *in the test program*. (Although such sequences could be used to detect cases when a precondition is too strong, this goal would prevent us from producing long sequences of calls. Thus, we choose to trust the specification rather than the code. We can note that if an operation uses a method whose precondition is too strong, this will produce an *internal precondition* error and the sequence will *not* be rejected.)

2. The specification is also used as a test oracle: the test detects an error when another assertion (e.g. an invariant, a postcondition or an *internal precondition*) is violated. Such an error corresponds to a fault in the Java program or in the JML specification.

Our work has been influenced by the Lutess tool [26,27], which aims at deriving test data for synchronous programs, with various generation methods, in particular a purely random generation and a generation guided by operational profiles. The good results obtained by Lutess, which won the best tool award of the first feature interaction detection contest [28], have encouraged us to consider random generation of tests as a viable approach.

3 Case Study

In this section, we present a case study to illustrate the use of Jartege. This case study defines *bank accounts* and some operations on these accounts.

We describe here an informal specification of bank accounts:

1. An account contains a certain available amount of money (its *balance*), and is associated with a minimum amount that this account may contain (the *minimum balance*).
2. It is possible to credit or debit an account. A debit operation is only possible if there is enough money on the account.
3. One or several last credit or debit operations may be cancelled.
4. The minimum balance of an account may be changed.

To represent accounts, we define a class Account with two attributes: balance and min which respectively represent the balance and the minimum balance of the account, and three methods: credit, debit and cancel. In order to implement the cancel operation, we associate with each account an *history*, which is a linked list of the previous balances of the account. The class History has one attribute, balance, which represents the balance of its associated account before the last credit or debit operation. With each history is associated its preceding history. Figure 1 shows the UML class diagram of bank accounts.

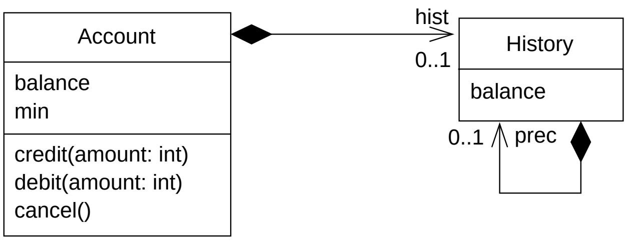


Fig. 1. UML diagram of the case study

We implement both class Account and History in Java, and specify them with JML. We choose to write *lightweight public specifications* to emphasize that these specifications are destined for clients and need not be complete.

In order to forbid uncontrolled modifications of the attributes, we declare them as private and define associated access methods: getBalance, getMin and getHist in class Account and getBalance and getPrec in class History. These methods are specified as *pure* methods in JML because they are side effect free, which allows us to use them in JML assertions.

The class Account has an invariant which specifies that the balance of an account must always be greater than the minimum balance.

```
/* Class of bank accounts. */
public class Account {
    /* Invariant of class Account. */
    /*@ public invariant getBalance() >= getMin(); */

    private int balance; // The balance of this account
    private int min;     // The minimum balance
    private History hist; // The history list of this account

    /* The balance of this account. */
    public /*@ pure */ int getBalance() { return balance; }

    /* The history list of this account. */
    public /*@ pure */ History getHist() { return hist; }

    /* The minimum balance of this account. */
    public /*@ pure */ int getMin() { return min; }
```

The constructor of class Account constructs an account with the specified balance and the specified minimum balance. Its precondition asserts that the specified balance is greater than the specified minimum balance.

```
    /* Constructs an account with the specified balance and
     * minimum balance. */
    /*@ requires balance >= min; */
    public Account(int balance, int min) {
        this.balance = balance;
        this.min = min;
        this.hist = null;
    }
```

As the minimum balance min is a private attribute, we have to introduce a method to change its value. The method setMin(*int* min) sets the minimum balance to the specified value. Its precondition asserts that the balance is greater than the specified minimum value.

```
  /* Sets the minimum balance to the specified value. */
  /*@ requires getBalance() >= min; */
  public void setMin(int min) { this.min = min; }
```

The method credit(*int* amount) credits the account with the specified amount. Its precondition requires the amount to be positive. Its postcondition asserts that the new balance is the former balance augmented by the specified amount, that a new history is created with balance the former balance of the account and with previous history the former history of the account. Its exceptional postcondition asserts that the method should never terminate abruptly.

```
  /* Credits this account with the specified amount. */
  /*@ requires amount >= 0;
   *@ ensures getBalance() == \old(getBalance()) + amount &&
   *@         \fresh(getHist()) &&
   *@         getHist().getBalance() == \old(getBalance()) &&
   *@         getHist().getPrec() == \old(getHist());
   *@ signals (Exception e) false;
   */
  public void credit(int amount) {
     hist = new History(balance, getHist());
     balance = balance + amount;
  }
```

The debit operation, which is very similar to the credit operation, is not detailed here. It has the additional precondition that the balance decreased by the specified amount is greater than the minimum balance.

The method cancel cancels the last credit or debit operation. Its precondition requires that the history is not null, which means that at least one operation of credit or debit has taken place since the account was created. Its postcondition ensures that the balance and the history of the account have been updated with their former values.

```
  /* Cancels the last credit or debit operation. */
  /*@ requires getHist() != null;
   *@ ensures getHist() == \old(getHist().getPrec()) &&
   *@         getBalance() == \old(getHist().getBalance());
   *@ signals (Exception e) false;
   */
  public void cancel() {
     balance = hist.getBalance();
     hist = hist.getPrec();
  }
} // End of class Account
```

We do not define any JML assertion for the class History.

```
/* Class of histories. */
public class History {
  private int balance;   // The balance of this history.
  private History prec; // The preceding history.
  /* Constructs a history with the specified balance and preceding history. */
  public History(int balance, History prec) {
     this.balance = balance; this.prec = prec;
  }
  /* The balance of this history. */
  public /*@ pure */ int getBalance() { return balance; }
  /* The preceding history. */
  public /*@ pure */ History getPrec() { return prec; }
} // End of class History
```

4 Jartege

Jartege (Java Random Test Generator) is a framework for automatic random generation of unit tests for Java classes specified with JML. This approach consists in producing test programs which are composed of test cases, each test case consisting of randomly chosen sequences of method calls for each class under test. Each generated test program can be executed to test the classes, and re-executed later on either after having corrected some faults or for regression test.

The tool is designed to produce *unit tests*, i.e. tests composed of calls of some methods which belong to a few classes. As noticed in [29], because of complex dependences that exist between classes in object-oriented programs, it is usually not possible to test a method or a class in complete isolation. Jartege thus is able to generate test cases which allow the integration of several classes.

4.1 Practical Use of Jartege

Suppose we wish to generate tests for the classes Account and History. We write the following Java program:

```
/** Jartege test cases generator for classes Account and History. */
class TestGen {
  public static void main(String[] args) {
     ClassTester t = new ClassTester(); // Creates a class tester
     t.addClass("Account"); // Adds the specified classes to
     t.addClass("History");   // the set of classes under test
     // Generates a test class TestBank, made of 100 test cases.
     // For each test case, the tool tries to generate 50 method calls.
     t.generate("TestBank", 100, 50);
  }}
```

The main class of the Jartege framework is ClassTester. This class must be instantiated to allow the creation of test programs.

The method addClass (String className) adds the class className to the set of classes under test. In this example, we wish to generate tests for the classes Account and History.

The method generate (String className, *int* numberOfTests, *int* numberOfMethodCalls) generates a file *className*.java which contains a class called *className*. This class is composed of *numberOfTests* test cases. For each test case, the tool makes *numberOfMethodCalls* attempts to generate a method call of one of the classes under test. Using the accessible constructors, the tool constructs objects which serve as parameters for these method calls.

When the program TestGen is executed, it produces a file TestGen.java which contains a main program. This main program calls successively 100 test methods test1 ( ), test2 ( ) ... test100 ( ). Each test method contains about 50 method calls.

While the program is generated, Jartege executes *on the fly* each operation call, which allows it to eliminate calls which violate the operation precondition. When this precondition is strong, it may happen that the tool does not succeed in generating a call for a given method, which explains that *about* 50 method calls are generated.

4.2 Test Programs Produced by Jartege

A test program produced by Jartege is a class with a main method which consists in calling sequentially all generated test cases. Each test case consists of a sequence of constructor and method calls of the classes under test. Here is an example of such a test case:

```
// Test case number 1
public void test1 ( ) throws Exception {
  try {
    Account ob1 = new Account (1023296578, 223978640);
    ob1.debit (152022897);
    History ob2 = new History (1661966075, (History) null);
    History ob3 = new History (-350589348, ob2);
    History ob4 = ob2.getPrec ( );
    int ob5 = ob3.getBalance ( );
    ob1.cancel ( );
    // ...
  } catch (Throwable _except) {
    error (_except, 1);
  }}
```

For each test method, if a JML exception is raised, then an error message (coming from jmlc) is printed. The test program terminates by printing an assessment of the test. As an example, here is an excerpt of what is printed with a generated program TestBank.java:

```
1) Error detected in class TestBank by method test2: JMLInvariantError:
    By method "credit@post⟨Account.java:79:18⟩" of class "Account"
    for assertions specified at Account.java:11:32 [...]
    at TestBank.test2(TestBank.java:138)
[...]
Number of tests: 100
Number of errors: 71
Number of inconclusive tests: 0
```

The program has detected 71 errors. The first error detected comes from a violation of the invariant of class Account (specified line 11), which happened after a credit operation.

The test program also indicates the number of *inconclusive* tests. A test case is inconclusive when it does not allow one to conclude whether the program behaviour is correct or not. A test program generated by Jartege indicates that a test case is inconclusive when it contains an operation call whose *entry precondition* is violated. As Jartege is designed to eliminate such operation calls, this situation may only arise when the code or the specification of one of the classes under test has been modified after the test file was generated. A high number of inconclusive tests indicates that the test file is no longer relevant.

5 Controlling Random Generation

If we leave everything to chance, Jartege might not produce interesting sequences of calls. Jartege thus provides a few possibilities to parameterize its random aspect. These features can be useful for stress testing, for instance if we want to test more intensively a given method. More generally, they allow us to define an *operational profile* for the classes under test, which describe how these classes are likely to be used by other components.

Weights. With each class and operation of a class is associated a weight, which defines the probability that a class will be chosen, and that an operation of this class will be called. In particular, it is possible to forbid to call some operation by associating a null weight with it.

Creation of Objects. Objects creation is commanded by *creation probability functions*, which define the probability of creating a new object according to the number of existing objects of the class against that of reusing an already created object. If this probability is low, Jartege is more likely to reuse an already created object than to construct a new one. This allows the user either to create a predefined number of instances for a given class, or on the opposite, to create numerous instances for a class.

In the example of bank accounts, it is not very interesting to create many accounts. It is possible to test the class Account more efficiently for example by creating a unique account and by applying numerous method calls to it.

The function changeCreationProbability(String className, CreationProbability creationProbabilityFunction) changes the creation probability function associated with the specified class to the the specified creation probability function.

The interface CreationProbability contains a unique method $f : (double) \rightarrow int$ such that $f(0) = 1$ and $f(n) \in [0, 1]$, $\forall n \geq 1$.

The class ThresholdProbability allows one to define *threshold probability functions* whose value is 1 under some threshold s and 0 above: $f(n) = 1$, if $n < s$; $f(n) = 0$, otherwise. A threshold probability function with threshold s allows one to define at most s instances of a given class. We can for instance forbid the creation of more than one instance of Account by adding the following statement in the test generator:

```
t.changeCreationProbability("Account", new ThresholdProbability(1));
```

Parameter Generation of Primitive Types. When a method has a strong precondition, the probability that Jartege, without any further indication, will generate a call to this method which does not violate this precondition is low. For primitive types, Jartege provides the possibility to define generators for some parameters of a given method.

