

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Raffaella Mirandola Ian Gorton
Christine Hofmeister (Eds.)

Architectures for Adaptive Software Systems

5th International Conference on the Quality
of Software Architectures, QoSA 2009
East Stroudsburg, PA, USA, June 24-26, 2009
Proceedings



Springer

Volume Editors

Raffaella Mirandola
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Golgi 40, 20133 Milano, Italy
E-mail: mirandola@elet.polimi.it

Ian Gorton
Pacific Northwest National Laboratory
Computational and Information Sciences
P.O. Box 999, MS: K7-90, Richland, WA 99352, USA
E-mail: ian.gorton@pnl.gov

Christine Hofmeister
East Stroudsburg University
Computer Science Department
200 Prospect Street, East Stroudsburg, PA 18301-2999, USA
E-mail: chofmeister@po-box.esu.edu

Library of Congress Control Number: 2009930684

CR Subject Classification (1998): D.2.11, D.3, C.4, B.8, D.4.8, D.2.4, F.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-02350-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02350-7 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12696598 06/3180 5 4 3 2 1 0

Preface

Much of a software architect's life is spent designing software systems to meet a set of quality requirements. General software quality attributes include scalability, security, performance or reliability. Quality attribute requirements are part of an application's non-functional requirements, which capture the many facets of how the functional requirements of an application are achieved. Understanding, modeling and continually evaluating quality attributes throughout a project lifecycle are all complex engineering tasks which continue to challenge the software engineering scientific community. While we search for improved approaches, methods, formalisms and tools that are usable in practice and can scale to large systems, the complexity of the applications that the software industry is challenged to build is ever increasing. Thus, as a research community, there is little opportunity for us to rest on our laurels, as our innovations that address new aspects of system complexity must be deployed and validated.

To this end the 5th International Conference on the Quality of Software Architectures (QoSA) 2009 focused on architectures for adaptive software systems. Modern software systems must often reconfigure their structure and behavior to respond to continuous changes in requirements and in their execution environment. In these settings, quality models are helpful at an architectural level to guide systematic model-driven software development strategies by evaluating the impact of competing architectural choices. At run time, quality models can play an important role in enabling calibration and validation of a system model to accurately reflect the properties of the executing system. This leads to the idea that architectural models should continue to exist at run time to facilitate the necessary dynamic changes that can support self-adaptation of the implemented system. In so doing, the conference continued QoSA's tradition of using software architectures to develop and evolve high-quality software systems.

In line with a broad interest, QoSA 2009 received 33 submissions. From these submissions, 13 were accepted as long papers after a careful peer-review process followed by an online Program Committee discussion. This resulted in an acceptance rate of 39%. The selected technical papers are published in this volume. For the third time, QoSA 2009 was held as part of the conference series Federated Events on Component-Based Software Engineering and Software Architecture (COMPARCH). The federated events were QoSA 2009, the 12th International Symposium on Component-Based Software Engineering (CBSE 2009). Together with COMPARCH's Industrial Experience Report Track and the co-located Workshop on Component-Oriented Programming (WCOP 2009), COMPARCH provided a broad spectrum of events related to components and architectures. By integrating QoSA's and CBSE's technical programs in COMPARCH 2009, both conferences elaborated their successful collaboration thus demonstrating the close relationship between software architectures and their constituting software components.

Among the many people who contributed to the success of QoSA 2008, we would like to thank the members of the Program Committees for their valuable work during

the review process, as well as David Garlan and Kevin Sullivan for their COMPARCH keynotes. Additionally, we thank Alfred Hofmann from Springer for his support in reviewing and publishing the proceedings volume.

April 2009

Ian Gorton
Raffaella Mirandola
Christine Hofmeister

Organization

QoSA 2009 (Part of COMPARCH 2009)

General Chair

Christine Hofmeister East Stroudsburg University, Pennsylvania, USA

Program Committee Chairs

Ian Gorton Pacific Northwest National Laboratory, USA

Raffaella Mirandola Politecnico di Milano, Italy

Program Committee

Danilo Ardagna Politecnico di Milano, Italy
Colin Atkinson University of Mannheim, Germany
Muhammad Ali Babar Lero, Ireland
Len Bass Software Engineering Institute, USA
Steffen Becker FZI, Germany
Jan Bosch Intuit, USA
Ivica Crnkovic Mälardalen University, Sweden
Rogerio De Lemos University of Kent, UK
Antinisca Di Marco Università dell'Aquila, Italy
Carlo Ghezzi Politecnico di Milano, Italy
Aniruddha Gokhale Vanderbilt University, USA
Vincenzo Grassi Università Roma "Tor Vergata", Italy
Jens Happe SAP/University of Karlsruhe, Germany
Darko Huljenic ERICSSON, Croatia
Samuel Kounev University of Karlsruhe, Germany
Heiko Koziulek ABB, Germany
Philippe Kruchten University of British Columbia, Canada
Nenad Medvidovic University of Southern California, USA
Jose' Merseguer University of Saragoza, Spain
Robert Nord Software Engineering Institute, USA
Boyana Norris Argonne National Laboratory, USA
Sven Overhage University of Augsburg, Germany
Dorina Petriu Carleton University, Canada
Frantisek Plasil Charles University, Czech Republic
Sasikumar Punnekkat Mälardalen University, Sweden
Ralf Reussner University of Karlsruhe, Germany
Roshanak Roshandel Seattle University, USA
Bernhard Rumpe University of Technology Braunschweig,
Germany

VIII Organization

Antonino Sabetta	ISTI-CNR Pisa, Italy
Raghu Sangwan	Penn State, USA
Anne-Marie Sassen	EU Commission
Jean-Guy Schneider	Swinburne University, Australia
Judith Stafford	Tufts University, USA
Clemens Szyperski	Microsoft, USA
Petr Tuma	Charles University, Czech Republic
Hans van Vliet	Vrije Universiteit, The Netherlands
Wolfgang Weck	Independent Software Architect, Switzerland
Michel Wermelinger	Open University, UK
Murray Woodside	Carleton University, Canada
Steffen Zschaler	Lancaster University, UK

Co-reviewers

Vlastimil Babka	Charles University, Czech Republic
Simona Bernardi	Universita' degli Studi di Torino, Italy
Hongyu Pei-Breivold	Mälardalen University, Sweden
Franz Brosch	University of Karlsruhe, Germany
Lubos Bulej	Charles University, Czech Republic
Radu Dobrin	Mälardalen University, Sweden
Thomas Goldschmidt	University of Karlsruhe, Germany
Michael Kuperberg	University of Karlsruhe, Germany
Diego Perez	University of Saragoza, Spain
Matteo Rossi	Politecnico di Milano, Italy
Paola Spoletini	Politecnico di Milano, Italy
Giordano Tamburrelli	Politecnico di Milano, Italy

Table of Contents

Model-Driven Quality Analysis

A Model-Based Framework to Design and Debug Safe Component-Based Autonomic Systems	1
<i>Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien</i>	
Applying Model Transformations to Optimizing Real-Time QoS Configurations in DRE Systems	18
<i>Amogh Kavimandan and Aniruddha Gokhale</i>	
Automated Architecture Consistency Checking for Model Driven Software Development	36
<i>Matthias Biehl and Welf Löwe</i>	

Architectural Performance Prediction

Improved Feedback for Architectural Performance Prediction Using Software Cartography Visualizations	52
<i>Klaus Krogmann, Christian M. Schweda, Sabine Buckl, Michael Kuperberg, Anne Martens, and Florian Matthes</i>	
Predicting Performance Properties for Open Systems with KAMI	70
<i>Carlo Ghezzi and Giordano Tamburrelli</i>	
Compositional Prediction of Timed Behaviour for Process Control Architecture	86
<i>Kenneth Chan and Iman Poernomo</i>	
Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines	101
<i>Stefan Björnander, Lars Grunske, and Kristina Lundqvist</i>	

Architectural Knowledge

Achieving Agility through Architecture Visibility	116
<i>Carl Hinsman, Neeraj Sangal, and Judith Stafford</i>	
Successful Architectural Knowledge Sharing: Beware of Emotions	130
<i>Eltjo R. Poort, Agung Pramono, Michiel Perdeck, Viktor Clerc, and Hans van Vliet</i>	

Toward a Catalogue of Architectural Bad Smells 146
*Joshua Garcia, Daniel Popescu, George Edwards, and
Nenad Medvidovic*

Case Studies and Experience Reports

On the Consolidation of Data-Centers with Performance Constraints . . . 163
Jonatha Anselmi, Paolo Cremonesi, and Edoardo Amaldi

Evolving Industrial Software Architectures into a Software Product
Line: A Case Study 177
Heiko Koziol, Roland Weiss, and Jens Doppelhammer

Adaptive Application Composition in Quantum Chemistry 194
*Li Li, Joseph P. Kenny, Meng-Shiou Wu, Kevin Huck,
Alexander Gaenko, Mark S. Gordon, Curtis L. Janssen,
Lois Curfman McInnes, Hirotohi Mori, Heather M. Netzloff,
Boyana Norris, and Theresa L. Windus*

Author Index 213

A Model-Based Framework to Design and Debug Safe Component-Based Autonomic Systems*

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

Université Lille 1 - LIFL CNRS UMR 8022 - INRIA
59650 Villeneuve d'Ascq, France

{Guillaume.Waignier, Anne-Francoise.LeMeur,
Laurence.Duchien}@inria.fr

Abstract. Building autonomic applications, which are systems that must adapt to their execution context, requires architects to calibrate and validate the adaptation rules by executing their applications in a realistic execution context. Unfortunately, existing works do not allow architects to monitor and visualize the impact of their rules, nor that they let them adjust these rules easily.

This paper presents a model-based framework that enables architects to design and debug autonomic systems in an iterative and uniformed process. At design-time, architects can specify, using models, the application's structure and properties, as well as the desired adaptation rules. At debugging-time, the running application and the models coexist such that the models control the application dynamic adaptation, thanks to a control loop that reified runtime events. Each triggered adaptation is first tested at the model level to check that no application property is broken. Furthermore, architects can at any time modify the models in order to adjust the adaptation rules or even parts of the application. All changes at the model level, if checked correct, are directly propagated to the running application. Our solution is generic regarding the underlying platforms and we provide a performance evaluation of our framework implementation.

1 Introduction

Being able to build autonomic systems [1] has become important in numerous application domains such as ubiquitous applications. These systems need to adapt to resource availability in their often changing executing environments. Furthermore they must be robust and provide a satisfactory level of quality of service.

One approach that has been recognized to facilitate the development of complex systems is proposed by the component-based paradigm [2], which promotes building an application by assembling software components. Component assemblies are specified at design-time but many component-based platforms offer also the capability of modifying assemblies at runtime, making these platforms interesting candidates to run adaptive applications. It is thus not surprising that some component-based approaches have proposed extensions to support the development of autonomic systems [3,4]. In these works, the Architecture Description Language (ADL), which commonly enables architects to describe the structure of their application at design-time, has been extended

* This work was partially funded by the French ANR TL FAROS project.

to also support the description of high-level adaptation rules that specify how the application should evolve with respect to various execution context changes.

If some solutions to build autonomic applications have been proposed, several challenges remain to be addressed [5]. One of them is related to the design of autonomic systems and more precisely to the design of the adaptation rules of an autonomic system. Indeed, in [5] the author argues that the design of autonomic systems should support an iterative development process in order to allow adaptation rules to be adjusted accordingly to the realist execution context of the system. This challenge calls thus for means to facilitate the debugging and tuning of autonomic systems.

This paper presents an approach to design and debug autonomic component-based systems. Our solution is a framework that relies on a strong coupling between the specifications of an application, expressed with models, and the application running on a platform. This coupling is concretized by a control loop that enables events at the platform level to be reified at the model level as inputs of adaptation rules. The rules are evaluated and applied on the models in order to first check if the modifications do not break the overall application properties. If no problem is detected, the modifications are automatically propagated to the running application. Architects can thus visualize the evolution of their applications at the model level and easily adjust the adaptation rules at any time. By iteratively performing this debugging process, architects finely tune their autonomic systems to better react to their realist execution context.

We have built our approach by extending and generalizing CALICO, a Component Assembly Interaction Control framework that we have previously developed [6]. One particularity of our framework is that it allows the specification of an application independently of any underlying runtime platform and maps this specification towards a given platform implementation. We have carefully designed CALICO so that generic concepts are factorized and platform-specific treatments are clearly separated. Consequently, CALICO's target platforms, which include Fractal [7], OpenCCM [8] and OpenCOM [9], all benefit from our extensions for autonomic systems. Furthermore the support of new runtime platforms and new context change events is largely facilitated.

We describe in this paper all CALICO's extensions related to autonomic systems and evaluate the performances of our framework. The rest of the paper is organized as follows. In Section 2 compares the debugging process in approaches such as [3,4] with the one of CALICO. Section 3 gives an overview of our control loop and iterative debugging process. Section 4 presents the specification models related to automatic systems and Section 5 describes our framework runtime support. Section 6 gives implementation details and evaluates the performances of our approach. Finally, Section 7 provides some related work and Section 8 concludes.

2 Comparison of Debugging Processes

In this section we compare Plastik [3] and DynamicAcme [4] with CALICO when designing and debugging autonomic systems. We illustrate the various debugging processes on an adaptation scenario of a subset of the French Personal Health Record system (PHR) [10], shown in Figure 1. The goal of this system is to provide health-care professionals with all patient medical information required to perform a diagnosis. This

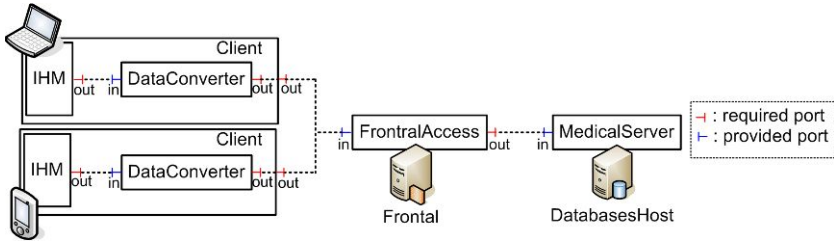


Fig. 1. Subset of the PHR architecture

system is composed of eight components and four hosts. Medical information, such as radiographies or echographies, is stored in the component `MedicalServer` running on the host `DatabasesHost`. The component `FrontalAccess` on the host `Frontal` authenticates the health-care professional and controls the access to confidential information. A health-care professional can consult medical data using a component `Client`, which may be running on various devices, such as a laptop or a PDA. The component `DataConverter` contained in the component `Client` is an image preprocessor that adapts medical images to the device capabilities. In this example, we consider one adaptation rule, which specifies that if a device is using more than 80% of its CPU resources, the component `DataConverter` must be moved on the host `Frontal`.

With Plastik [3] (and similarly with DynamicAcme [4]), the architect defines the structure of the application and the adaptation rules by writing an Acme ADL description file. Once the system is designed and the structural specifications have been checked to be correct, the architect must deploy the system on the OpenCOM component platform to debug it. If the system starts to behave inappropriately, the architect has to detect the problem, modify the Acme description file accordingly and re-deploy the whole system. The architect needs to replay all the execution context changes that have led to the previous malfunction, and iters this whole debugging process until the design resulting system is satisfactory.

With CALICO, the architect can use a graphical interface to specify the structure of the PHR system, the application properties, *e.g.*, QoS constraints, as well as the adaptation rules. All of these specifications are independent of any underlying platform. Then the architect can choose the target runtime platform, such as Fractal [7], OpenCCM [8] or OpenCOM [9]. CALICO statically analyzes all the system specifications to check if they are coherent, and deploys automatically the system in its real execution environment. During the debugging process, a change in the execution context is reified at the design view level and its associated adaptation rule is executed. Modifications are first applied on the specifications, in order to notify the architect of any potential error. If no problem is detected, the modifications are performed on the still running application. Furthermore, if the architect decides to adjust the PHR adaptation rule so that the component is moved when the CPU usage reaches 65%, he/she can simply update the specification in the design view, no stop of the system is required. The architect can design and visually debug the PHR system from the design view in an iterative and uniformed process until the resulting system is satisfactory.

3 Our Proposition to Design and Debug Autonomic Systems

This section presents the architecture of CALICO and focuses on the newly added features to support the debugging of autonomic systems, which includes a QoS and Adaptation metamodels and the design of an autonomic control loop.

3.1 CALICO Architecture

CALICO is composed of two levels: a model level and a platform level (cf. Figure 2). The model level is independent of any component-based or service-oriented platform. It contains the CALICO Architecture Description (AD) metamodels that enable an architect to describe the structure and the properties, *i.e.*, structural, behavioral, dataflow and QoS properties, of an application, as well as contextual adaptation rules, independently of any platform. This level controls also the autonomic adaptation by analyzing each reconfiguration and allows an architect to debug autonomic systems. The platform level holds the running system on a target platform, such as Fractal [7], OpenCCM [8] or OpenCOM [9].

3.2 CALICO Autonomic Control Loop

The support for the debug of autonomic system relies on an enhanced autonomic computing control loop. The notion of control loop was initially composed of four steps [1]: The *monitoring* of the execution context, the *analysis* of the situation, the *planning* of the reconfiguration and the *execution* of the reconfiguration. In CALICO, we have more finely decomposed this loop into six steps as shown in Figure 2:

(1) The running system reifies runtime context changes, as events, to the model level which controls the autonomic adaptation. This step corresponds to the monitoring step.

(2) At the model level, the *Adaptation* tool analyzes the runtime events and if needed, triggers the execution of the corresponding adaptation rules. This step corresponds to the analysis of the context.

(3) The adaptation, expressed in the triggered adaptation rules, is performed on the CALICO AD model. This step corresponds to the planning of the adaptation.

(4) The *Interaction Analysis* tool statically analyzes the AD models to check if the specified application properties are respected. The platform specific constraints are also included in the analysis to guarantee that the architecture respects the platform specifications and can be thus deployed on this platform. Each detected inconsistency is notified to the architect and the adaptation is not propagated to the running system. This allows the architect to correct the adaptation rules and guarantee that an adaptation can never break the system. This step corresponds to the debugging of the adaptation.

(5) This step aims to fill the gap between the evaluation of the specification and the monitoring of the context. The *Code Instrumentation* tool analyzes the updated AD models to identify if and where sensors must be added or removed from the underlying platform. This step relies on both code generation and code instrumentation using an aspect weaving approach [11] to prepare the code that needs to be deployed on the system.

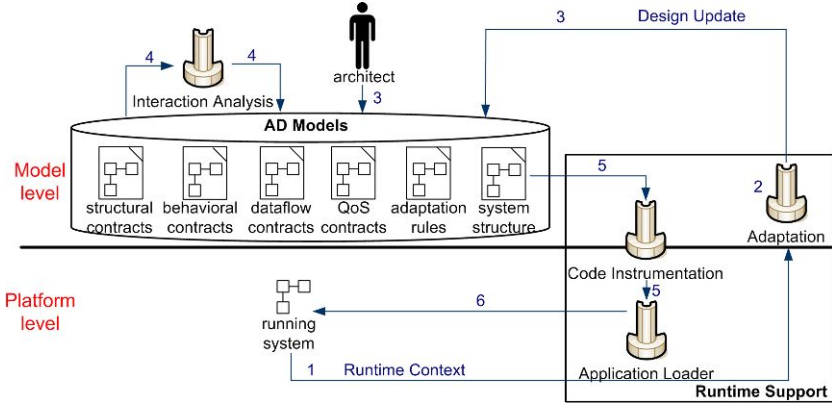


Fig. 2. CALICO autonomic control loop

(6) The *Application Loader* applies automatically the changes on the running system. To do so, it compares the difference between the new AD models and the models of the running system to identify the elementary actions to perform (e.g., adding/removing components or bindings) and calls the platform specific API to modify accordingly the running system. This step corresponds to the execution of the reconfiguration.

Overall, CALICO allows architects to visualize the evolution of this application at the model level, which is kept synchronized with the running system. Our autonomic control loop is not closed since it enables architects to dynamically debug their autonomic systems and adjust the adaptation rules if needed. Furthermore, they can directly modify the AD models to update their design to cope with an unexpected evolution of the runtime context.

4 CALICO Metamodels to Support Autonomic Systems

The model level contains the AD metamodels to describe the application structure and the structural [12], behavioral [13], dataflow and QoS properties of an application. The system structure metamodel (cf. Figure 3) offers metaclasses to represent the structure of a component-based system or a service-oriented architecture. It is composed of common concepts: the *Entity*, which can be a component or a service, the *CommunicationPoint*, i.e., generally called a port, and the *Connector*. These concepts result from a domain analysis that we have performed on several component and service platforms [14]. The behavioral metamodel describes the control flow paths of the system, where the control flow operators are the sequence, the fork and the merge of the control flow. The dataflow metamodel enables the architect to specify constraints on the values of the data exchanged by the application. The architect can also describe, using the *Adaptation* metamodel, the modification rules to perform on the application when a given context change occurs. All the metamodels are independent of any runtime platform. Details of the structural, behavioral and dataflow metamodels, as well as of the static interaction analysis can be found in our previous work [6,14]. However

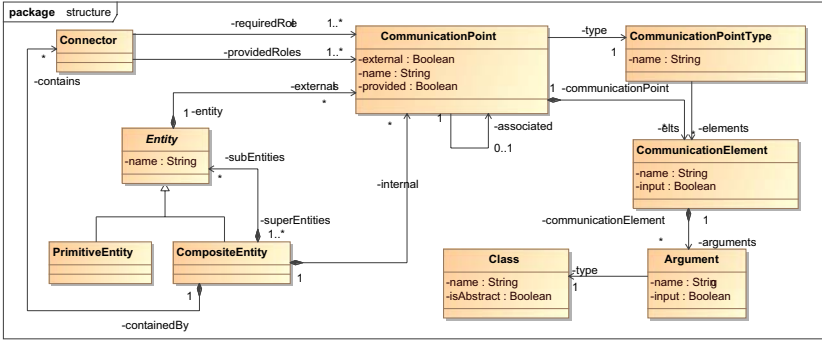


Fig. 3. System structure metamodel

these details are not necessary to understand the rest of the paper. We now focus on the autonomic-related metamodels, which include the QoS and Adaptation metamodels.