For example, the precondition of the debit operation requires the parameter amount to be in range [0, getBalance( ) − getMin( )]. Let us suppose the range is small, in other words that the balance is closed to the minimum balance. If Jartege chooses an amount to be debited entirely randomly, this amount is not likely to satisfy the method precondition.

Jartege provides a way of generating parameter values of primitive types for operations. For this, we define a class JRT_Account as follows.

```
public class JRT_Account {
  private Account theAccount; // The current account
  /* Constructor. */
  public JRT_Account(Account theAccount){ this.theAccount = theAccount;}
  /** Generator for the first parameter of operation debit (int). */
  public int JRT_debit_int_1( ) {
     return RandomValue.intValue(0,
        theAccount.getBalance( ) - theAccount.getMin( ));
  }}
```

The class JRT_Account must contain a private field of type Account which will contain the current object on which an operation of class Account is applied. A constructor allows Jartege to initialize this private field. The class also contains one parameter generation method for each parameter for which we specify the generation of values. In the example, to specify the generation of the first parameter of operation debit(*int* amount), we define the method *int* JRT_debit_int_1( ). We use the signature of the operation in the name of the method to allow overloading. The method RandomValue.intValue(*int* min, *int* max) chooses a random integer in range [min, max].

Fixtures. If we want to generate several test cases which operate on a particular set of objects, we can write a *test fixture*, in a similar way to JUnit. A test fixture is a class which contains:

– attributes corresponding to the objects a test operates on;
– an optional setUp method which defines the preamble of a test case (which typically constructs these objects);
– an optional tearDown method which defines the postamble of a test case.

6 Applying Jartege to the Case Study

The 100 test cases generated by Jartege, showing 71 failures, revealed three different errors: one error caused by a credit operation, and two errors caused by a cancel operation. We extracted the shorter sequence of calls which resulted in each failure and obtained the following results. We also changed the parameter values and added some comments for more readability.

Error 1. The credit operation can produce a balance inferior to the previous balance because of an integer overflow.

```
public void test1() {
  Account ob1 = new Account(250000000, 0);
  ob1.credit(2000000000); // Produces a negative balance,
}                         // below the minimum balance.
```

Error 2. The cancel operation can produce an incorrect result if it is preceded by a setMin operation which changes the minimum balance of the account to a value which is superior to the balance before cancellation.

```
public void test11() {
  Account ob1 = new Account(-50, -100);
  ob1.credit(100);
  ob1.setMin(0);
  ob1.cancel(); // Restores the balance to a value
}               // inferior to the minimum balance.
```

Error 3. The third error detected is a combination of an overflow on a debit operation (similar to Error 1, which comes from an overflow on a credit operation) and of the second error.

```
public void test13() {
  Account ob1 = new Account(-1500000000,-2000000000);
  ob1.debit(800000000); // Produces a positive balance.
  ob1.setMin(0);
  ob1.cancel();   // Restores the balance to a value
}                 // inferior to the minimum balance.
```

We have the feeling that the three errors detected with test cases generated by Jartege are not totally obvious, and could have easily been forgotten in a manually developed test suite. Errors 2 and 3 in particular require three method calls to be executed in a specific order and with particular parameter values.

It must be noted that the case study was originally written to show the use of JML to undergraduate students, without us being aware of the faults.

7 Comparison with Related Work

Our work has been widely inspired by the JML-JUnit approach [25]. The JML-JUnit tool generates test cases for a method which consist of a combination of calls of this method with various parameter values. The tester must supply the object invoking the method and the parameter values. With this approach, interesting values could easily be forgotten by the tester. Moreover, as a test case only consists of one method call, it is not possible to detect errors which result of several calls of different methods. At last, the JML-JUnit approach compels the user to construct the test data, which may require the call of several constructors. Our approach thus has the advantage of being more automatic, and of being able to detect more potential errors.

Korat [30] is a tool also based on the JML-JUnit approach, which allows exhaustive testing of a method for all objects of a bounded size. The tools automatically construct all non isomorphic test cases and execute the method on each test case. Korat therefore has the advantage over JML-JUnit of being able to construct the objects which invoke the method under test. However, test cases constructed by Korat only consist of one object construction and one method invocation on this object.

Tobias [31,32] is a combinatorial testing tool for automatic generation of test cases derived from a "test pattern", which abstractly describes a test case. Tobias was first designed to produce test objectives for the TGV tool [33] and was then adapted to produce test cases for Java programs specified in JML or VDM. The main problem of Tobias is the combinatorial explosion which happens if one tries to generate test cases which consist of more than a couple of method calls. Jartege was designed while testing a small industrial application [34] in order to allow the generation of long test sequences without facing the problem of combinatorial explosion.

8 Discussion and Future Work

Jartege is only in its infancy and a lot of work remains to be done.

Primitive values generation for methods parameters is currently done manually by writing primitive parameters generating methods. Code for these methods could be automatically constructed from the JML precondition of the method. This could consist in extracting range constraints from the method precondition and automatically produce a method which could generate meaningful values for the primitive parameters.

Jartege easily constructs test cases which consist of hundreds of constructors and methods calls. It would be useful to develop a tool for extracting a minimum sequence of calls which results in a given failure.

We developed Jartege in Java, and we specified some of its classes with JML. We applied Jartege to these classes to produce test cases, which allowed us to experiment our tool on a larger case study and to detect a few errors. We found much easier to produce tests with Jartege than to write unit tests with JUnit or JML-JUnit. We intend to continue our work of specifying Jartege in JML and testing its classes with itself. We hope that this real case study will help us to evaluate the effectiveness and scalability of the approach.

A comparison of our work with other testing strategies still remains to be done. We can expect systematic methods, using for instance boundary testing such as BZTT [10], to be able to produce more interesting test cases that ours. Our goal is certainly not to pretend that tests produced randomly can replace tests produced by more sophisticated methods, nor a carefully designed test set written by an experienced tester.

Our first goal in developing Jartege was to help the developer to write unit tests for unstable Java classes, thus for "debug unit testing". It would also be interesting to use Jartege to evaluate the reliability of a stable component before it is released. Jartege provides some features to define an operational profile of a component, which should allow statistical testing. However, the definition of a correct operational profile, especially in the context of object-oriented programming, is a difficult task. Moreover, the relation between test sets generated by Jartege and the reliability of a component requires more theoretical work, one difficult point being to take into account the state of the component.

9 Conclusion

This paper presents Jartege, a tool for random generation of unit tests for Java classes specified in JML. The aim of the tool is to easily produce numerous test cases, in order to detect a substantial number of errors without too much effort. It is designed to produce automated tests, which can in part replace tests written by the developer using for instance JUnit. We think that the automatic generation of such unit tests should facilitate continuous testing as well as code refactoring in the context of extreme programming.

The JML specifications are used on the one hand to eliminate irrelevant test cases, and on the other hand as a test oracle. We think that the additional cost of specification writing should be compensated by the automatic oracle provided by the JML compiler, as long as we wish to intensively test the classes. Moreover, this approach has the advantage of supporting the debugging of a specification along with the corresponding program. This allows the developer to increase his confidence in the specification and to use this specification in other tools.

Most test generation methods are deterministic, while our approach is statistical. We do not wish to oppose both approaches, thinking that both have their advantages and drawbacks, and that a combination of both could be fruitful.

At last, we found JML to be good language to start learning formal methods. Its Java-based syntax makes it easy to learn for Java programmers. As JML specifications are included in Java source code as comments, it is easy to develop and debug a Java program along with its specification. Moreover, automatic test oracles as well as automatic generation of test cases are good reasons of using specification languages such as JML.

References

1. The JUnit Home Page, http://www.junit.org.
2. Beck, K., Gamma, E.: Test infected: Programmers love writing tests. Java Reports **3** (1998) 51–56
3. Gannon, J., McMullin, P., Hamlet, R.: Data-abstraction implementation, specification, and testing. ACM Transactions on Programming Languages and Systems **3** (1981) 211–233
4. Rosenblum, D.S.: Towards a method of programming with assertions. In: International Conference on Software Engineering — ICSE'92, IEEE Computer Society Press (1992)
5. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
6. Meyer, B.: Applying design by contract. IEEE Computer **25** (1992) 40–51
7. Gaudel, M.C.: Testing can be formal too. In: Proceedings of TAPSOFT'95, Aarhus, Denmark. Number 915 in LNCS, Springer-Verlag (1995) 82–96
8. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Proceedings of FME'93. Number 670 in LNCS, Springer-Verlag (1993) 268–284
9. Aertryck, L.V., Benveniste, M., Métayer, D.L.: Casting: a formally based software test generation method. In: Proceedings of the First IEEE Internatinal Conference on Formal Engineering Methods — ICFEM'97, Hiroshima, Japan. (1997) 101–111
10. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Proceedings of FME'02, Formal Methods Europe, Copenhaguen, Denmark. Number 2391 in LNCS, Springer-Verlag (2002) 21–40
11. Myers, G.J.: The Art of Software Testing. John Wiley and Sons, New York (1994)
12. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering **10** (1984) 438–444
13. Hamlet, D., Taylor, R.: Partition testing does not inspire confidence. IEEE Transactions on Software Engineering **16** (1990)
14. Ntafos, S.C.: On comparisons of random, partition, and proportional partition testing. IEEE Transactions on Software Engineering **27** (2001) 949–960
15. Frankl, P.G., Hamlet, R.G., LittleWood, B., Strigini, L.: Evaluating testing methods by delivered reliability. IEEE Transactions on Software Engineering **24** (1998) 586–601
16. Hamlet, R.: Random testing. In Marciniak, J., ed.: Encyclopedia of Software Engineering, Wiley (1994) 970–978
17. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Transactions on Software Engineering **17** (1991) 703–711
18. Beck, K.: Embracing change with extreme programming. IEEE Computer **32** (1999) 70–77
19. Beck, K.: Extreme Programming Explained. Addison Wesley (2000)

20. The Java Modeling Language (JML) Home Page, `http://www.cs.iastate.edu/~leavens/JML`.
21. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University (1998–2003)
22. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C.: JML Reference Manual. (Draft) (April 2003)
23. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. Technical Report NIII-R0309, Department of Computer Science, University of Nijmegen (March 2003)
24. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Hamid R. Arabnia and Youngsong Mun (eds.), International Conference on Software Engineering Research and Practice — SERP'02, Las Vegas, Nevada. (2002) 322–328
25. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Boris Magnusson (ed.), 16th European Conference on Object-Oriented Programming — ECOOP'02, Malaga, Spain. Number 2374 in Lecture Notes in Computer Science, Springer-Verlag (2002) 231–255
26. Parissis, I.: Test de logiciels synchrones spécifiés en Lustre. PhD thesis, Grenoble, France (1996)
27. du Bousquet, L., Ouabdesselam, F., Richier, J.L., Zuanon, N.: Lutess: a specification-driven testing environment for synchronous software. In: International Conference on Software Engineering — ICSE'99, Los Angeles, USA, ACM Press (1999)
28. du Bousquet, L., Zuanon, N.: An overview of Lutess: a specification-based tool for testing synchronous software. In: 14th IEEE International Conference on Automated Software Engineering — ASE'99. (1999) 208–215
29. Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., Durand, M.H.: Testing levels for object-oriented software. In: Proceedings of the 22nd International Conference on Software Engineering — ICSE'00, Limerick, Ireland, ACM (2000) 136–145
30. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proceedings of the International Symposium on Software Testing and Analysis — ISSTA'02, Rome. (2002) 123–133
31. Ledru, Y.: The TOBIAS test generator and its adaptation to some ASE challenges (position paper). In: Workshop on the State of the Art in Automated Software Engineering, ICS Technical Report UCI-ICS-02-17, University of California, Irvine, USA. (2002)
32. Maury, O., Ledru, Y., Bontron, P., du Bousquet, L.: Using TOBIAS for the automatic generation of VDM test cases. In: Third VDM Workshop (at FME'02), Copenhaguen, Denmark. (2002)
33. Jéron, T., Morel, P.: Test generation derived from model-checking. In: Proceedings of the 11th International Conference on Computer Aided Verification — CAV'99, Trento, Italy. Number 1633 in LNCS, Springer-Verlag (1999) 108–121
34. du Bousquet, L., Lanet, J.L., Ledru, Y., Maury, O., Oriat, C.: A case study in JML-based software validation. In: 19th IEEE International Conference on Automated Software Engineering, Austria — ASE'04. (2004) 294–297

FlexTest: An Aspect-Oriented Framework for Unit Testing

Dehla Sokenou and Matthias Vösgen

Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik,
Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Softwaretechnik,
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin
{dsokenou, mvoesgen}@cs.tu-berlin.de

Abstract. This paper examines whether test problems that occur specifically during unit testing of object-oriented programs can be solved using the aspect-oriented programming paradigm.