4.1 QoS Metamodel

QoS specifications are used by the architect to constraint the non-functional properties of a system, such as reliability, performance or timing. To express the QoS specifications in a generic way, we have extended our previous QoS metamodel [14] and identified the common concepts existing in several QoS specification languages such as QML [15], WSLA [16] and WS-agreement [17]. The result of this analysis has led to the definition of the QoS metamodel shown in Figure 4. In this metamodel, a QoS specification is represented by the `QoSSpecification` metaclass. It is composed of `Assumption` and `Guarantee` terms. The assumption term defines QoS properties that a client entity requires and the guarantee term defines QoS properties that a service entity provides. For example, the architect can specify that the component `Client` of the PHR system must receive medical information in less than 5 seconds and that the component `MedicalServer` guarantees to always send medical data in less than 1 second. The syntax of a QoS expression is similar to a QML expression, *i.e.*, it is a boolean expression manipulating real, integer and string values and the comparison operators are `<`, `>`, `≤`, `≥`, `=`, `≠`.

In order to be extensible and allow the support of new QoS properties, we introduce the concept of `QoSType`. Each QoS type corresponds to a kind of QoS property, such as the CPU load in percentage or the available memory in Mb, and each QoS specification is associated with a QoS type. In CALICO, we have defined a set of QoS types and a QoS expert can add the support of new kinds of QoS property by defining new QoS types. A QoS type is similar to the concept of *contract type* in QML.

The use of the `Assumption` and `Guarantee` terms allows CALICO static analysis tool to compute the QoS contract over the control flow, to check if all QoS assumption terms are respected based on the guarantee terms [14]. Each QoS type is defined with its set of composition operators describing how QoS specifications must be composed accordingly to the control flow operators, *i.e.*, the sequence, the fork and the merge of the control flow (cf. Figure 4). For example, let `T` be a QoS type corresponding to

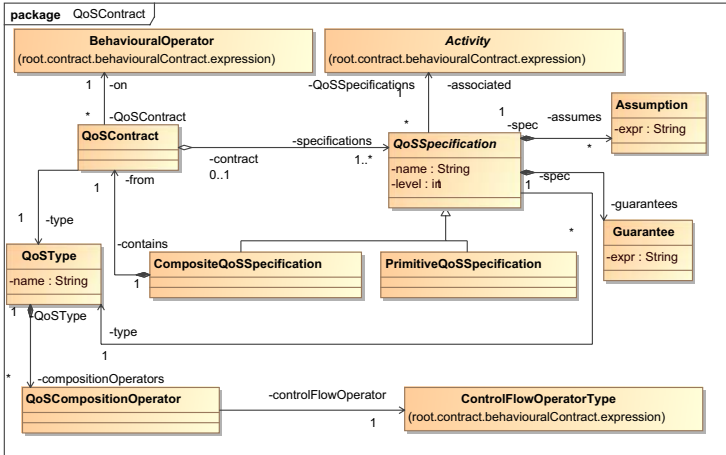


Fig. 4. QoS specification metamodel

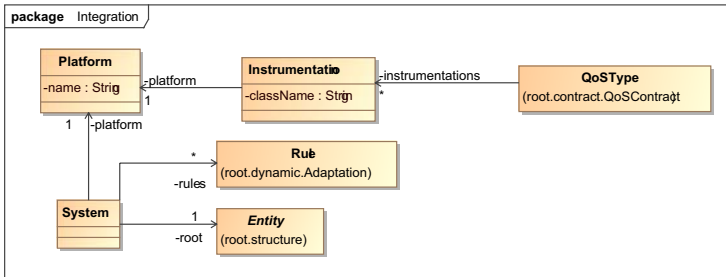


Fig. 5. Integration metamodel

the maximal response time of an entity and T_1 (*resp.* T_2) the maximal response time of an entity E_1 (*resp.* E_2). Thus the QoS composition operator of T for the parallel control flow operator is the maximum because the maximal response time of the parallel execution of E_1 and E_2 is $\max(T_1, T_2)$.

4.2 Link between QoS Specification and QoS Sensors

In CALICO, the design phase and the debugging phase are strongly coupled. This coupling relies on the association between each QoS specification expressed at the model level and the corresponding QoS sensors that is needed to monitor the evolution of the quality properties in the application context. In our approach, this association is described with the Integration metamodel (cf. Figure 5). Each `QoSType` is linked with an `Instrumentation` metaclass that represents the CALICO plugin that is in charge of adding the needed sensor in the platform. Furthermore, to be independent of any platform, each `Instrumentation` metaclass is also linked with the `Platform` metaclass, which represents a given underlying platform. This allows CALICO to load the

right sensor depending on the QoS types used in the specifications and the underlying platform.

Furthermore, the Integration metamodel (cf. Figure 5) is also used to associate the adaptive System designed by the architect with the target platform represented by the metaclass Platform. A System is composed of the description the architecture with its properties (cf. the root Entity) and the set of adaptation rules (cf. the metaclass Rule).

Moreover, the Integration model is extensible and allows a QoS expert to define new sensors by describing how the system must be instrumented, accordingly to the QoS specifications expressed by the architect and the underlying platform. He/she describes which type of event the sensor reifies and on which platform it can be applied, by instantiating the Integration metamodel. OCL constrains are defined to guarantee the consistency of the Integration model. For example the following OCL constraint checks that there always exists at least one instrumentation sensor for each QoS type expressed in the Adaptation model and for the underlying platform chosen by the architect:

```
context System
  inv : self.rules->forall(r:Rule | r.event.ocIsTypeOf(QoSEvent) implies
    r.event.ocIsType(QoSEvent).type.instrumentations->exists(
      i:Instrumentation|i.platform=self.platform))
```

These OCL constraints are included in the set of constraints verified by the Static Analysis tool. Consequently, if no sensor is provided for the underlying platform, CALICO notifies the architect during the static analysis, before the system deployment, that this adaptation rule will not be taken into account during the debugging phase.

Overall the Integration metamodel is the key element of CALICO to enable architects to build autonomic systems in an iterative design process. It always keeps the link between the design specifications and the needed sensors in the platform. Moreover this metamodel is flexible to support new QoS types and their QoS sensors for the different platforms.

4.3 Adaptation Metamodel

We have defined an Adaptation metamodel to enable an architect to design the adaptation rules of a system independently of any reconfigurable underlying platform (cf. Figure 6). The expression of a Rule is based on the Event Condition Action (ECA) paradigm [18]. A Rule is composed of three parts: an Event, a Condition and an Action. The Event represents a change in the runtime context such as the modification of the CPU load. The Condition is a boolean expression on the value of the event, such as checking if the CPU load is greater than 80%. The Action describes the action to perform when the condition holds, such as moving CPU resource consuming components on another host (cf. Section 2).

Events. To be generic, we have performed a domain analysis to identify events that are common to several component-based platforms and service-oriented platforms. We have found three categories of events : dataflow events, structural events and QoS events, as shown in Figure 6. Dataflow events correspond to the reification of the values data exchanged in the application. There are two dataflow events: MessageEnter and MessageExit, which correspond respectively to the reception and sending of a new

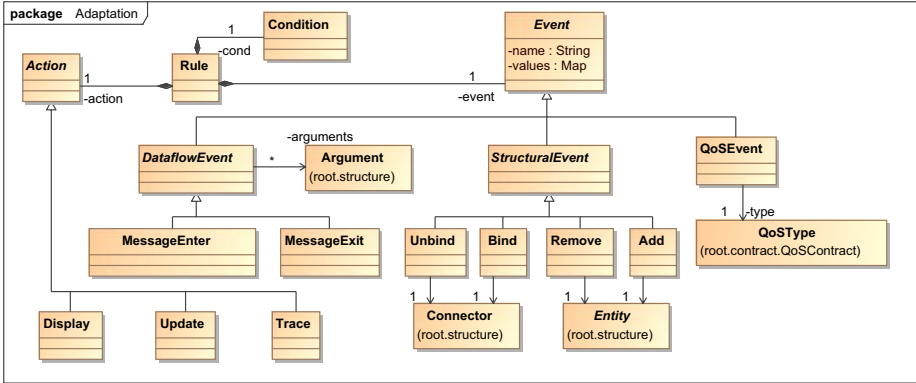


Fig. 6. Adaptation metamodel

message at a component or service port. Structural events express a structural modification of the running system. There are four structural events. *Add* (*resp.* *Remove*) corresponds to the addition (*resp.* removing) of a component in the system or to the publication (*resp.* unpublication) of a service in a registry. *Bind* (*resp.* *Unbind*) corresponds to the addition (*resp.* removing) of a connector between two components or to the addition (*resp.* removing) of a partnerLink in a service-oriented architecture. QoS events reflect a modification in the application context. Each QoS event is associated with one QoS type.

Condition. The syntax of a condition is a boolean expression as in QML. The variables can be integers, reals or strings and the comparison operators are the same than the ones used in the QoS expressions, *i.e.*, $<$, $>$, \leq , \geq , $=$, \neq .

Actions. The action part of the rule defines the kind of action to perform, independently of any platform. The scope of the action is limited to the model level in order to avoid the direct modification of the system on the platform level. This approach guarantees that no action can break the running system because the modification is propagated on the platform only if no application property is violated at the model level.

In our approach, contrary to several existing works [4,3], the actions are not limited to structural modifications of the running application. Indeed in CALICO, the action part is extensible and allows the architect to define new action types when needed. In the current implementation, we have three predefined types of action. To debug the application, the architect can use the action type `Display` to print the triggered rules at the model level. This is useful, if an error occurs, to identify the model element that could lead to a runtime problem and thus to fix the design accordingly. The action type `Trace` enables the architect to visualize at the model level all messages exchanged during the execution of the system in order to facilitate the understanding of the application behavior. To specify an autonomic adaptation of the running system, the architect can use the action type `Update` to specify the changes that must be performed on the AD models. This action type is a script, expressed in a script language, such as EMFScript [19], that modifies the AD models by adding/removing components and connectors.

Overall, the genericity of the adaptation metamodel allows an architect to express adaptation rules independently of any platform. Furthermore the adaptation rules can be reused on different platforms, since in CALICO the adaptation concerns are clearly separated from the business concerns.