It presents the various problems in unit testing, shows conventional solutions and describes aspect-oriented solutions to the problems. The aspect-oriented solutions are supported by the unit test framework *FlexTest* of which the paper gives an overview.

1 Introduction

Unit testing is popular in the domain of object-oriented software development. Unit tests are written using the same implementation language as is used for the application under test.

However, some problems occur when unit testing object-oriented programs. There are two different reasons for this. Firstly, problems result from object-oriented characteristics like encapsulation and inheritance. Secondly, problems are caused by unit testing itself. Writing test cases directly in an implementation language is easy for programmers but it is sometimes a repetitive task. All constraints of the used language hold for the test, too.

In this paper, we investigate aspect-oriented programming techniques as a solution to a few of the problems encountered in unit testing. We present the various problems in unit testing, show conventional solutions and describe aspect-oriented solutions to the problems.

The unit test framework *FlexTest* [1] is presented, which supports the given solutions. For each solution, we give an example of how it is implemented in *FlexTest*.

The paper is organized as follows. Section 2 gives a brief introduction to aspect-oriented programming. Section 3 looks at the question of wether unit testing is a cross-cutting concern in the sense of aspect-oriented programming with regard to the application under test. In Section 4, we consider the unit test framework *FlexTest* in relation to the given problems and their solutions. Section 5 compares our work with similar approaches. Finally, we give a conclusion and suggest some entry points for discussion in Section 6. This section also includes an outlook on future work.

2 Aspect-Oriented Programming Techniques

Some concerns cannot be encapsulated in a class using object-oriented software development. Classical examples here are logging, security, and synchronization. For example, the logging server is implemented in one class, but client code is needed in all classes that support logging. We say that code is scattered over the system and is tightly coupled or tangled with the system. If both characteristics, scattering and tangling, apply to a concern we call it a cross-cutting concern.

Aspect-oriented programming is an extension of object-oriented programming that provides a solution for the given problem. Cross-cutting concerns are encapsulated in modules called aspects. Scattering and tangling are hidden in the source code. An aspect weaver is used to integrate cross-cutting concerns into the business logic of the system. Source code is not changed but the compiled or run-time code is modified, depending on the weaving strategy. Aspect-oriented programming provides a non-invasive way of adding new functionality to an implementation without affecting the source code.

An aspect encapsulates method-like code fragments, called advices. To weave new functionality into an existing (adapted) system, join points have to be defined. A join point is a point in the control flow of the adapted system, e.g. a method call, method execution or an access to an instance variable. Important for this paper is also the `cflowbelow` statement that refers to all points in the stack trace below a given control-flow point. Point-cuts are collections of join points. Advice code can be executed before, after or instead of the code addressed by a point-cut, e.g. instead of a referred method execution with the `around` statement. The original method can be called within the `around` advice using the `proceed` statement.

We use aspect-oriented programming techniques for testing object-oriented systems. We regard testing as a cross-cutting concern. Aspect-oriented programming helps to encapsulate test code and provides a flexible test instrumentation solution.

Before presenting some applications of aspect-oriented programming techniques in our unit testing framework *FlexTest* in Section 4, we give a brief introduction to aspect-oriented unit testing in the following section.

3 Aspect-Oriented Unit Testing

This paper investigates the use of aspect-oriented programming techniques for unit testing. We view unit testing as it is defined in the field of extreme programming. The focus of a unit test is a class, "but test messages must be sent to a method, so we can speak of method scope testing" [2]. For each method, the tester must implement independent unit tests. A unit testing framework is generally used to automatically execute and evaluate test cases.

Using aspect-oriented techniques for unit testing assumes that testing is a cross-cutting concern with respect to the implementation under test (IUT). In most cases, testing involves inserting additional test code into the IUT.

By looking at the additional test code and the IUT, we are able to determine that

– We must abstract from the context of the methods under test in the testing phase.
– Methods are dependent on instance variables. In some cases, test code must have privileged access to instance variables to initialize and evaluate test cases.
– Test code must be inserted in certain methods and classes in a similar way.
– Normally, unit test code is removed after the testing phase.

Given these requirements, we can say that testing is a cross-cutting concern with regard to the system under test. Both characteristics of aspects, scattering and tangling, hold for test code.

If, then, testing is a cross-cutting concern, the use of aspect-oriented programming techniques will yield benefits.

The next section shows some applications of aspect-oriented programming in unit testing and presents our unit test framework *FlexTest*.

4 Aspect-Oriented Programming Techniques in *FlexTest*

In this section, we examine some selected problems encountered while unit testing object-oriented systems. First, conventional solutions to a presented problem are shown and then these are compared with the aspect-oriented solution provided by *FlexTest*.

FlexTest is a unit test framework for testing object-oriented programs. It has two parts, an object-oriented and an aspect-oriented part. The object-oriented part is similar to the JUnit test framework [3] and provides classes for defining

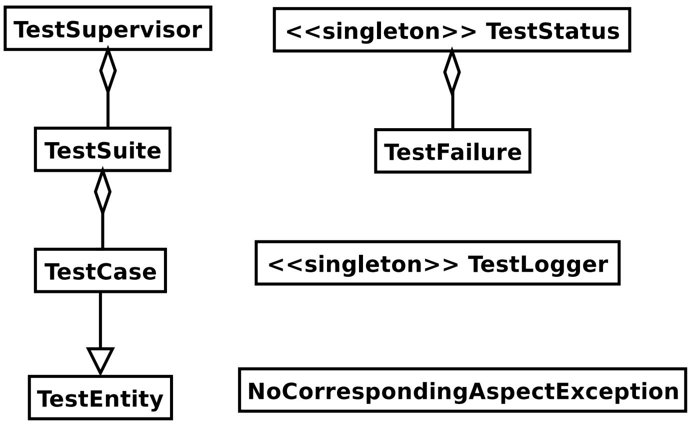


Fig. 1. Class Hierarchy of Flextest

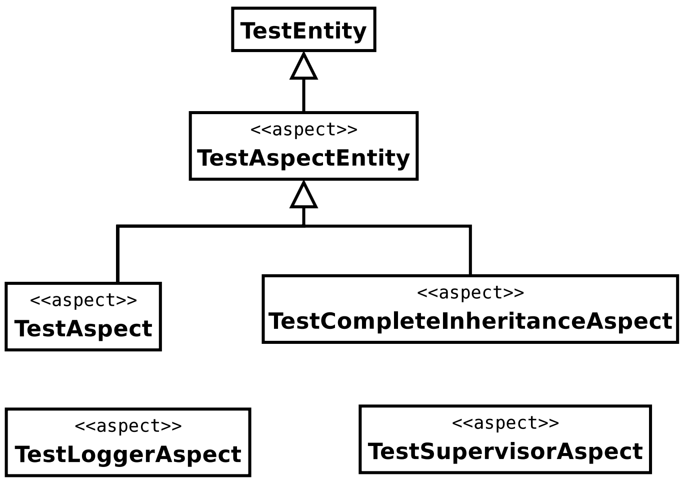


Fig. 2. Aspect Hierarchy of Flextest

test cases and test suites and functionality for test execution (see Fig. 1). Examples of framework classes are `TestCase` and `TestSuite`, similar to JUnit. The aspect-oriented part supports additional functionality for using aspect-oriented techniques in *FlexTest* (see Fig. 2). The main class is the class `TestAspect`. When using the `FlexTest` framework, the class `TestCase` and the aspect `TestAspect` have to be subclassed. Each subclass of `TestAspect` is assigned to a subclass of `TestCase`. When we refer to classes of the implementation under test in the following sections, we annotate them with *(IUT)*, the ones of the object-oriented part of the framework with *(OOF)* and the aspects of the aspect-oriented part with *(AOF)*. Subclasses of framework classes and aspects are annotated like their superentities.

We decided to develop a new framework instead of using an existing framework like JUnit because this allows us to design the object-oriented part depending on the aspect-oriented part and to enhance the aspect-oriented part flexibly with respect to the unit test problems.

In the following subsections, we look at the different problems encountered in unit testing. The aspect-oriented part of the *FlexTest* framework provides aspect-oriented solutions to the given problems, as described below. The IUT is a library of linear-algebra algorithms written in Java. The examples given in the following subsections are written in AspectJ [4].

4.1 Encapsulation

In object-oriented systems, attributes and methods are encapsulated in classes. As a result of encapsulation, classes are only accessible by interfaces. In testing, encapsulation leads to the problem of insufficient access to the IUT. Classes can only be treated as black boxes without access to inner states, which leads to a less precise test oracle. Private methods cannot be tested by the test framework, which is a test driver problem.

Without using external tools, there are two main solutions to this problem:

1. **Changing the code for testing**
   The tester simply replaces all private members in the source code with public members, which can be accessed by the test framework. The disadvantage of this solution is that it means maintaining two versions of the same implementation.
2. **Using language features**
   Some languages support exclusive access. For example, we can define friend classes in C++ that have privileged access to a given class, here the tested class. The disadvantage of this solution is the dependency of the class under test and the test framework class. The class under test cannot be compiled without the friend class. Moreover, the test concern still cross-cuts the application concern.

We propose an aspect-oriented solution to the encapsulation problem. AspectJ, the aspect-oriented language used in *FlexTest*, provides the keyword `privileged` for aspects, which allow privileged access to the adapted classes. In the *FlexTest* framework, privileged aspects are used in addition to test classes of the object-oriented part.

We show the privileged access using result checking as an example. Test aspects select results of test cases via point-cut definitions. To make definitions of point-cuts easier, they should be defined per object. This means each creation of an object causes the creation of a test aspect, whose point-cuts select only join points of the object's class.

Using *FlexTest*, the tester must define a subaspect of the aspect `TestAspect` (AOF) and activate it. This is done in two steps:

1. For each class under test, the tester must define a test case class and an aspect adapting the test case class. The aspect class inherits the abstract aspect `TestAspect` (AOF) and must override the abstract point-cut `initMe` (cf. line 3 in Fig. 3). In Fig. 3, a fragment of the implementation of the aspect `TestVectorDouble4Aspect` (AOF) is shown, adapting the test case class `TestVectorDouble4` (OOF) (see line 1). The aspect `TestVectorDouble4-`

```
1  public privileged aspect TestVectorDouble4Aspect extends TestAspect
2  {
3    public pointcut initMe() :
4        within(TestVectorDouble4);
5     [...]
6  }
```

Fig. 3. Aspect `TestVectorDouble4Aspect` (AOF) supporting class `TestVectorDouble4` (OOF)

`Aspect` (AOF) has privileged access to its adapted class and to the objects of the IUT defined in this class.