5 CALICO Runtime Support for Autonomic Systems

The execution of an autonomic system requires tedious preparation: runtime events must be reified, then analyzed by an autonomic computing control loop and the adaptation rules must be launched accordingly. Usually, the developer must provide the support for all of these tasks. To reify runtime events, he/she must integrate the appropriate sensors since few component or service platforms provide natively QoS sensors. He/she must also implement the autonomic control loop. In CALICO, all these tasks are automatically handled by the CALICO runtime support. This support relies on three tools: the Code Instrumentation tool, the Application Loader tool and the Adaptation tool.

5.1 Code Instrumentation Tool

The *Code Instrumentation* tool instruments the application code to enable the running system to reify runtime events. We focus on the support related to QoS events. In CALICO, the use of QoS sensors is transparent to the architect. CALICO is able to automatically identify the kind of sensors required to reify the type of events used in the Adaptation model. The Code Instrumentation tool analyzes each rule expressed in the Adaptation model (cf. Figure 6) and identifies the types of event, associated with `QoSType`, that must be captured. By using the Integration model (cf. Figure 5), the tool finds the `Instrumentation` plugin that must be applied to insert the appropriate sensor in the platform, accordingly to the type of the event and the underlying platform. The instrumentation plugin takes as input the `Event` that must be reified. It has access to all CALICO AD models, which allows it to compute the location where the sensor must be inserted, accordingly to this event. This enables CALICO to insert sensors so that only the required events are reified, avoiding the overcost of sending useless events.

5.2 Application Loader Tool

The goal of the Application Loader tool is to deploy on a target platform the application corresponding to the architecture design at the model level. This tool keeps the models of the running system in order to perform incremental deployment. Indeed, when an architect modifies his/her design to fix detected runtime errors, the Application Loader tool computes the difference between the model of the running system and the new model designed by the architect. Thanks to this analysis, it identifies the changes that must be performed on the running system, *i.e.*, adding/removing components and connectors independently of any underlying platform. Furthermore, this tools always keeps a link between each model element designed by the architect and the corresponding running element in the platform.

5.3 Adaptation Tool

The Adaptation tool controls the autonomic adaptations. The instrumented running system submits the runtime event to the Adaptation tool, so that this tool can perform the adaptation independently of any platform. It implements a rule-based engine that analyzes all the rules expressed in the Adaptation model and triggers the adaptations accordingly to the events. For each triggered action type, the Adaptation tool loads at the model-level the necessary support to execute the action. For example, for the action type `Update`, it launches the EMFScript interpreter. Thanks to this tool, the architect can modify the rules at any time and the changes will be transparently taken into account by the whole autonomic system.

6 CALICO Prototype

This section gives some details on the implementation of CALICO¹. Furthermore, we evaluate to reactivity of CALICO to debug autonomic systems.

6.1 Implementation Status

The current implementation of CALICO is developed in Java. All CALICO metamodels are implemented with the Eclipse Modeling Framework (EMF). A graphical editor, implemented with the Graphical Modeling Framework (GMF), enables the architect to edit the model during the debugging of the running autonomic system.

CALICO has been carefully designed to allow new extensions in terms of support for new platforms, new QoS sensors and new kinds of adaptation action. For example, to add the support for a new platform, the platform expert has to write two Java classes: one class to generate the code skeleton of the architecture and one class to deploy the architecture. The second class is a wrapper for the platform-specific API used to add/remove components and connectors. The complexity and sizeof of this wrapper depends on the platform API to add/remove new components in the platform. For example, the OpenCOM wrapper contains 300 lines of code and the OpenCCM driver contains 440 lines of code. The semantics of these two wrappers are the same, only the platform API used changes. The current implementation of CALICO fully supports three component-based platform: Fractal, OpenCCM and OpenCOM and we are currently working on the Service Component Architecture (SCA) [20].

To reify structural events, we plan to define aspects to instrument the platform code so that the platform submits a structural event each time the API for adding/removing components/connectors is called. To reify the QoS events, an existing extensible sensor framework, called WildCAT [21], has been integrated. But other frameworks can be integrated thanks to the Integration metamodel. We plan to integrate sensor frameworks such as Lewys [22] and COSMOS [23]. The reification of the dataflow events is also implemented [6].

CALICO is ready to support distributed component platforms since communications between the platform and the model use the Remote Method Invocation (RMI). Nevertheless, further supports will be needed, for example to propose solutions to handle network failures or disconnections.

¹ CALICO is freely available at <http://calico.gforge.inria.fr>

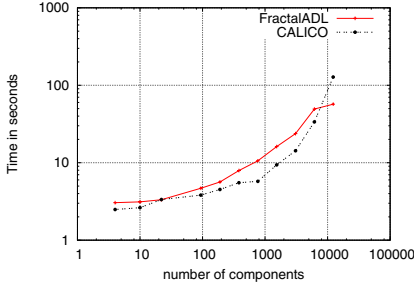


Fig. 7. Time to deploy a system

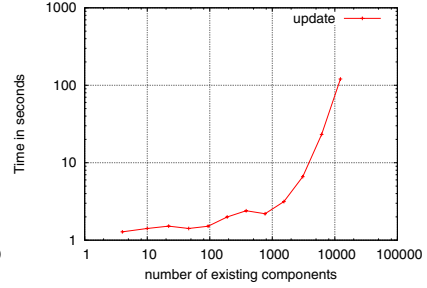


Fig. 8. Cost to add a component

6.2 Evaluation

We have evaluated the performances of CALICO to deploy a whole system, add new components, analyze the structural and behavioral application properties, as well as to reify runtime events. The benchmarks have been executed on a laptop with an Intel®-Core™2 Duo CPU processor at 1.33GHz and Java version 1.6.0_11.

The experimentation has been performed on an application deployed on the Fractal component platform. The structure of this application is an hierarchical tree architecture, where the top level is a composite component A containing a primitive component connected with two composite components. Recursively, the structure of each composite component is similar to the structure of A.

Deployment. First, we have compared the time taken by the native Fractal ADL tool and by CALICO to deploy the application on the Fractal platform (cf. Figure 7). For Fractal ADL, the deployment includes the parsing of the Fractal ADL files and the instantiation of the Fractal components in the platform. For CALICO, it includes the parsing of the EMF description file and the instantiation of the Fractal components in the platform. The benchmark shows that CALICO is as efficient as the native Fractal ADL deployment tool. For an application with 3000 components, CALICO takes 14s whereas Fractal ADL takes 23s.

Figure 8 represents the time it takes to add one component in the root composite component based on the number of existing components in the running system. Up to 3000 components, the complexity is linear and the cost is reasonable for a debug mode: CALICO takes 6s to compare the two models, to deploy and bind the new component.

Memory. Figure 9 represents the memory consumption of both CALICO and the running system based on the number of components. The reduction of the memory used is caused by the launch of the Java garbage collector. The increase of the memory consumption is linear. For 3000 components, the memory consumption is of 47 Mb, which includes CALICO and the running system.

Static analysis. We have also measured the cost of the static analysis that checks if the application properties expressed by the architect are respected. Figure 10 represents the time taken to execute the structural and behavioral analyses based on the number of components. The cost of the structural analysis is negligible, it takes 2.5s to check

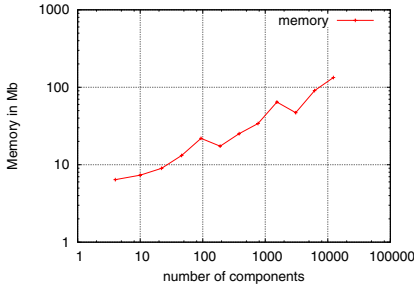


Fig. 9. Memory consumption

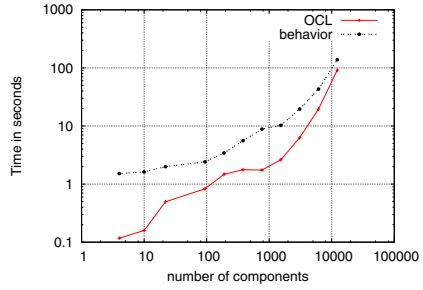


Fig. 10. Cost of the static analysis

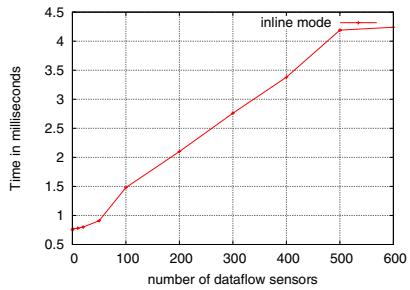
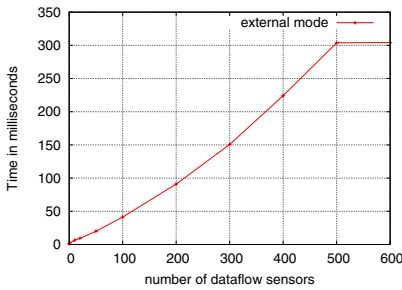


Fig. 11. Time to call a method

1500 components. However, the behavioral analysis takes 10s for 1500 components up to several minutes for 10000 components. Nevertheless, this is not surprising since it is well-known that behavioral analyses may be expensive, intrinsically to the complexity of the algorithms. Moreover, the CALICO analysis tool is flexible and allows the architect to select the kinds of static analysis to perform.

Event reification. Finally, to evaluate the cost of the reification of events from the platform-level to the model-level, we have built a pipe and filter system composed of one client component, one server component that performs an addition of two integers and 500 filter components between the client and the server. We have measured the impact of the reification of the messages that enter and go out from each filter component. CALICO provides the architect with two modes to generate sensors: an external mode and an inline mode. In the external mode, the sensors submit an event to the model level and the condition is evaluated at the model level. In the inline mode, the condition is generated in the sensor code so that the sensor submits an event only if an adaptation is needed. In this case we have implemented a sensor as a component. Consequently, reifying a message entering or leaving a filter component has required sensor components to be added in front and behind the filter component. Figure 11 shows the time taken by the method call based on the number of the sensors added in the system, for both the external and inline modes. In this scenario, the worst case is reached when 600 sensors have been added in the system, the Java stack being unable to handle call propagation through more than 500 filters and 600 sensor components.

The impact of the inline mode is minor: the time taken by the method call is of 4ms in the worst case with 600 sensors, and of 0.75ms without any sensor. Nevertheless, in this mode, each time the architect adjusts the condition expression, CALICO must regenerate the sensor code, recompile it and redeployed it on the platform.

With the external mode, the running system is slower due to the RMI communication between the platform level and the model level. For example in the worst case, it takes 304ms (cf. Figure 11). Nevertheless, the architecture is greatly and quickly reconfigurable since a change of the condition expression at the model level is directly effective at the platform level. Moreover, in practical experience, the maximal number of sensors in a dataflow path is more around 10 than 600, which takes 6ms with the external mode and makes thus the approach still usable.

The cost of the reification of the other QoS properties is directly dependent on the efficiency of the reused existing QoS sensor frameworks [22]. CALICO introduces an overcost due to the RMI communication between the sensors and the model-level, which is nevertheless necessary in order to handle distributed systems.

Discussion. We have chosen to perform our benchmarks on test scenarios that push our implementation of CALICO to its limits. The measures show that the performances of CALICO make it usable for designing and debugging of autonomic systems up to 10000 components. Note that however that not all component platforms are able to execute very large systems. For example, the Apache Tuscany Java version 1.3.2, the reference implementation of SCA, can handle up to 6000 components. Thus it is more likely that the number of components of an application will often be a lot less than 10000 components.

For very large systems, *i.e.*, after 15000 components, the time taken by CALICO to handle autonomic systems becomes exponential. To identify the source of the problem, we have performed some experimentations in which we vary the amount of memory of the Java Virtual Machine (JVM). We have noticed that the time to deploy a system decreases when the amount of the JVM memory increases. For example, by increasing the JVM memory, we gain 30s to deploy 12000 components. We believe that when the available memory becomes very low, EMF puts in cache the model elements and thus the time to process the elements increases greatly.

Overall, CALICO lets the architect choose between a rapid running system that is slowly reconfigurable and a slower running system which is quickly reconfigurable. This choice is however not so strict, the architect can choose more precisely which parts of the architecture are highly reconfigurable and which parts are more static, depending on the adaptation rules that need to be adjusted. At the end of debugging and development activity, the framework CALICO is entirely removed and the adaptive application is executed with a typical autonomic middleware. We plan to implement a migration tool to generate automatically the various configuration files required by the target autonomic middleware.

7 Related Work

To the best of our knowledge, few works address the challenge related to the design and debug of autonomic systems. The work by King et al. [24] is the approach the

closest to our work. They have enhanced the autonomic control loop to include a testing step. During this step, their framework triggers runtime tests, such as Junit tests, and the adaptation is propagated to the running system if the tests succeed. Nevertheless, their framework can only be used on design object-oriented systems. Moreover, their framework does not enable the architect to fix the problem when tests failed. Thus, there is still a gap that remains between the design and the runtime debugging phases.

If few works have addressed the design and debug of autonomic systems, there are however many works that have focused on the monitoring and planning steps of the autonomic control loop.

With respect to monitoring, some works have proposed solutions to capture multidimensional QoS properties, such as COSMOS [23], and thus go further than the current approach offered by CALICO. The part of CALICO that captures and reifies runtime information is however designed so that new sensor frameworks can be easily integrated. To do so, the multidimensional QoS adaptation metamodel of the existing sensor framework must be included in the CALICO AD metamodels and the appropriate plugins of the Code Instrumentation tool must be implemented to deploy the sensors accordingly to the model.

With respect to planning, several works have defined complex adaptation paradigms, such as the mathematical control theory paradigm used in [25,26]. For example, the work of Li et al. [25] proposes a dynamically reconfigurable middleware to adapt multimedia systems accordingly to QoS fluctuation in the surrounding environment. The planning step is based on mathematical models: a linear model and a fuzzy model. Nevertheless in this approach, the architect has to manually express the adaptation actions to perform with respect to QoS variations, which is difficult and error prone. Moreover an error in the adaptation actions could lead to a system failure. The CALICO approach has specifically been developed to address this situation. Indeed we have designed CALICO to be extensible and thus new adaptation engines can be integrated and directly benefit from the whole design and debugging support that CALICO offers.

Several approaches exist to express QoS specifications, such as QML [15] and WSLA [16]. QML [15] is a generic QoS specification language and provides no details on how the specification must be implemented. Thus, the developer has to rewrite the specification with a programming language and develop the QoS sensors. In CALICO, the platform is automatically instrumented to enable the QoS specifications to be analyzed.

WSLA is a standard QoS specification for Web services [16]. These QoS specifications are written in XML but the general concepts to describe QoS constraints are similar to QML. Moreover, WSLA proposes that QoS specifications be handled and evaluated at runtime. The WSLA specification explains how QoS sensors must be added, how QoS events must be monitored and how QoS specifications must be evaluated. For example, QoS specifications must be evaluated by a Web service. When needed, this Web service notifies the adaptation Web service, which is in charged of performing the runtime adaptation. In CALICO, we have the same separation between the sensors, the tool to evaluate the expression and the tool to adapt the architecture. But our framework can be used with different component-based platforms. Finally, there is

no verification on the adaptation in WSLA, whereas in CALICO, only safe adaptations are applied.

8 Conclusion

We have presented a model-based framework to design and debug autonomic systems using an uniform and iterative process. It provides metamodels to allow architects to specify their system, the application properties and the adaptation rules. At debugging time, the models are kept and architects can adjust at any time the models in order to calibrate the adaptation rules or even update the application. Dynamic adaptations are controlled by an autonomic control loop driven by the models.

Furthermore, our framework is generic and highly extensible. All metamodels for specifying the structure, the application properties and the adaptation rules are independent of any underlying platform. Moreover, CALICO allows the supports of new underlying platforms and new QoS sensors. The current implementation supports three different component-based platforms. Finally, our evaluations have shown that the performance of our approach makes it usable for the debug of autonomic applications.

Future work includes several extensions. First, we plan to support more platforms, such as SCA and Web service platforms, more analysis tools and more QoS sensor frameworks. Then, we would like to add the support of high-level adaptation rules, such as moving or replacing a component. These rules will be translated into elementary modification actions, *i.e.*, adding/removing components and connectors.

References

1. Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* 36(1), 41–50 (2003)
2. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering* 26(1), 70–93 (2000)
3. Batistal, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) *EWSA 2005*. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
4. Wile, D.: Using dynamic acme. In: *Proceedings of Working Conference on Complex and Dynamic Systems Architecture* (2001)
5. Kephart, J.O.: Research challenges of autonomic computing. In: *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*, pp. 15–22. ACM, New York (2005)
6. Wagnier, G., Sriplakich, P., Le Meur, A.F., Duchien, L.: A model-based framework for statically and dynamically checking component interactions. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) *MODELS 2008*. LNCS, vol. 5301, pp. 371–385. Springer, Heidelberg (2008)
7. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) *CBSE 2004*. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
8. OMG: CORBA Component Model, v4.0, formal/06-04-01 (April 2006)
9. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: Opencom v2: A component model for building systems software. In: *IASTED Software Engineering and Applications* (2004)

10. Nunziati, S.: Personal health record, www.d-m-p.org/docs/EnglishVersionDMP.pdf
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoaka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
12. OMG: Object Constraint Language (OCL). 2.0 edn. (May 2006)
13. Plasil, F., Visnovsky, S.: Behavior protocols for software components. *IEEE Trans. Softw. Eng.* 28(11), 1056–1076 (2002)
14. Wagnier, G., Le Meur, A.F., Duchien, L.: Architectural specification and static analyses of contractual application properties. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 152–170. Springer, Heidelberg (2008)
15. Frolund, S., Koisten, J.: QML: A Language for Quality of Service Specification (1998)
16. IBM: WSLA Language Specification, V1.0. (2003)
17. (OGF), O.G.F.: Web Services Agreement Specification (WS-Agreement) (2007)
18. Paton, N.W., Schneider, F., Gries, D. (eds.): Active Rules in Database Systems (1998)
19. Tombelle, C., Vanwormhoudt, G.: Dynamic and generic manipulation of models: From introspection to scripting. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 395–409. Springer, Heidelberg (2006)
20. BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase: Assembly Component Architecture - Assembly Model Specification Version 1.00 (March 2007)
21. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, pp. 1–7. ACM, New York (2005)
22. Cecchet, E., Elmeleegy, H., Layaida, O., Quéma, V.: Implementing probes for j2ee cluster monitoring. In: Proceedings of OOPSLA 2004 Workshop on Component and Middleware Performance (2004)
23. Rouvoy, R., Conan, D., Seinturier, L.: Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online* 9(6) (2008)
24. King, T.M., Ramirez, A., Clarke, P.J., Quinones-Morales, B.: A reusable object-oriented design to support self-testable autonomic software. In: Proceedings of the 2008 ACM symposium on Applied computing (SAC 2008), pp. 1664–1669. ACM, New York (2008)
25. Li, B., Nahrstedt, K.: Dynamic reconfiguration for complex multimedia applications. In: Proceedings of Multimedia Computing and Systems, pp. 165–170 (1999)
26. Zhang, R., Lu, C., Abdelzaher, T., Stankovic, J.: Controlware: A middleware architecture for feedback control of software performance. In: Proceedings of Distributed Computing Systems, pp. 301–310 (2002)

Applying Model Transformations to Optimizing Real-Time QoS Configurations in DRE Systems

Amogh Kavimandan and Aniruddha Gokhale*

ISIS, Dept. of EECS, Vanderbilt University, Nashville, TN, USA
{amoghk, gokhale}@dre.vanderbilt.edu

Abstract. The quality of a software architecture for component¹-based distributed systems is defined not just by its source code but also by other systemic artifacts, such as the assembly, deployment, and configuration of the application components and their component middleware. In the context of distributed, real-time, and embedded (DRE) component-based systems, bin packing algorithms and schedulability analysis have been used to make deployment and configuration decisions. However, these algorithms make only coarse-grained node assignments but do not indicate how components are allocated to different middleware containers on the node, which are known to impact runtime system performance and resource consumption. This paper presents a model transformation-based algorithm that combines user-specified quality of service (QoS) requirements with the node assignments to provide a finer level of granularity and precision in the deployment and configuration decisions. A beneficial side effect of our work lies in how these decisions can be leveraged by additional backend performance optimization techniques. We evaluate our approach and compare it against the existing state-of-the-art in the context of a representative DRE system.

Keywords: Model-driven engineering, Graph/model transformations, component-based systems, deployment and configuration.

1 Introduction

Component-based software engineering (CBSE) has received much attention over the past few years to develop distributed systems including distributed, real-time, and embedded (DRE) systems, such as emergency response systems, aircraft navigation and command supervisory systems, and total shipboard computing systems. CBSE provides a simplified programming model and various mechanisms to separate functional and non-functional concerns of the system being designed, which lends it to rapid prototyping, (re-) configuration, and easier maintenance of DRE systems.

DRE systems have stringent runtime quality of service (QoS) requirements including predictable end-to-end latencies, reliability and security, among others. Naturally, the software architecture of the DRE system plays an important role in ensuring that the runtime QoS needs of DRE systems are met. In component-based DRE systems,

* Corresponding author.

¹ Our use of the term component is specific to CORBA Component Model and refers to the basic building block used to encapsulate an element of cohesive functionality.

the software architecture is defined not just by the source code of the application functionality but also by a wide range of systemic issues including the assembly of application components, their deployment on the target nodes of the system and allocation of resources such as the CPU, and the component middleware that hosts the application components.