2. The test case class is inherited from the abstract class `TestCase` (OOF). The test case class, in the example the class `TestVectorDouble4` (OOF), must call `initTestAspect` to activate the aspect, in the example `TestVector-Double4Aspect` (AOF).

Fig. 4 shows an example of a constructor test. The members of `Vector Double4` (IUT) are encapsulated. They cannot be accessed directly.

```
public void testConstructor()
{
  test1 = new VectorDouble4(1., 2., 3., 4.);
  [...]
}
```

Fig. 4. Constructor test in `TestVectorDouble4` (OOF)

```
1   pointcut vector4ConstructorTest(double nX, double nY,
2                                   double nZ, double nW) :
3   call (VectorDouble4.new(double, double, double, double))
4   && args(nX, nY, nZ, nW);
5
6   after(double fX, double fY, double fZ, double fW)
7     returning(VectorDouble4 newConst) :
8       vector4ConstructorTest(fX, fY, fZ, fW)
9   {
10    check(((newConst.m_fX == fX) && (newConst.m_fY == fY)
11       &&  (newConst.m_fZ == fZ) && (newConst.m_fW== fW)));
12  }
```

Fig. 5. Aspect-oriented result check in `TestVectorDouble4Aspect` (AOF)

To gain access to the members of `VectorDouble4` (IUT), we define a pointcut for the construction of `VectorDouble4` (IUT) objects and check the results within an `after` advice (see Fig. 5, lines 3 and 6-8, respectively).

4.2 Quantification of Test Cases

A popular definition of aspect-oriented programming (AOP) [5] states that "AOP is quantification and obliviousness". In this section, we examine how quantification can be used to improve the test process.

Testing can be a tedious task. It involves writing lots of test cases. Often, test cases can be checked in a similar way, because they return similar results.

```
1   public void testIsNormalPositive()
2   {
3     VectorDouble4 vec2 = new VectorDouble4(0, 1, 0, 0);
4     vec2.isNormal();
5     VectorDouble4 vec3= new VectorDouble4(1, 0, 0, 0);
6     vec3.isNormal();
7     [...]
8   }
9
10  public void testIsNormalNegative()
11  {
12    VectorDouble4 vec = new VectorDouble4(3, 2, 1, 5);
13    vec.isNormal();
14    [...]
15  }
```

Fig. 6. `isNormal` calls in TestVectorDouble4 (OOF)

AOP quantification can express conditions that must hold for a set of method calls in a certain test case. Grouping test cases gives them a context, which allows an AOP language to check results of method calls in one advice.

In Fig. 6, the method `isNormal` is tested returning `true` if a vector is normal and `false` otherwise. We define two contexts. In the first context, `testIsNormalPositive`, we expect all method calls to return `true` (lines 1-8). The second context groups all method calls returning `false` (lines 10-15).

The results are selected by a point-cut defined using the `cflowbelow` statement to select the context (see Fig. 7, lines 2-4).

Above, we describe the use of aspect-oriented programming as test oracle, but it can also serve as test driver. Experiments have shown that other separations of responsibilities are possible using the `FlexTest` framework. For example, we can encapsulate all calls of `isNormal` in an aspect instead of implementing them in the test case class. The next section shows another example the use of aspects as a test driver.

```
1   [...]
2   pointcut vectorDouble4PositiveIsNormalTest() :
3     call (* VectorDouble4.isNormal())
4     && cflowbelow(call(* TestVectorDouble4.testIsNormalPositive()));
5
6   after() returning(boolean retValue) :
7     vectorDouble4PositiveIsNormalTest()
8   {
9     if (!retValue) makeError(thisJoinPoint);
10  }
```

Fig. 7. Testing the results of `isNormal` calls with the expected result `true`

4.3 Testing Object Hierarchies

In [6], subtyping is defined as follows: "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

Type hierarchies in which subtype relations confirm to this criteria are called Liskov-conform.

To check classes for Liskov-conformity, we have to run superclass test cases on all subclasses, as proposed in [2]. Liskov-conform subclasses should pass all test cases of their superclasses.

Designing test cases for subclasses raises the problem of considering test cases of all superclasses. We can say that the concern of testing for Liskov-conformity interferes with the concern of testing a single class.

Our solution encapsulates the object hierarchy test in one aspect. Thus, the concern of testing for Liskov-conformity is separated from the concern of testing a single subclass. New classes can be added to the hierarchy without changing the hierarchy test module. Our solution selects the relevant test suites for each class and the relevant classes for each test suite itself.

As mentioned earlier, AOP can be understood as quantification and obliviousness. For this task, we need quantification to select the classes under test. This selection must occur without the help of individual test classes and aspects. Thus, the aspect-oriented part of the framework must be more active than in the previous sections. It should not only be triggered to check the results of certain test cases but acts as a test driver that is responsible for running a test suite on a hierarchy of classes. The first instantiation of an object of the superclass or one of its subclasses is intercepted to perform the superclass test suite.

```
1   void testToString()
2   {
3     String strCompare = new String("");
4     for (int i = 0; i < _vecExample.getNoOfElements(); i++)
5     {
6       strCompare += _vecExample.getElement(i);
7     }
8     check(strCompare.equals(_vecExample.toString()));
9   }
```

Fig. 8. Testing the `toString()`-method for all sub-classes of `Vector` (IUT)

For example, if we wish to test the method `toString` of all `Vector` (IUT) subclasses we must apply the method `testToString`, given in Fig. 8, to all subclasses of `Vector` (IUT). The object `_vecExample` (lines 4, 6, 8) is the actual object under test.

```
1 public abstract aspect TestCompleteInheritanceAspect
2   extends TestAspectEntity percflow(newInstance())
3 {
4   public pointcut newInstance() :
5     call (* TestSuite.run());
6   [...]
7 }
```

Fig. 9. Point-cut selecting join point to create a new instance of a hierarchy test aspect

```
1 public aspect TestVectorAspect extends TestCompleteInheritanceAspect
2 {
3   public pointcut newSubclass():
4     call (Vector+.new()) && within (TestVector*);
5   [...]
6 }
```

Fig. 10. Point-cut selecting the creation of a new class under test

All hierarchy test aspects in `FlexTest` are subaspects of `TestCompleteInheritanceAspect` (AOF). An instance of this subaspect is assigned to each run of a test suite. This is achieved by the code in Fig. 9 (lines 4+5).

We must now specify which subclasses should be tested by the hierarchy test aspect. This is done by overriding the abstract point-cut `newSubclass` (line 3 in Fig. 10), so that it selects the creation of certain subclasses in a test context.

Creations of subclass objects of `Vector` (IUT) that are located in classes of a name starting with `TestVector` (OOF) are addressed by the point-cut in Fig. 10 (lines 3+4). `TestCompleteInheritanceAspect` (AOF) subaspects memorize which classes they have already tested. Each class is tested only once for each run of the test suite.

4.4 Convenience Functionality

This section covers issues concerning the organization of a test framework and its context. Some of these can be very neatly implemented using aspect-oriented programming techniques.

Logging. Testing must be documented. A test framework must therefore offer functionality to log the results of test runs following a certain pattern. For example, the execution of all methods that are subclasses of class `TestCase` (OOF) or all results of methods with the name pattern `test*` should be logged. Flexibility is needed in the definition of logging.

Since logging is a classical example of the use of AOP, it is supported by our test framework as well. Instead of scattering the logging calls over the code, they are defined in a central module.

Failure Localization. It is not sufficient for a test framework to merely state that a failure has occurred. A test framework must also locate which class and which source code line have caused the failure.

JUnit [3] throws an exception and analyzes the stack traces for failure localization. In our opinion, this is not a good solution because it abuses the exception mechanism of Java. Exceptions should be thrown when a routine does not know how to handle a situation and passes the responsibility to its caller. A failed test case is normal behaviour in a test framework and using exceptions conflicts with the application's exceptions. This can lead to confusion with exceptions originating from the application under test or with exceptions thrown in the test framework.

The straight modular AOP variant is used to define a point-cut for all methods that check conditions for the test suite and create error strings in their advices. These advices are able to reason about the point-cut shadow and even know the line of code.

4.5 Other Applications

Finally, we present two applications of AOP in the test framework that do not specifically focus on testing. Assertions are part of quality assurance, and mock objects are needed in unit testing in some situations.

Mock objects. Mock objects [7] mimic objects without having much functionality themselves. Their purpose is to check if the object under test communicates in the right way with objects of the mimicked type. The use of mock objects may be necessary, if the simulated object is not available at the time of testing or if it takes to much time to test with all functionality of the mimicked object. Unlike to stubs, mock objects are not just the original classes stripped of functionality. They must provide some means of determining if other objects communicate with the mimicked object in the right way.

Aspects can be mock objects without interfering with the code of the class under test as follows:

– Each mock object is implemented by one aspect.
– The aspect contains a point-cut for each method of the mimicked class. Each point-cut selects the call of one of its methods in the test context.
– The point-cuts are advised by `around` advices without a `proceed` statement.
– The mock object advice checks if it is called at the right time in the right state with the right parameters. Otherwise, it logs an error.

The aspect-oriented mock object approach is especially useful if the insertion of the mock object into a context is difficult. This can be the case if the object to be mocked is, for example, determined in the method under test so that the testing class is not able to influence this object or replace it with a mock object. [8] discusses this problem at greater length.

Assertions. Assertions [2] involving pre- and postconditions are easy to implement in object-oriented programming languages at the beginning and the end of each method. Class invariants, however, are hard to implement. They must hold before and after methods of the actual object are called from other objects. They are therefore scattered across the code if implemented conventionally.

Points at which the class invariant must hold are potential join points in most AOP languages. Thus, inserting class invariants is a typical application for aspect-oriented programming. An aspect selects all method calls of a class via point-cuts and checks the class invariant for the class. With the `cflowbelow`-statement, it is even possible to distinguish calls of other objects from those of the actual object for which the invariant does not necessarily have to hold.

5 Related Work

A couple of research papers have investigated testing using aspect-oriented programming techniques. The non-invasive integration of test code into the system under test has been shown to be the main advantage here.

Most of the work in this area focuses on the popular language AspectJ. Although our framework, too, is based on AspectJ, we see advantages in using other aspect-oriented languages or platforms, like *abc* [9] that provides more join point flexibility, or Object Teams that allows us to explicitly activate and deactivate aspects at run-time. There is, however, no known work based on *abc*. An instrumentation technique using Object Teams for integrating state-based test oracles into the system under test is shown in [10]. The implementation of state-based test oracles with AspectJ is shown in [11].

Aspect-oriented programming techniques are also used in [12,13,14,15]. The main focus of these approaches is monitoring run-time behaviour on the system level. Our approach concentrates on unit testing of methods, which means it has a completely different granularity.

[8] proposes integrating mock objects using AspectJ. The work is based on the JUnit test framework and has many similarities with our own work but we have also considered many other applications where aspects can be helpful in testing.

The realization of assertions with aspect-oriented programming techniques is also proposed in [16,17,18]. In all cases, assertions are defined using a specification language (OCL or JML). For example, in [18] JML specifications are transformed into AspectJ code. In our approach, the tester defines assertions and test cases using advices. The advantage here is that the tester does not have to learn a new language in addition to AspectJ. However, directly written advice code is less expressive than a specification language like JML.

In contrast to the presented related work, our approach is not confined to one test application. We have investigated the use of aspect-oriented programming techniques in different applications, resulting in a flexible test framework for unit testing object-oriented systems.

6 Discussion

Our experiments with the *FlexTest* framework have shown that aspect-oriented programming techniques are suitable for unit testing. Developing a new framework instead of using JUnit enables us to enhance the framework flexibly with aspect-oriented features. Our framework provides some solutions to typical unit test problems.

We now go on to suggest some entry points for discussion and indicate a potential direction for further research in this area.