In component middleware, such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB), a container is a central concept that hosts application components. Containers hosting DRE system components provide a high degree of configurability by allowing (1) the choice of the number of thread resources to be configured for each component, their type (*i.e.*, static or dynamic), and their attributes, such as stacksize, etc., (2) control over asynchronous event communication, and event filtering and delivery options, and (3) control over client request invocation priorities on the server component. The *configuration space* – identified by all the mechanisms for specifying system QoS and their appropriate values – becomes highly complex. Thus, making the right configuration decisions is one key factor that determines the quality of the DRE system software architecture.

Prior research in improving the quality of DRE systems software architectures has focused on: (1) analysis-driven decomposition [1] of DRE system functionality into reusable application components that can be assembled and deployed; (2) component-to-node assignment [2] and resource allocations [3], and (3) schedulability and timing analysis [4,5] to determine whether specified priority assignments are feasible for an application or not, and in turn helping in partitioning the system resources and configuring the middleware.

Despite these advances, key issues still remain unresolved in the deployment and configuration problem space of DRE system software architectures. For example, although bin packing algorithms [2] make effective decisions on component deployment, and schedulability analysis determines whether priorities of components can be honored, both these decisions are at best coarse-grained since they determine only the nodes on which the components must be deployed but do not indicate how they are deployed within the containers of the component middleware. This lack of finer-grained decisions often leads to suboptimal runtime QoS since these decisions are now left to application developers who are domain experts but often lack detail understanding of the middleware.

To address these limitations, this paper presents a heuristics-based algorithm implemented within a model-transformation [6,7] framework that combines models of user-specified QoS requirements, node assignment decisions, and priority values. It then transforms the combined models into optimal middleware configurations thereby enhancing the quality of the DRE system software architecture. Our research prototype has been implemented using the GReAT [8] model transformation framework for the Lightweight CCM (LwCCM) [9] middleware.

Two significant benefits accrue from our approach. First, by realizing the heuristic-based algorithm as an automated model transformation process, it can be seamlessly reapplied and reused during the iterative DRE system development process. Second, the configurations generated by our algorithm can be leveraged by additional backend optimization tools and techniques, such as the Physical Assembly Mapper (PAM) [10]

which reduces time and space overheads by merging collocated components at system deployment-time.

This paper is organized as follows: Section 2 discusses the challenges developers face in achieving optimal QoS configuration² for DRE systems; Section 3 presents the overall approach taken, the enabling technologies used in our technique, and the model transformation algorithm we have developed; Section 4 empirically evaluates our approach in the context of a representative case study; Section 5 discusses the related work in the area; Section 6 gives concluding remarks.

2 Impediments to the Quality of DRE System Software Architectures

We now present the deployment and configuration-imposed impediments to the quality of DRE systems software architectures. We focus on issues that are both innate to the underlying middleware platforms as well as those that are accidental.

2.1 Overview of a Real-Time Component Middleware

To better articulate the challenges we address in this paper, we first present an overview of a representative component middleware, which forms an integral part of a DRE system software architecture. Note, however, that our solution approach is general and not specific to the outlined middleware.

Figure 1 illustrates the Lightweight CORBA Component Middleware (LwCCM) [9] architecture. DRE system developers can realize large-scale DRE systems by assembling and deploying LwCCM components. The applications within these DRE systems can use publish/subscribe communication semantics (by using the component event source and sink ports) or request/response communication semantics (by using the facet and receptacle ports).

In the context of component middleware platforms, such as LwCCM, a container is an execution environment provided for hosting the components such that they can access the capabilities of the hardware, networking and software (OS and middleware) resources. In particular, containers act as a higher-level of abstraction for hosting the components in which all the developer-specified QoS policies can be properly configured. Components with similar QoS configuration specifications are hosted within the same container so that all components in that container obtain the same QoS capabilities. Note that the bin packing algorithms described earlier cannot make these fine-grained decisions.

LwCCM – and in particular its container – leverages Real-time CORBA (RTCORBA) [11] to support the real-time QoS properties. RTCORBA in turn extends traditional CORBA artifacts, such as (a) the object request broker (ORB), which mediates the request handling between clients and servers, (b) the portable object adapter (POA), which manages the lifecycle of CORBA objects, (c) stubs and skeletons, which are

² QoS configuration and QoS policy are used interchangeably throughout the remainder of the paper.

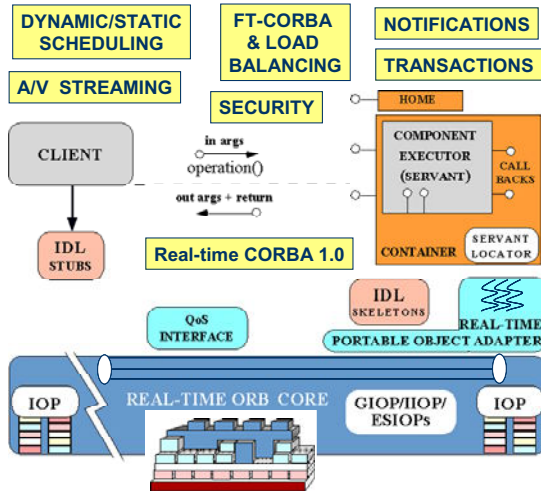


Fig. 1. Lightweight CORBA Component Model Architecture

generated by an interface definition language (IDL) compiler that hide the distribution aspects from the communicating entities, with real-time policies and interfaces.

RTCORBA (and hence LwCCM) defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools. For example, the priority at which requests must be handled can be propagated from the client to the server (the `CLIENT_PROPAGATED` model) or declared by the server (the `SERVER_DECLARED` model).

2.2 Inherent and Accidental Complexities in Deployment and Configuration

In the CBSE paradigm, application developers must determine how to deploy components within the containers, and grapple with the multiple different configuration options provided by the containers. In this context we outline two critical challenges impeding the quality of DRE system software architectures.

Challenge 1: Inherent Challenges in Deployment and Configuration. The different analyses techniques used in the development of DRE systems (*e.g.* schedulability and timing analysis) and deployment and resource allocation decisions (*e.g.*, where each component resides in the available computing node farm) dictate what QoS configurations are chosen for individual components of the application. For example, as shown earlier, LwCCM provides configuration mechanisms to assign priorities to every component, defines a fixed/variable priority request invocation and handling model (`PriorityModelPolicy`), allows defining the number of thread resources, their type (*i.e.*, static or dynamic), and concurrency options (`ThreadPool`).

For a component-based application, the mapping of the above analyses onto these available policies results in a number of unique QoS configurations, and naturally, as many containers. Unfortunately, the principles of separation of concerns in the design of containers in component middleware architectures force service request invocations between components hosted on different containers to go through the typical request demultiplexing layers and marshaling/demarshaling and mechanisms even though they may be hosted in the same address space of the application server. Therefore, such invocations are considerably slower than the invocations between components that share the same container [12].

Thus, in effect, components placed on different containers (which are in turn created from unique QoS configurations) are unable to exploit the collocation optimizations performed by the middleware.³ As such, the sub-optimal QoS configuration of the application leads to increased average end-to-end latencies. Since DRE systems are made up of hundreds of components, as the number of components in the system that are sub-optimally configured increases, the adverse impact on end-to-end latencies can be significant.

Challenge 2: Accidental Complexities in Deployment and Configuration. It may be argued that the developers can keep track of the QoS configurations that are produced, and depending on DRE system QoS needs, make decisions on how to minimize the containers. Such a manual approach, however, introduces several non-trivial challenges for the application developers:

- Large-scale DRE systems typically consist of hundreds of components spanning multiple assemblies of components. Manually keeping track of all the configurations (and potentially combining them to minimize the number of containers) in such large-scale systems is very difficult and in some cases infeasible.
- Development of DRE systems is often an iterative process where new requirements are added. Thus, the system configuration needs to evolve accordingly to cater to new requirements, and the optimizations listed above need to be performed at the end of each reconfiguration cycle.
- The configuration optimization activity forces the developers to have a detailed knowledge of the middleware platform. Further, the activity itself is not central to the development of application logic and may in fact be counter-productive to the promise of CBSE.

Addressing both these challenges calls for automated tools and techniques to perform the deployment and configuration optimizations so that the quality of the resulting DRE system software architecture is enhanced.

3 Enhancing the Quality of DRE System Software Architectures

We now present our model transformation-based approach to address the impediments to the quality of software architectures of component-based DRE systems stemming

³ Many middleware optimize the communication path for entities that reside in the same address space.

from suboptimal deployment and configuration decisions. We use a simple representative example to discuss our approach.

3.1 Representative Case Study

The Basic Single Processor (BasicSP) scenario shown in Figure 2 is a reusable component assembly available in the Boeing Bold Stroke [13] component avionics computing product line. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed using a component middleware platform, such as LwCCM.

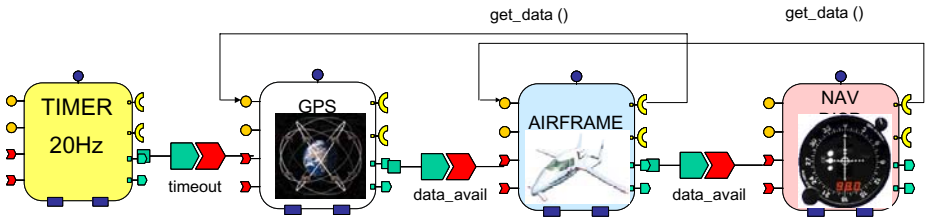


Fig. 2. Basic Single Processor Component Assembly

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario shown in Figure 2 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot.

In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Timer* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space.

For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is LwCCM) must ensure that these application-level QoS requirements are satisfied. For achieving the goal of reducing the latency between *Timer* and *NavDisplay* components, it is crucial to carefully analyze the QoS options chosen for each component in

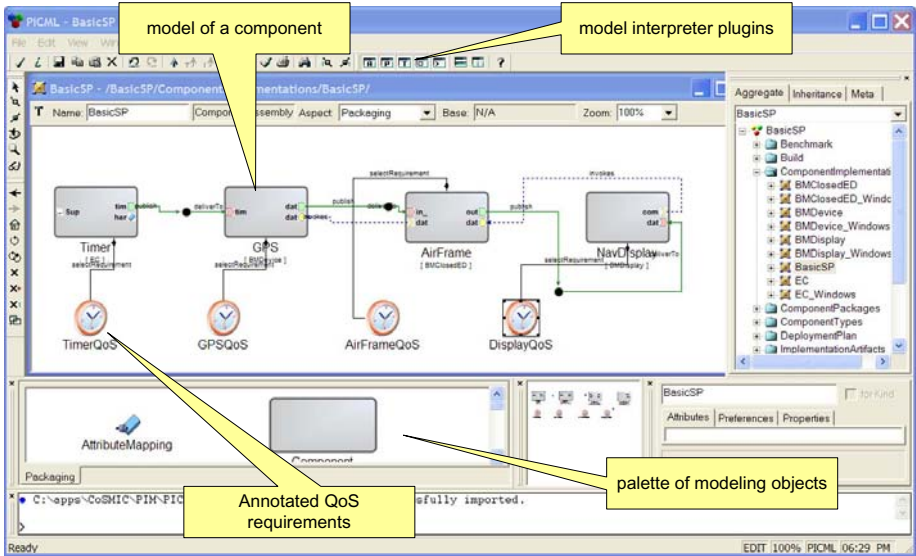


Fig. 3. BasicSP Model

BasicSP, and exploit opportunities to either reuse or combine them such that this goal can be met.

The system is deployed on two physical nodes. Application developers of our case study choose a modeling environment to model the BasicSP component assembly and annotate its real-time requirements as shown in Figure 3. We have used the Generic Modeling Environment (GME) [14] for modeling the DRE system. GME provides a graphical user interface that can be used to define both modeling language semantics and system models that conform to the languages defined in it. *Model interpreters* can be developed using the generative capabilities in GME that parse and can be used to generate deployment, and configuration artifacts for the modeled application.

The developers use the model interpreter plugins shown in the figure to automatically synthesize deployment and configuration metadata that describes how the components are assigned to nodes of the operating environment, and what configuration options of the component middleware are to be used for each component. These model interpreters encapsulate the bin packing and schedulability analyses algorithms alluded to earlier.

The generated metadata is usually in the form of XML, which is then parsed by the underlying middleware’s deployment and configuration tool to deploy and configure the DRE system before operationalizing it. As mentioned earlier, due to a lack of finer-grained decisions, the generated XML metadata will often result in DRE system software architectures that perform suboptimally.

3.2 Heuristics-Based Model-Transformation Algorithm

The model transformation algorithm we developed takes the following models and generated artifacts as its input: (1) DRE QoS requirements specification in the form of models

as shown in Figure 3, and (2) the generated DRE system deployment plan indicating the coarse-grained component-to-node mapping and configuration options to be used for the middleware. We assume that this mapping includes collocation groups, which are sets that include the components that can be placed together on a node and that too in the same address space. The objective of our algorithm is to improve the end-to-end latencies in DRE system as well as reduce the memory footprint of the DRE system by virtue of minimizing the number of containers needed to host the DRE system components.

The output of our algorithm is an enhanced QoS policy set⁴, which is incorporated into the DRE system model. Our approach produces optimized QoS policy sets by employing novel ways of reusing and/or combining existing deployment and configuration metadata and applying deployment heuristics in an application-specific manner.

We have used the Graph Rewriting And Transformation (GReAT) [8] language for defining our transformation algorithms. GReAT, which is developed using GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine). The graph rewriting rules are defined in GReAT in terms of source and target languages (*i.e.*, metamodels).

Below we explain the individual steps in our transformation process.

Step I: Modeling Language used in the Transformation Algorithm: To demonstrate our technique we required a modeling language to enable the developers to annotate their QoS requirements on the DRE system models. A simplified UML QoS configuration metamodel that we used is shown in Figure 4.

The metamodel defines the following elements corresponding to several LwCCM real-time configuration mechanisms:

- Lane, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure the number of *static* threads (*i.e.*, those that remain active till the system is running, and *dynamic* threads (*i.e.*, those threads that are created and destroyed as required) using Lane element.
- ThreadPool, which controls various settings of Lane elements, or a group thereof. These settings include `stacksize` of threads, whether borrowing of threads across two Lane elements is allowed, and maximum resources assigned to the buffer requests that cannot be immediately serviced.
- PriorityModelPolicy, which controls the policy model that a particular ThreadPool follows. It can be set to either `CLIENT_PROPAGATED` if the invocation priority is preserved, or `SERVER_DECLARED` if the server component changes the priority of invocation.
- BandedConnections, which defines separate connections for individual (client) service invocations. Thus, using BandedConnections, it is possible to define a separate connection for each (range of) service invocation priorities of a client component. The range can be defined using `low_range` and `high_range` option values of *BandedConnections*.

⁴ QoS policy set is a group of configuration files that completely capture the DRE system QoS. These files are used by the middleware to ultimately provision infrastructure resources such that the QoS requirements can be met.

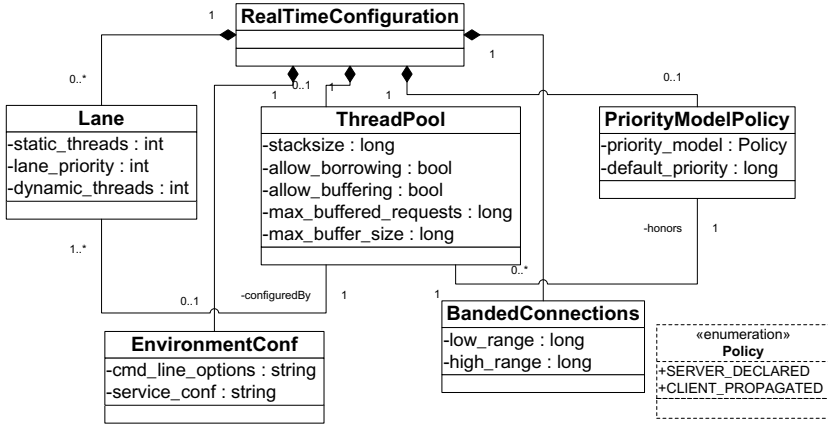


Fig. 4. Simplified UML Notation of QoS Configuration Meta-model in CQML

Step II: QoS Policy Optimization Algorithm Algorithm 1 depicts our heuristics-based model-transformation algorithm, which uses the metamodel shown in Figure 4 as its source and target language, for optimizing the deployment and configuration decisions.

The algorithm is executed for all the deployment plans specified for an application and the policy optimizations are applied for each such plan as shown in Line 2. In Line 5, all the components from a single collocation group are found.⁵ Based on whether they have SERVER_DECLARED or CLIENT_PROPAGATED priority model, they are grouped together in *SCS* and *SCC* as shown in Lines 7 and 9, respectively.

Finally, for each set of components above, the algorithm minimizes the number of QoS policies in Line 12 subject to the condition in Line 13. This condition stipulates that if QoS policies of two (sets of) components *a* and *b* each indicated in the Algorithm by qp_a and qp_b , respectively, are similar (binary Boolean function \cong finds whether the policies are similar), then they are combined (indicated by \bowtie) leading to a reduction in the size of SQ_1 .

For example, two policies would be similar if the lane borrowing feature of their ThreadPool configuration option is same. Similarly, if the value of lane_priority in Lane configuration option matches, the two Lanes are similar and can be combined. This test in the Algorithm is applied pairwise to all components in the set. The Algorithm implements symmetric rules for CLIENT_PROPAGATED policy model.

In Line 21 the results from applying all the above rules to the DRE system model are used to modify the current deployment plan, and the process is repeated for all the remaining plans of the DRE system until no more optimizations are feasible. The finalized deployment plans are then fed to a deployment and configuration tool supplied with the component middleware so that the components can be deployed and configured, and the DRE system can be activated.

⁵ Note that this is a host-based collocation group.

Algorithm 1. Transformation Algorithm for Optimizing Deployment and Configuration Metadata

Input. set of deployment plans SP ;
 set of components SC , SCS (those that use server declared policy), SCC (those that use client propagated policy);
 component c , c_p ;
 deployment plan p ;
 set of QoS policies SQ_1 , SQ_2 , qp_a (QoS policy set of a specific component 'a'), qp_b (similarly for a component 'b');
 set of collocation groups SCG ;
 collocation group g

```

1 begin
2   foreach  $p \in SP$  do
3      $SCG \leftarrow collocationGroups(p)$ ; // collect all collocation groups in the deployment plan
4     foreach  $g \in SCG$  do
5        $SC \leftarrow SC + components(g)$ ; // Collect all components of a single collocation
        group
6       if  $c \in SC \mid c.priorityModel == SERVER\_DECLARED$  then
7          $SCS \leftarrow SCS + c$ ; // Collect all components using the server declared policy
8       else if  $c \in SC \mid c.priorityModel == CLIENT\_PROPAGATED$  then
9          $SCC \leftarrow SCC + c$ ; // Collect all components using the client propagated policy
10      foreach  $c \in SCS$  do
11         $SQ_1 \leftarrow SQ_1 + c.QoSPolicy()$ ;
12        minimize  $SQ_1$ 
13        subject to  $qp_a \bowtie qp_b \mid qp_a \cong qp_b$ ;
14      end
15      foreach  $c \in SCC$  do
16         $SQ_2 \leftarrow SQ_2 + c.QoSPolicy()$ ;
17        minimize  $SQ_2$ 
18        subject to  $qp_a \bowtie qp_b \mid qp_a \cong qp_b$ ;
19      end
20    end
21     $modifyDeploymentPlan(p, SQ_1, SQ_2)$ ; // modify the plan and repeat until no more
        optimizations are feasible
22  end
23 end

```

DRE developers can subsequently test their system, and can iterate through the development lifecycle if the right end-to-end QoS is not observed or if other requirements change.

3.3 Resolving the Challenges in Optimizing QoS Configurations

At the end of step I, the developers create the application model that capture the initial QoS policies. The transformation algorithm shown in Algorithm 1 is applied in step II to that DRE system model, which updates it and generates a modified QoS configuration policies using the rules described in the algorithm.

Our automated, model transformation-based approach resolves the challenges we have discussed in Section 2 as follows: The inherent platform-specific complexities in optimizing DRE system QoS configurations are encapsulated in the transformation rules described in Section 3.2. The developers can thus focus on application business logic, and use our approach to optimize the QoS configuration. Further, the model transformation rules are reusable and can be applied repeatedly, during application development, and maintenance thereby addressing the accidental complexities.

4 Evaluating the Merits of the Transformation Algorithm

This section evaluates our approach to optimizing the original deployment and configurations for component-based DRE systems. We claim that the quality of the resulting software architecture is improved if it is able to demonstrate an improved performance.

We describe our results in the context of our case study explained in Section 3.1. We show how the end-to-end latency results after applying our algorithm achieves considerable improvement over the existing state-of-the-art. Moreover, we also demonstrate a beneficial side effect of our solution by discussing how the algorithm can be combined with additional backend optimization frameworks like the Physical Assembly Mapper (PAM) [10].

4.1 Experimental Setup and Empirical Results

We have used ISISLab (www.dre.vanderbilt.edu/ISISLab) for evaluating our approach. Each of the physical nodes used in our experiments was a 2.8 GHz Intel Xeon dual processor with 1 GB physical memory, 1 GHz network interface, and 40GB hard disks. Version 0.6 of our Real-time LwCCM middleware called CIAO was used running on Redhat Fedora Core release 4 with real-time preemption patches. The processes that hosted BasicSP components were run in the POSIX scheduling class `SCHED_FIFO`, enabling first-in-first-out scheduling semantics based on the priority of the process.