First, we consider some disadvantages of our approach. As an implementation language for the aspect-oriented part of the framework, AspectJ requires recompilation of the whole application with all test aspects. This adds extra time to the compile process. Furthermore, each change of test aspects leads to a recompilation of the whole application. But this is a specific AspectJ problem. Other aspect-oriented languages that support load-time or run-time weaving only require the compilation of test aspects.

Another disadvantage is due to the tester's experience with aspect-oriented programming. A tester who has no experience with AOP cannot use the *FlexTest* framework.

The separation of the `TestCase` class and the `TestAspect` aspect distributes test implementations to different modules. This seems to be a disadvantage, but in our presented examples the separation is clear between test cases (`TestCase`) and test oracle (`TestAspect`), and it naturally fits with mock object implementation. `FlexTest` supports the strict separation ot testing concern and application concern. There is no need for a preparation of the classes under test. Also, subconcerns are separated by the use of AOP:

– Logging and failure localization are separated from testing.
– The concern of testing for Liskov-conformity is separated from the concern of testing each individual class.

We believe that this kind of modularity helps in managing the complexity in large test suites.

FlexTest provides solutions to some common test problems encountered when unit testing object-oriented applications, like encapsulation bypassing. Aspect-oriented programming provides a flexible instrumentation solution that can be easily extended to other instrumentation problems. One of the main drawbacks - the lack of instrumentation support per source-code statement - is compensated for by the flexible and entensible join point language provided by the *abc* compiler [9]. The *abc* compiler allows us to enhance the join point language so it even supports instrumentation per statement.

Owing the modular structure of the *FlexTest* framework, it is easy to customize the logging and failure localization functionality for test suites. To log in a different manner or change the failure localization feature, one merely has to edit few lines in a central aspect.

The pros and cons of our approach show that the main problem is the difficulty of expressing the functionality in an AOP language in an easy understandable manner. The complex expressions to select test method calls should be replaced by expressions that are easier to read and understand. This means that we have to change the AOP language.

Future work involves considering the following issues:

– Comparing the FlexTest framework with an object-oriented framework like JUnit in a case study.
– Defining new keywords for use in test situations so that point-cut selections are easier to read.
– Extending the point-cut language to a point where it is possible to check loop invariants.
– Integrating the test concern into a language in which test classes and test aspects are more closely related.

The new AOP language could be based on the extensible compiler *abc*, which implements full AspectJ support but provides a flexible join point language.

References

1. Vösgen, M.: FlexTest Framework. `http://swt.cs.tu-berlin.de/ mvoesgen/ Stuff/flextest.zip`(2005)
2. Binder, R.V.: Testing Object-Oriented Systems. Object Technology Series. Addison-Wesley (1999)
3. JUnit: JUnit-Homepage. http://www.junit.org (2005)
4. AspectJ: AspectJ-Homepage. http://www.aspectj.org (2005)
5. Filman, R.E., Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness. In: Workshop on Advanced Separation of Concerns, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota, USA (2000)
6. Liskov, B.: Data Abstraction and Hierarchy. In: Addendum to the proceedings on Object-oriented programming systems, languages and applications, Orlando, Florida, USA (1987) 17 – 34
7. Mackinnon, T., Freeman, S., Craig, P.: EndoTesting: Unit Testing with Mock Objects. In: eXtreme Programming and Flexible Processes in Software Engineering (XP), Cagliari, Italy (2000)
8. Lesiecki, N.: Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/ (2002)
9. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An Extensible AspectJ Compiler. In: International Conference on Aspect-Oriented Software Development (AOSD), Chicago, Illinois, USA (2005)
10. Sokenou, D., Herrmann, S.: Using Object Teams for State-Based Class Testing. Technical report, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, Germany (2004)
11. Bruel, J.M., Araújo, J., Moreira, A., Royer, A.: Using Aspects to Develop Built-In Tests for Components. In: AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML), San Francisco, California, USA (2003)

12. Deters, M., Cytron, R.K.: Introduction of Program Instrumentation using Aspects. In: Workshop of Advanced Separation of Concerns in Object-Oriented Systems, 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM Sigplan Notices, Tampa, Florida, USA (2001)
13. Filman, R.E., Havelund, K.: Source-Code Instrumentation and Quantification of Events. In: Workshop on Foundations of Aspect-Oriented Languages, 1st Intern ational Conference on Aspect-Oriented Software Development (AOSD), Enschede, Netherlands (2002)
14. Low, T.: Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In: Workshop on Aspect-Oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, Netherlands (2002)
15. Mahrenholz, D., Spinczyk, O., Schröder-Preikschat, W.: Program Instrumentation for Debugging and Monitoring with AspectC++. In: Proceedings of The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Crystal City, Virginia, USA (2002)
16. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML), San Francisco, California, USA (2003)
17. Briand, L.C., Dzidek, W., Labiche, Y.: Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technical report, Carlton University, Ottawa, Canada (2004)
18. Xu, G., Yang, Z., Huang, H.: A Basic Model for Aspect-Oriented Unit Testing. www.cs.ecnu.edu.cn/sel/ harryxu/research/ papers/fates04_aspect-oriented

Quality Assurance in Performance: Evaluating Mono Benchmark Results

Tomas Kalibera[1], Lubomir Bulej[1,2], and Petr Tuma[1]

[1] Distributed Systems Research Group, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Malostranske nam. 25, 118 00 Prague, Czech Republic
phone +420-221914267, fax +420-2219143232
{kalibera,bulej,tuma}@nenya.ms.mff.cuni.cz

[2] Institute of Computer Science, Czech Academy of Sciences, Pod Vodarenskou vezi 2, 182 07 Prague, Czech Republic
phone +420-266053831
bulej@cs.cas.cz

Abstract. Performance is an important aspect of software quality. To prevent performance degradation during software development, performance can be monitored and software modifications that damage performance can be reverted or optimized. Regression benchmarking provides means for an automated monitoring of performance, yielding a list of software modifications potentially associated with performance changes. We focus on locating individual modifications as causes of individual performance changes and present three methods that help narrow down the list of modifications potentially associated with a performance change. We illustrate the entire process on a real world project.

1 Introduction

The ever-increasing amount of available computing power is closely followed by increasing scale and complexity of software systems, which in turn causes increase in the size of development teams producing the software. For various reasons, be it a distributed development model, which makes communication and coordination difficult, or the Extreme Programming [1] approach, which prefers rewriting code and rigorous testing to detailed analysis of all possible requirements, the development of software puts more and more emphasis on quality assurance.

Both the distributed development and the Extreme Programming approaches depend on regular testing of all components of the developed software, which is often automated and performed either on a regular basis, such as every day, every week, or as soon as a change is introduced into the source code management system. This testing process is known as regression testing.

The current practice typically limits the testing to functional correctness and robustness of the code and neglects an important aspect of quality, which is performance. Regression benchmarking fills the gap by extending the regression testing practice with automatic benchmarking and evaluation of software performance [2,3].

Regression benchmarking requires the testing process to be fully automatic, which includes downloading and building the software, and executing the benchmarks in a robust environment. The execution environment must be able to handle most of the typical failure scenarios to allow running without systematic supervision. These requirements alone constitute a technical challenge, if only because the executed software is under development, where deadlocks, crashes, or infinite loops can be expected, but their occurrence cannot be easily predicted. Often, the same program repeatedly executed under (to a maximum possible and reasonable extent) identical conditions finishes with success in one case and with failure in another.

Many of these problems have been already solved for the purpose of regression testing, yet in case of regression benchmarking, the goal is also to minimize the influence of undesirable effects on the benchmark results. During benchmark execution, there should be no other users of the system, minimum number of concurrently running system services, minimum or preferably no network communication unrelated to the benchmark, and no substantial changes to the configuration of the system. Also, the monitoring of benchmarks must not be too intrusive, etc.

We have elaborated the requirements and described the design of a platform independent environment for automatic execution of benchmarks in [4]. At present, the benchmarking environment is under development.

As an experimental testbed, we have created a simple system for regression benchmarking of Mono [5], which is being developed by Novell as an open source implementation of Common Language Infrastructure specification [6], known as the .Net platform. The Mono implementation of CLI comprises a C# compiler, a virtual machine interpreting the Common Intermediate Language instructions, and the implementation of runtime classes.

Since August 2004, the system monitors the performance of daily development snapshots of Mono on the Linux/IA32 platform, using four benchmarks focused at .Net remoting and numeric computing. The results are updated on daily basis and publicly available on the web of the project [7]. The currently used benchmarks do not cover all the functionality of Mono, but rather serve as a test bed for developing methods for detecting performance changes and locating their causes.

Compared to the envisioned generic benchmarking environment, the system for benchmarking Mono is less universal, depends on the Unix/Linux environment, and is not distributed. Nevertheless, its operation is fully automatic and provides experience which is used to drive the design and implementation of the environment described in [4].

Software projects that employ regular and automated benchmarking, such as [8] or [9], typically focus on tracking the performance of the particular project and lack the automatic detection of performance changes. The automatic detection of performance changes is also missing in other related projects, such as the generic framework for automated testing [10] and the framework for execution and advanced analysis of functionality tests [11]. The foundations of a generic environment for automated benchmarking in grids are implemented in a discontinued project [12]. We are not aware of any other project that would support automatic detection of performance changes and location of their causes.

The rest of the paper has the following structure. Section 2 describes the methods for automatically detecting performance changes, while Section 3 deals with identifying the source code modifications causing the performance changes and provides an analysis of regressions identified by the regression testing system. Section 4 concludes the paper and provides an outlook on future work.

2 Automatic Detection of Regressions

Regression benchmarking requires automatic analysis of benchmark results to discover performance changes, be it performance regressions or improvements. The complexity of contemporary platforms and software causes the durations of operations measured by a benchmark to differ each time the operations are executed, making it impossible to discover performance changes simply by comparing the durations of the same operations in consecutive versions of software. The differences in operation durations often exhibit random character, which can originate for example in the physical processes of hardware initialization, or in the intentionally randomized algorithms such as generators of unique identifiers and subsequent hashing.

Typically, the operations measured by a benchmark are therefore repeated multiple times and the durations are averaged. The precision of such a result can be determined if the the durations are independent identically distributed random variables. Unfortunately, the requirements of independence and identical distribution are often violated. In Section 2.2, we describe several ways of processing the benchmark data which help to satisfy the requirements without distorting the results.

Another frequently overlooked effect of the complexity of contemporary platforms and software is the influence of random initial conditions on the durations of operations measured by a benchmark [13]. It is generally impossible to discover performance changes even by comparing the averaged durations of the same operations in consecutive versions of software, because the averaged durations differ each time the benchmark is executed. The difference is not related to the number of samples and therefore cannot be overcome by increasing the number of samples.

Regression benchmarking thus requires not only repeating the operations measured by the benchmark within the benchmark execution, but also repeating the execution of the benchmark within the benchmark experiment. The precision of the averaged durations can then be calculated as outlined in Section 2.1. Each execution of a benchmark during a benchmark experiment, however, increases the overall cost of the experiment. That is mainly due to initialization and warm-up phases of a benchmark, which are costly in terms of time but collect no data. During the benchmark warm-up, operation durations can be influenced by transitory effects such as initialization of the tested software, operating system or hardware. Ignoring this fact in analysis can lead to incorrect results, as shown in [14].

The number of samples in a benchmark run and the number of benchmark executions during a benchmark experiment both contribute to the precision of the result of a benchmark experiment. The contribution of each of the components depends on the character of the developed software. Because the most costly factor of a benchmark run is the initialization and warm-up, our objective is to minimize the number of benchmark runs and maximize the number of samples collected in each run. From this naturally follows the incentive to determine the optimal number of samples that should be collected in a single benchmark run and beyond which increasing the number of samples does not contribute to the precision of the result anymore. After that, as explained in Section 2.1, we only need to determine the number of benchmark runs required to achieve the desired precision.

Knowing the precision of the result of a benchmark experiment is important so that a comparison of results that differ by less than their respective precision is not interpreted as a change in performance. Detection of performance changes in regression benchmarking can be carried out using the approach described in Section 2.3.

Digressing somewhat from the outlined approach of executing multiple measurements and collecting multiple samples, we can also consider reducing the variance of the samples by modifying the benchmark experiment in ways that remove the sources of variance. Intuitively, things such as device interrupts, scheduler events and background processes are all potential sources of variance that could be disabled, minimized or stopped. The sources of variance, however, may be not only difficult to identify [15,16], but also inherent to the benchmark experiment and therefore impossible to remove. Coupled with the fact that the sources of variance would have to be identified and removed individually for each benchmark experiment, this suggests that sufficiently reducing the variance of the samples by modifying the benchmark experiment is generally impossible. For illustration, a checklist of precautions to be taken to minimize variance in microbenchmarking in FreeBSD is given in [17]. The complexity of these precautions demonstrates the infeasibility of this approach.

It should also be pointed out that to verify whether the sources of variance were removed, executing multiple measurements and collecting multiple samples is necessary anyway. Finally, the relevance of the modified benchmark experiment to practice, where the sources of variance are present, may be questionable.

2.1 Statistics Behind the Scenes

We presume that the durations of operations measured by a benchmark in a run are random, independent and identically distributed, that the distributions of the durations from different runs can differ in parameters, and that the mean values of the distributions from different runs are independent identically distributed random variables. The benchmark result is the average of all measured operation durations from all runs. We consider the benchmark precision to be the confidence with which the result estimates the mean value of the distribution. Specifically, we define the precision as the half width of the confidence interval for the mean, for a given fixed confidence level.

For each $j = 1..m$ as a benchmark run with $i = 1..n$ measurements, the random durations of operations R_{ji}, for $i = 1..n$ and fixed j, are independent identically distributed with a distribution that is described by the two traditional parameters: the mean and the variance. The conditional mean and variance of the distribution are $E(R_{j1}|\mu_j, \sigma_j^2) = \mu_j < \infty$, $var(R_{j1}|\mu_j, \sigma_j^2) = \sigma_j^2 < \infty$.

The means μ_j are independent identically distributed random variables for each $j = 1..m$ as a benchmark run, $E(\mu_1) = \mu < \infty$, $var(\mu_1) = \rho^2 < \infty$.

The result of a benchmark experiment is

$$\overline{R}_{ji} = \frac{1}{mn} \sum_{j=1}^{m} \sum_{i=1}^{n} R_{ji}$$

as an estimate of μ. Note that μ is also mean of R_{ji}, if we do not know the specific μ_j, since

$$E\left(R_{ji}\right) = E\left(E\left(R_{ji}|\mu_j\right)\right) = \mu.$$

From CLT, the distribution of $\overline{\mu}_j$ as an estimate of μ is asymptotically normal:

$$\frac{1}{m} \sum_{j=1}^{m} \mu_j = \overline{\mu}_j \sim N\left(\mu, \frac{\rho^2}{m}\right). \tag{1}$$

From CLT, the average M_j of operation durations from run j has asymptotically the normal distribution:

$$M_j = \frac{1}{n} \sum_{i=1}^{n} R_{ji}|\mu_j \sim N\left(\mu_j, \frac{\sigma_j^2}{n}\right).$$

From the properties of the normal distribution, it follows that

$$\frac{1}{m} \sum_{j=1}^{m} M_j = \overline{M}_j; \overline{M}_j|\overline{\mu}_j \sim N\left(\overline{\mu}_j, \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}\right). \tag{2}$$

It can be shown that from (1), (2) and the known fact that the convolution of Gaussians is again a Gaussian, it follows that:

$$\overline{M}_j \sim N\left(\mu, \frac{\rho^2}{m} + \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}\right). \tag{3}$$

The confidence interval for the estimate of μ can be constructed from (3). The result of a benchmark experiment therefore is $\overline{R}_{ji} = \overline{M}_j$ and with the probability $1 - \alpha$, the precision *pr* of the result value is

$$pr = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{\rho^2}{m} + \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}}, \qquad (4)$$

where u are quantiles of the standard normal distribution.

This result holds asymptotically for sufficiently large n and sufficiently large m, needed for CLT to apply. The unknown variance ρ^2 of the means μ_j can be approximated by the variance of the averages of samples from individual runs, which can be estimated using the S^2 estimate:

$$S_\rho^2 = \frac{1}{m-1} \sum_{j=1}^{m} \left[\left(\frac{1}{n} \sum_{i=1}^{n} R_{ji} \right) - \overline{R}_{ji} \right]^2 .$$

The variance of the samples in a run σ_j^2 is still unknown. If the variance of the individual runs was the same, $\sigma_j^2 = \sigma^2$, we could estimate it as:

$$S_\sigma^2 = \frac{1}{m(n-1)} \sum_{j=1}^{m} \sum_{i=1}^{n} \left(R_{ji} - \frac{1}{n} \sum_{i=1}^{n} R_{ji} \right)^2$$

to get the precision of the result value:

$$pr_\sigma = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{nS_\rho^2 + S_\sigma^2}{mn}}.$$

If we know the maximum variance σ_{max}^2 of the samples in a run, we can use a similar approach and estimate the upper bound of the precision pr (lowest possible precision).

Note also that the formula for pr does not rely on the individual values of variance, but only on the average variance. We can therefore use the formulas for the precision pr_σ and the variance estimate S_σ^2 for large m, as implied by the Weak Law of Large Numbers, except for the precision of the estimate itself, which is not included in the formula for pr_σ.

From (4) it follows that increasing the number of runs m always improves the precision of the result, while increasing the number of measurements in a run n improves the precision only to a certain limit. In practice, each benchmark run has to invoke the measured operation w times discarding the results to warm-up, prior to measuring n operation durations. Usually there is only a limited time c (cost) for each experiment, which can be expressed as the number of all invoked operations: $c = (w+n)m$. The question is how to choose m and n to achieve the best precision pr for a given c. From (4) we can derive that the optimal number of (non-warmup) measurements in a benchmark run is

$$n_{opt} = \sqrt{\frac{wS_\sigma^2}{S_\rho^2}}.$$

The optimal strategy to achieve the desired precision of the result is to first increase n up to n_{opt}, and only then increase m. However, determining n_{opt}

requires the knowledge of the estimates S_σ^2 and S_ρ^2. An experiment can therefore be split into two parts – in the first part, a sufficient number of runs and number of measurements in a run are chosen to get good estimates S_σ^2 and S_ρ^2. The benchmark precision is also calculated, and when it is not sufficient, n_{opt} is calculated using the variance estimates and a second part of the experiment is performed which only invokes n_{opt} operations in each run.

2.2 Handling the Auto-dependence and Outliers

When determining the result of a benchmark experiment and its precision as described in Section 2.1, the key assumption is that the samples of the durations of operations in a single benchmark run are independent and identically distributed. Our experience suggests that these assumptions do not generally hold. If the benchmark data is found to violate the assumptions, a simple transformation can be often found that will remedy the situation without significant impact on the results.

The assumption of independence can be easily verified using lag plots. The lag plot is generally used for visually inspecting auto-dependence in time series by plotting the original data against lagged version of themselves to check for any identifiable structure in the plot. For example, in one of the benchmarks used by the Mono regression benchmarking system, the data contained four clearly visible clusters. Inspection using lag plot suggested that the values from the individual clusters were systematically interleaved. We have numbered the clusters and transformed the original data so that each value was replaced by the number of the cluster it belonged to, which made the interleaving pattern in the data obvious and confirmed the suspected violation of the independence assumption.

We solve the auto-dependence problem by resampling the original data. For each benchmark run, we generate a reduced subset from the original data using sampling with replacement. The use of sampling with replacement is important, because it eliminates a potential dependency on the number of samples in case of systematic interleaving of values from different clusters. From the resampled data, we can calculate the estimates of mean and the variance of the distribution of the values collected by the benchmark, as outlined in Section 2.1.

The assumption that the samples come from the same distribution is typically violated in presence of outliers in the collected data. In benchmarking, the nature of the outliers is such that under certain circumstances, the duration of an operation can be as much as several orders of magnitude longer than in most other cases. The exact circumstances leading to the occurrence of outliers are difficult to identify, but some of the outliers can have a plausible explanation [3]. Even so, explicitly removing the outliers from the data is typically impossible, because we do not have enough information to discriminate between valid data and outliers.

The outliers can significantly influence the results, especially when they are based on sample average or variance. The preferred solution is to use robust statistics such as the median or median absolute deviation in place of the traditional but fragile sample average or sample variance [14,3]. The main drawback of using robust statistics lies with the fact that the analysis of their precision is difficult compared to sample average or variance.

To handle outliers and auto-dependence in the benchmark data from the Mono project, we use a combination of resampling and robust statistics, but still report sample average or variance as the result of a benchmark run, which makes the analysis of the precision easier.

If there are any outliers in the original data, they will likely be also in the resampled subset, which means that the sample average or variance will be influenced by the outliers. We therefore generate a high number of reduced subsets from the original data, and compute the required estimators using the generated subsets. From the resulting set of sample averages or sample variances, we select the median to obtain a single estimate of the mean or the variance from a single benchmark run. These values still have the nature of the original sample mean or variance, because they were computed using the data from one of the subsets.

2.3 Detecting and Quantifying the Changes

Detecting changes in performance between different version of the software in development requires comparing benchmark results for the two versions. As outlined in Section 2.1, processing the benchmark data in order to carry out the comparison is not so trivial a task.

Comparing performance of two versions of the same software requires comparing the mean values of the respective distributions of samples obtained by running the benchmarks. Since the true means are unknown, the comparison has to be carried out on their estimates and take into account the precision of the estimates. Benchmarks provide the estimate of the distribution mean as a grand average of average durations of an operation execution in multiple benchmark runs and the precision of the estimate as a confidence interval for the grand average.

When using confidence intervals to compare unpaired samples from the Normal distribution, Jain [18] suggests a method which considers the distribution means different when the confidence intervals of sample averages do not overlap, and equal when the center of either of the confidence intervals falls in the other. In the last case, when the confidence intervals overlap, the decision is based on the t-test.

For the purpose of regression benchmarking, the algorithm for detecting changes in performance must avoid or minimize the amount of false alarms, otherwise the whole system would be useless. Therefore, our system only reports performance change if the confidence intervals do not overlap.

The developers, as the users of the regression benchmarking system would be mainly interested in performance regressions resulting from an inappropriate modification to the source code. Not all the changes in performance are caused by inappropriate modifications though.

Changing the name of a variable or an identifier may change the binary layout of the executable, thereby changing the layout of the executable in memory, which may result in a performance change [16]. While not necessarily true in general, the cosmetic code modifications such as the renaming of identifiers can be assumed to have smaller impact on performance than a modification introducing a real performance regression into the code.

In addition to detecting a performance change, it is therefore desirable to assess the magnitude of the change, so that the developers can focus on performance regressions they consider important. We report the magnitude of a performance change as a ratio of the distance between the centers of the confidence intervals for the older and the newer version and the center of the confidence interval for the older version.

2.4 Automatically Detected Regressions

The method for detecting performance changes has been applied on the results of TCP Ping, HTTP Ping, FFT Scimark and Scimark benchmarks compiled and executed under Mono.

The TCP Ping and HTTP Ping benchmarks measure the duration of a remote method execution using .Net remoting, the client and server being different processes running on the same machine. The remote method is invoked with a string argument, which it returns unchanged. The time is measured at the client side and includes the time spent processing the request on the server side. The TCP Ping uses the TCP Channels, while the HTTP Ping uses the HTTP Channels of the .NET remoting infrastructure.

The Scimark benchmark is the C# version of the Scimark2 numerical benchmark [19,20]. The benchmark evaluates the performance of several numerical algorithms, such such as Fast Fourier Transform, Jacobi Successive Over-Relaxation, Monte Carlo Integration, Sparse Matrix Multiplication, and Dense Matrix Factorization.

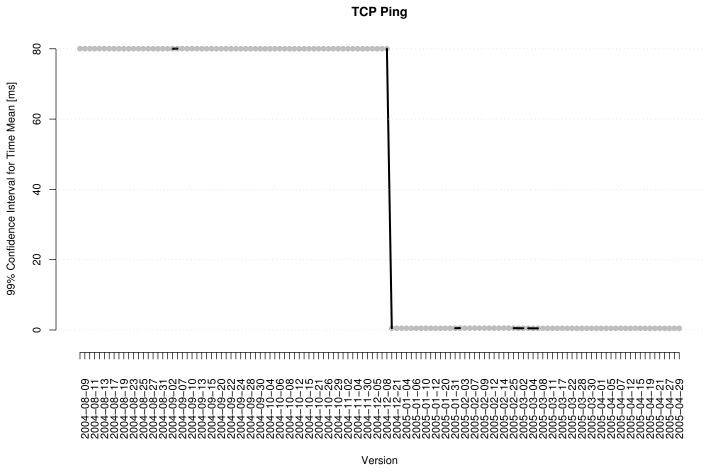
The original benchmark does not measure the computations repeatedly, which means that the results come from the warm-up phase of a benchmark and can be influenced by initialization noise [14]. Based on the original benchmark, we have created the FFT benchmark, which only contains the code for computing the Fast Fourier Transform. The modified benchmark performs the execution repeatedly and can provide data collected after the warm-up phase has passed, which is more suitable for regression benchmarking.

All benchmarks are run on two configurations of Mono. The first configuration has only the default optimizations turned on in the just-in-time compiler, the other has all optimizations turned on. The performance has been monitored from August 2004 till April 2005.

Within that period, the regression benchmarking system has found a number of performance changes. The most notable change was a 99% improvement in performance of the TCP Ping benchmark, which occurred in a development version of Mono from December 20, 2004. The performance changes in the TCP Ping benchmark are shown in Figure 1 and summarized in the table beside the plot.

The detected performance changes are marked by bold black lines, confidence intervals are marked by grey lines. The table shows the magnitude of the changes, along with dates of versions between which a performance change occurred (positive number represents an increase in the duration of a method and thus a performance regression).

A number of significant performance changes has been identified using the HTTP Ping benchmark and the FFT benchmark, see Figure 2 and Figure 3 respectively. The modifications of source code, which cause some of the more significant changes are analyzed in Section 3.



Newer Version	Older Version	Change
2004-09-03	2004-09-02	0.03%
2004-12-20	2004-12-08	-99.37%
2005-02-02	2005-01-31	6.53%
2005-02-28	2005-02-25	-5.02%
2005-03-02	2005-02-28	-2.77%
2005-03-04	2005-03-03	-4.87%
2005-03-07	2005-03-04	2.38%

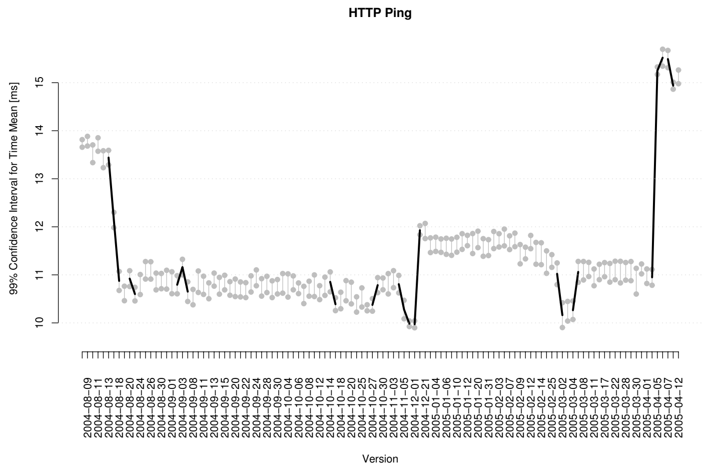
Fig. 1. Changes detected in TCP Ping mean response time

3 Analysis of Discovered Regressions

Regression benchmarking uses the method for detecting changes in performance from Section 2 on daily versions of software subject to modifications to yield a list of versions of software where the performance changes first exhibited themselves. Performance changes are either due to modifications that were done with the intent of improving performance, or due to modifications that were done for other reasons and changed performance inadvertently. In the former case, the modifications that caused the performance changes are known and the list can merely confirm them. In the latter case, the modifications that caused the performance changes are not known and the list can help locate them.

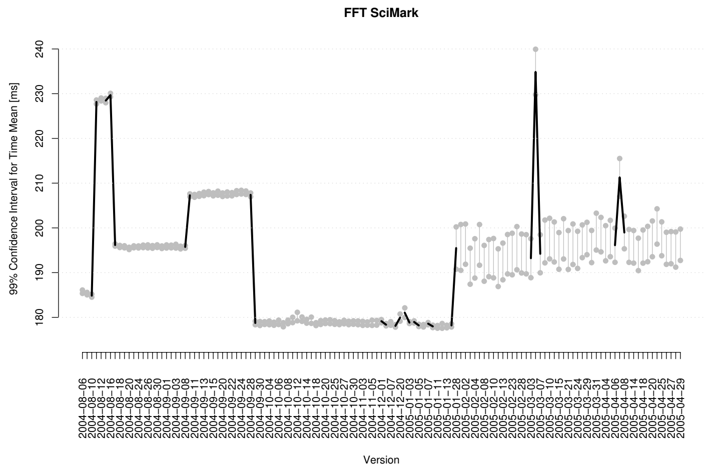
It is necessary to correlate the performance changes with the modifications, so that the modifications can be reviewed to determine whether to revert them, to optimize them, or to keep them. Locating a modification that caused a performance change is, however, difficult in general.

An extreme example of a modification that is difficult to locate is a change of an identifier that leads to a change of the process memory layout, which leads



Newer Version	Older Version	Change
2004-08-17	2004-08-13	-9.67%
2004-08-18	2004-08-17	-10.44%
2004-08-23	2004-08-20	-2.95%
2004-09-03	2004-09-02	3.4%
2004-09-07	2004-09-03	-4.57%
2004-10-15	2004-10-14	-4.29%
2004-10-29	2004-10-27	3.96%
2004-11-05	2004-11-04	-4.9%
2004-11-30	2004-11-05	-2.87%
2004-12-20	2004-12-01	19.64%
2005-03-02	2005-02-28	-7.81%
2005-03-07	2005-03-04	7.77%
2005-04-05	2005-04-04	39.29%
2005-04-06	2005-04-05	1.77%
2005-04-08	2005-04-07	-3.58%

Fig. 2. Changes detected in HTTP Ping mean response time



Newer Version	Older Version	Change
2004-08-11	2004-08-10	23.43%
2004-08-16	2004-08-13	0.53%
2004-08-17	2004-08-16	-14.59%
2004-09-10	2004-09-08	5.9%
2004-09-29	2004-09-28	-13.81%
2004-12-05	2004-12-01	-0.44%
2004-12-20	2004-12-08	1.03%
2005-01-03	2004-12-21	-1.22%
2005-01-05	2005-01-04	-0.43%
2005-01-10	2005-01-07	-0.34%
2005-01-28	2005-01-20	9.71%
2005-03-04	2005-03-03	21.55%
2005-03-07	2005-03-04	-17.31%
2005-04-07	2005-04-06	7.73%
2005-04-08	2005-04-07	-5.83%

Fig. 3. Changes detected in FFT mean response time

to a change in the number of cache collisions, causing a significant performance change [16]. Locating such a modification is not only difficult, but also useless, because the performance change will be tied to a specific platform and a specific benchmark and not reliably reversible.

Locating a modification that caused a performance change is also difficult in large and quickly evolving software projects, where the number of benchmarks that cover various features of the software will range in tens or hundreds, and where the number of modifications between consecutive versions will be high.

Achieving a fully automated correlation of the performance changes with the modifications seems unlikely. Given that the correlation is indispensable for practical usability of regression benchmarking, we focus on devising methods that aid in a partially automated correlation. Specific methods that narrow down the list of modifications potentially associated with a performance change are described in sections 3.1, 3.2 and 3.3.

3.1 Modifications as Differences in Sources

An obvious starting point for correlating a performance change with a modification is the list of modifications between the version of software where the performance change first exhibited itself and the immediately preceding version. This list is readily available as an output of version control tools such as Concurrent Versions System or Subversion or text comparison utilities such as diff.

The list of modifications between consecutive versions is often large. Figure 4 shows the size of modifications between consecutive versions of Mono, with performance changes detected using the method from Section 2 denoted by triangles pointing upwards for regressions and downwards for improvements. The size of modifications is calculated as a sum of added, deleted and changed source lines. The scale of the graph is logarithmic, namely each displayed value is the decadic logarithm of the modification size plus one.

The version control tools or text comparison utilities typically output a list of physical modifications to source files and blocks of source lines. The knowledge of physical modifications is less useful as it puts together multiple logically unrelated modifications that were done in parallel. More useful is the knowledge of logical modifications as groups of physical modifications done with a specific intent, such as adding a feature or fixing a bug. The knowledge of the logical modifications, ideally with their intent, helps correlating the performance changes with the modifications.

The knowledge of logical modifications can be distilled from the output of version control tools when the versions are annotated by a change log. A change log entry typically contains a description of the intent of modifications. Given that a change log entry is associated with a commit of a new version, this requires maintaining a policy of a separate commit for each logical modification.

3.2 Statically Tracking Modifications

In large and quickly evolving software projects, the number of modifications between consecutive versions will be high, even when logical rather than physical modifications are considered. To further narrow down the list of modifications potentially associated with a performance change, we can use the fact that a benchmark that covers a specific feature of a software typically uses only the part of the software that provides the feature. In general, this allows us to consider only the modifications that can influence the result of the benchmark. Given that determining whether a modification can influence the result of a benchmark is difficult, we consider only the modifications that influence the part of the software used by the benchmark instead.

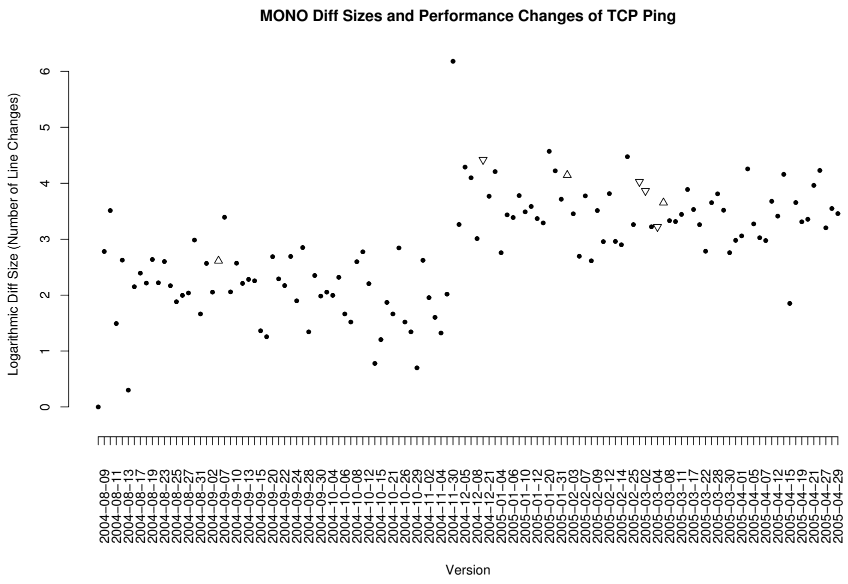


Fig. 4. Diff sizes of all sources with performance changes detected by TCP Ping benchmark

Determining the modifications that influence the part of a software used by a benchmark begins by obtaining the list of the called functions. This can be done either by analyzing the sources of the benchmark and the software, or by running the benchmark and collecting the list of the called functions by a debugger or a profiler. The called functions are then associated with the source files that implement them, and only the modifications that touch these source files are considered.

Alternatively, only the modifications that touch the called functions could be considered, which would further narrow down the list of modifications potentially associated with a performance change. Applying this method in case of Mono is complicated, because the result of a benchmark depends not only on the application libraries, but also on the virtual machine and the compiler, which are subject to modifications. Obtaining the list of called functions is easiest for the application libraries, where the tracing function of the virtual machine can be used. Obtaining the same list for the virtual machine and the compiler is more difficult, because a debugger or a profiler has to be used. This makes regression benchmarking more time consuming. Furthermore, the usefulness of the list for the functions of the compiler is rather limited, because of the indirect nature of the influence of a compiler function on the result of the benchmark.

In case of the HTTP Ping and TCP Ping benchmarks, determining the modifications that influence the parts of Mono used by the benchmarks narrowed down the list of modifications significantly. For a performance regression of almost 40% in the HTTP Ping benchmark between Mono versions from April 4

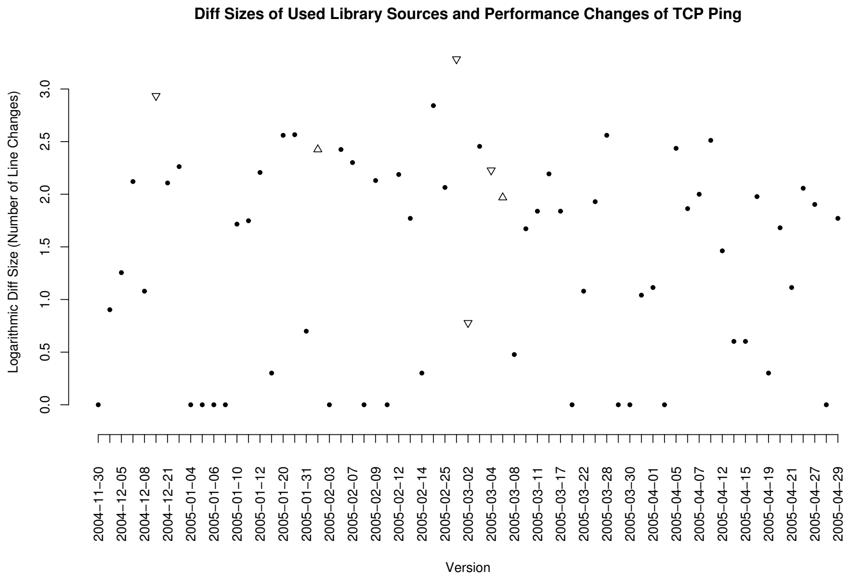


Fig. 5. Diff sizes of sources used by TCP Ping benchmark with detected performance changes

and April 5, the list of modifications contained 39 physical modifications of the application libraries, but only 7 physical modifications belonging to 4 logical modifications influenced the parts of Mono used by the benchmark.

Figure 5 shows how considering only the modifications that can influence the result of a benchmark by means discussed above narrows down the size of modifications between consecutive versions of Mono from figure 4 for the TCP Ping benchmark. Figure 5 also demonstrates that the magnitude of detected performance changes needs not to follow the size of the modifications, even when only modifications that can potentially influence performance are taken into account.

Even when considering only the modifications that can influence the result of a benchmark, it can be difficult to determine which of the logical modifications potentially associated with a performance change is the one that caused the change. When the descriptions of the intent of modifications from the change log do not help, additional benchmark experiments can be used for correlating the performance change with one of the modifications.

3.3 Experimentally Tracking Modifications

Given a list of logical modifications potentially associated with a performance change and the version of software where the performance change first exhibited itself, auxiliary versions of software can be created by reverting the modifications one by one, or by applying the modifications one by one to the immediately preceding version. Regression benchmarking of the auxiliary versions can help determine which of the list of logical modifications is the one that caused the performance change.

Applying this method consistently, regression benchmarking can consider all logical modifications on daily versions of software one by one. While such an approach can be automated, it makes regression benchmarking more time consuming.

Alternatively, the modification most likely to be the one that caused the performance change can be selected manually. A pair of auxiliary versions can then be created, one by reverting the modification from the version of software where the performance change first exhibited itself, another by applying the modification to the immediately preceding version. Regression benchmarking of the auxiliary versions can help confirm or reject the choice of modification.

This method was used on the performance regression of almost 40% in the HTTP Ping benchmark between Mono versions from April 4 and April 5. The list of modifications potentially associated with the regression was first narrowed down to 4 logical modifications. Regression benchmarking has shown that the modification that caused the regression was a rewrite of the function for converting the case of a string.

This method was also used on the performance regression of almost 24% in the FFT benchmark between Mono versions from August 10 and August 11. The FFT benchmark does not use the application libraries and the list of changes potentially associated with the regression therefore concerned only the virtual machine and the compiler. Regression benchmarking has shown that the modification that caused the regression was introduction of a particular loop optimization into the set of default optimizations performed by the just-in-time compiler.

In case of the performance improvement of almost 17% in the FFT benchmark between Mono versions from March 4 and March 7, regression benchmarking has shown that the modification that caused the improvement was a rewrite of the function for passing control between native and managed code and confirmed the improvement.

Finally, by far the biggest performance improvement of almost 99% in the TCP Ping benchmark between Mono versions from December 8 and December 20 was caused by a rewrite of the communication code to pass data in chunks rather than byte by byte. Regression benchmarking confirmed the improvement. Before this particular modification, the HTTP Ping benchmark reported smaller duration of operations than the TCP Ping benchmark, which is intuitively a suspect result. Eventually, similar dependencies between benchmark results could also be described and tested.

It should be noted that while regression benchmarking can detect performance regressions, it cannot detect performance problems that have been part of the software prior to regression benchmarking. The passing of data byte by byte rather than by chunks is an example of such a situation.

We can consider other methods that narrow down the list of modifications potentially associated with a performance change. In case of Mono, these include separate regression benchmarking of modifications that influence the compiler, the modifications that influence the virtual machine, and the modifications that influence the application libraries. These methods would apply to any other platform with a compiler, virtual machine and application libraries, such as the Java platform.

As described, the method of using logical modifications does not depend on the software, but only on the versioning system. Although the implementation of the method of tracing calls to application libraries and then locating sources of the called methods is specific to the Mono platform, it can be implemented for other platforms that use dynamic linking or have debuggers and debug information in files. The implementation is then only specific to the platform the tested software uses.

A challenge for future work on regression benchmarking lies in automating the methods from sections 3.1, 3.2 and 3.3, so that performance changes can be routinely correlated with modifications.

4 Conclusion

We have developed a regression benchmarking environment that automatically and reliably detects performance changes in software. The features of the environment include a fully automated download, building and benchmarking of new versions of software and statistically sound analysis of the benchmark results. The environment is used to monitor daily versions of Mono, the open source implementation of the .Net platform. From August 2004 to April 2005, the environment has detected a number of performance changes, 15 of those exceeding 10%, and 6 of those exceeding 20% of total performance.

The regression benchmarking environment makes the results of the analysis of the benchmark results available on the web [7], giving the developers of Mono the ability to check the impact of modifications on the performance of the daily versions. The developers of Mono have reacted positively especially to the confirmation of modifications done with the intent of improving performance. Our effort focuses on locating the modifications that caused inadvertent performance changes, which currently requires a good knowledge of the software and therefore an attention of the developers to be successful.

In sections 3.1, 3.2 and 3.3, we have proposed three methods that help narrow down the list of modifications potentially associated with a performance change. The three methods were applied on the daily versions of Mono to locate modifications that caused 5 out of 15 performance changes exceeding 10%, 4 out of 6 of those exceeding 20% of total performance. While these methods have been tested on the Mono platform, they can be implemented for other platforms as well. They specifically do not depend on the tested software itself.

Future work on regression benchmarking includes extending the methods from sections 3.1, 3.2 and 3.3, as well as making the methods automated, so that correlating performance changes with modifications is less demanding. This is necessary to help regression benchmarking achieve the same status as regression testing in the software quality assurance process.

Acknowledgement. The authors would like to express their thanks to Jaromir Antoch, Alena Koubkova and Tomas Ostatnicky for their help with mathematical statistics. This work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 201/05/H014.

References

1. Jeffries, R.E., Anderson, A., Hendrickson, C.: Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
2. Bulej, L., Kalibera, T., Tuma, P.: Repeated results analysis for middleware regression benchmarking. Performance Evaluation **60** (2005) 345–358
3. Bulej, L., Kalibera, T., Tuma, P.: Regression benchmarking with simple middleware benchmarks. In Hassanein, H., Olivier, R.L., Richard, G.G., Wilson, L.L., eds.: International Workshop on Middleware Performance, IPCCC 2004. (2004) 771–776
4. Kalibera, T., Bulej, L., Tuma, P.: Generic environment for full automation of benchmarking. In Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., Schweiggert, F., eds.: SOQUA/TECOS. Volume 58 of LNI., GI (2004) 125–132
5. Novell, Inc.: The Mono Project. http://www.mono-project.com (2005)
6. ECMA: ECMA-335: Common Language Infrastructure (CLI). ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland (2002)
7. Distributed Systems Research Group: Mono regression benchmarking. http://-nenya.ms.mff.cuni.cz/projects/mono (2005)
8. DOC Group: TAO performance scoreboard. http://www.dre.vanderbilt.edu/-stats/performance.shtml (2005)
9. Prochazka, M., Madan, A., Vitek, J., Liu, W.: RTJBench: A Real-Time Java Benchmarking Framework. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
10. Dillenseger, B., Cecchet, E.: CLIF is a Load Injection Framework. In: Workshop on Middleware Benchmarking: Approaches, Results, Experiences, OOPSLA 2003. (2003)
11. Memon, A.M., Porter, A.A., Yilmaz, C., Nagarajan, A., Schmidt, D.C., Natarajan, B.: Skoll: Distributed continuous quality assurance. In: ICSE, IEEE Computer Society (2004) 459–468
12. Courson, M., Mink, A., Marçais, G., Traverse, B.: An automated benchmarking toolset. In Bubak, M., Afsarmanesh, H., Williams, R., Hertzberger, L.O., eds.: HPCN Europe. Volume 1823 of Lecture Notes in Computer Science., Springer (2000) 497–506
13. Kalibera, T., Bulej, L., Tuma, P.: Benchmark precision and random initial state. In: accepted for 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005). (2005)
14. Buble, A., Bulej, L., Tuma, P.: CORBA benchmarking: A course with hidden obstacles. In: IPDPS, IEEE Computer Society (2003) 279

15. Hauswirth, M., Sweeney, P., Diwan, A., Hind, M.: The need for a whole-system view of performance. In: Component And Middleware Performance workshop, OOPSLA 2004. (2004)
16. Gu, D., Verbrugge, C., Gagnon, E.: Code layout as a source of noise in JVM performance. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
17. The FreeBSD Documentation Project: FreeBSD Developers' Handbook. http://-www.freebsd.org/doc/en/books/developers-handbook (2005)
18. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley-Interscience, New York, NY, USA (1991)
19. Re, C., Vogels, W.: SciMark – C#. http://rotor.cs.cornell.edu/SciMark/ (2004)
20. Pozo, R., Miller, B.: SciMark 2.0 benchmark. http://math.nist.gov/scimark2/ (2005)

Author Index