To showcase our results, we first modeled the BasicSP scenario and generated the deployment and configuration metadata for each of its components. Note that the metadata is generated using the model interpreters that encapsulate appropriate bin packing and schedulability analysis techniques. We collected the end-to-end latency metrics for the BasicSP scenario using the initial deployment and configuration metadata.

We then applied the transformation algorithm 1 to our BasicSP model which resulted in more fine-grained optimizations to the existing deployment and configuration metadata. The BasicSP scenario was then executed again with the updated QoS policies, and the results were collected. For both these experiments, the results were obtained by repeating invocations for 100,000 iterations after 10,000 warmup iterations and averaging them.

Figures 5 and 6 show the results of applying our approach to BasicSP scenario comparing them to those derived from the original deployment and configurations. The figure plots the average end-to-end latency and its standard deviation for the invocations from *Timer* to *NavDisplay* components in BasicSP with and without our approach.

As shown in Figure 5, the average latency improved by $\sim 70\%$ when our technique was used for optimizing BasicSP QoS configurations. The standard deviation on the other hand, improved by $\sim 59\%$ as plotted in Figure 6.

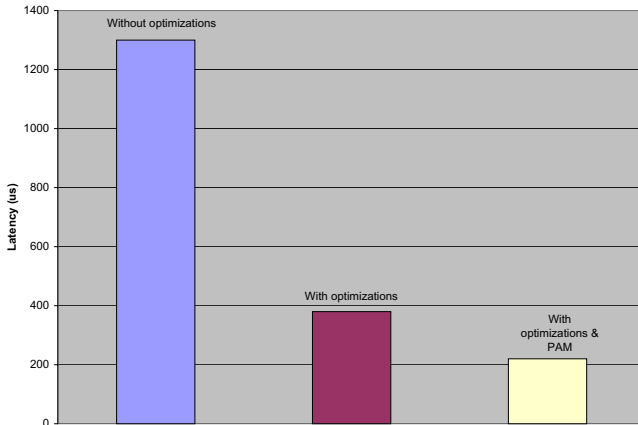


Fig. 5. Average end-to-end Latency

Without our approach, the initial BasicSP QoS configuration contained separate policies for each of its four components. Out of the four components, only the *Timer* component has `SERVER_DECLARED` priority model, while the rest of the components have `CLIENT_PROPAGATED` priority model. Thus, as indicated on Lines 12 and 13, when Algorithm 1 is applied to BasicSP, the QoS policy set is reduced to a size of two, one for each kind of priority model. This reduction in the size of the QoS policy set leads to the $\sim 70\%$ improvement in end-to-end latency between *Timer* and *NavDisplay* components.

The third graph in each figure indicates the additional improvements in end-to-end latencies accrued as a result of leveraging backend optimization frameworks, such as PAM [10]. PAM is a deployment-time technique that *fuses* a set of components collocated in a container to reduce memory footprint and latency between service invocations. Our approach simply indicates what components should be part of the same container, however, individual components continue to require their own stubs and skeletons, and other glue code, which continues to be a source of memory footprint overhead.

Approaches like PAM can then be used to eliminate this remaining overhead, and our model-transformation technique can give hints to PAM on which components should be part of the same container. Since PAM is essentially a model-driven tool, the modified DRE system QoS configuration model resulting from applying our model transformation algorithm can directly be used to investigate fusion opportunities for the application. As shown in Figures 5 and 6, when applied in conjunction with PAM, our approach leads to a combined improvement of $\sim 83\%$ in the end-to-end latency and $\sim 65\%$ in the observed standard deviation in latency for BasicSP scenario.

4.2 Discussion

Our transformation algorithm described in Sections 3.2 relies on QoS configuration analyses in a platform-specific manner. We specifically showed how it has been real-

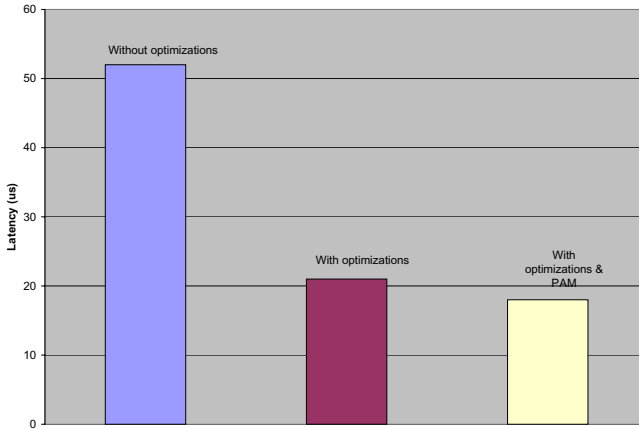


Fig. 6. Standard Deviation in Latency

ized in the context of a LwCCM middleware implementation. Naturally to extend it to other middleware platforms requires a careful study of the other platform’s configuration space.

The results indicated an improvement of $\sim 70\%$ in invocation latency between an execution path consisting of four components (the execution path here refers to the invocations from *Timer*, to *GPS*, to *AirFrame*, and finally to *NavDisplay* components in BasicSP). Recall that the BasicSP is an assembly of components that is used in the context of larger DRE system architectures. With increasing scale of the DRE system, it becomes necessary to leverage every opportunity for optimizations.

We expect the improvements accrued using our approach to be even higher. This is because the reduction in end-to-end latency is dependent upon how effectively the QoS policy sets SQ_1 and SQ_2 in Algorithm 1 are minimized. Large-scale DRE systems would have a number of QoS policies specified across their component assemblies, and in general, would be expected to have more opportunities to combine and reuse these policies leading to further latency improvements.

5 Related Work

Since the work presented in this paper results in performance optimizations to the underlying middleware thereby improving the quality of the DRE system software architecture, we compare our work with synergistic works. Moreover, since our research is applicable at design-time, we focus primarily on design- and deployment-time techniques to compare our work against.

A significant amount of prior work has focused on estimating the performance of software architectures via prediction techniques thereby allowing architects to weed out bad architectural choices. In this paper we have not applied these techniques. We assume that the architectures of the underlying middleware are sound. Our objective is to optimize performance even further by combining similar QoS policies.

Good software engineering principles argue for improving the functional cohesion and decreasing functional decoupling in software systems. The work presented here is similar in spirit. We strive to improve QoS cohesion by combining QoS policies that are similar. In doing so we continue to preserve the functional cohesion and decoupling provided by the container mechanisms in component middleware.

Design-time approaches to component middleware optimization include eliminating the need for dynamic loading of component implementation shared libraries and establishing connections between components done at runtime, as described in the static configuration of CIAO [15]. Our approach is different since it uses model transformations of configurations at design-time. Our approach is thus not restricted to optimizing just the inter-connections between components. Moreover, the static configuration approach can be applied in combination with our approach.

Another approach to optimizing the middleware at design/development-time employs context-specific middleware specializations for product-line architectures [16]. This work is based on utilizing application-, middleware- and platform-level properties that do not vary during the normal application execution in order to reduce the excessive overhead caused by the generality of middleware platforms.

Some work has also been done in the area of Aspect-Oriented Programming (AOP) techniques that rely essentially on automatically deriving subsets of middleware based on the use-case requirements [17], and modifying applications to bypass middleware layers using aspect-oriented extensions to CORBA Interface Definition Language (IDL) [18]. In addition, middleware has been synthesized in a “just-in-time” fashion by integrating source code analysis, and inferring features and synthesizing implementations [19].

Contrary to the above approaches, our model transformation-based technique relies only on (1) the specified QoS requirements specification and (2) the initial deployment plan, so that the QoS configurations can be optimized. Our approach does not necessitate any modifications to the application, *i.e.*, the application developer need not design his/her application tuned for a specific deployment scenario. As our results in Section 4.2 have indicated, our approach can be used in a complementary fashion to any of the design/development-time approaches discussed above since there exist several opportunities for QoS optimization at various stages in application development.

Deployment-time optimizations research includes BluePencil [20], which is a framework for deployment-time optimization of web services. BluePencil focuses on optimizing the client-server binding selection using a set of rules stored in a policy repository and rewriting the application code to use the optimized binding. While conceptually similar, our approach differs from BluePencil because it uses models of application structure and application deployment to serve as the basis for the optimization infrastructure.

BluePencil uses techniques such as *configuration discovery* that extract deployment information from configuration files present in individual component packages. By operating at the level of individual client-server combinations, the QoS optimizations achieved in our transformation-based approach are non-trivial to perform in BluePencil. BluePencil also relies on modification to the application source code to rewrite the application code, while our approach is non-intrusive and does not require application

source code modifications, and it only relies on the specified application policies and deployment plans.

Research on approaches to optimizing middleware at runtime has focused on choosing optimal component implementations from a set of available alternatives based on the current execution context [21]. QuO [22] is a dynamic QoS framework that allows dynamic adaptation of desired behavior specified in *contracts*, selected using proxy objects called *delegates* with inputs from runtime monitoring of resources by *system condition* objects. QuO has been integrated into component middleware technologies, such as LwCCM.

Other aspects of runtime optimization of middleware include domain-specific middleware scheduling optimizations for DRE systems [23], using feedback control theory to affect server resource allocation in Internet servers [24] as well as to perform real-time scheduling in Real-time CORBA middleware [25]. Our work is targeted at optimizing the middleware resources required to host composition of components in the presence of a large number of components, whereas, the main focus of these related efforts is to either build the middleware to satisfy certain performance guarantees, or effect adaptations via the middleware depending upon changing conditions at runtime.

6 Concluding Remarks

The last few years have seen a significant increase in the popularity of component middleware platforms for developing distributed, real-time and embedded (DRE) systems, such as emergency response systems, intelligent transportation systems, total shipboard computing environment across a wide range of application domains. Its higher levels of programming abstractions coupled with mechanisms that support sophisticated and highly tunable infrastructure configuration are well suited for rapid development and/or maintenance of such systems.

The generality of contemporary component middleware platforms, however, has increased the complexity in properly configuring these platforms to meet application-level QoS requirements. Automated solutions [26] are an attractive alternative to achieving the QoS configuration of component-based systems, however, they incur excessive system resource overheads often leading to sub-optimal system QoS.

In this paper, we discussed an automated, model transformation-based approach that takes into account the component collocation heuristics to optimize application QoS configuration thereby improving the quality of the software architecture. We discussed the design of our approach, and the transformation algorithm used to optimize the QoS configuration. We also evaluated our approach and compare it against the existing state-of-the-art. The results demonstrated the effectiveness of our approach in optimizing QoS configuration in the context of a representative DRE system reusable component assembly.

The following are the lessons learned from our research:

- Optimal QoS configuration for component-based systems is a crucial research area that has been unaddressed till date. As component middleware gains popularity, and available resources become constrained, especially in the context of DRE systems, it is critical to improve the overall quality of the DRE system software architectures.

- Existing research in QoS configuration have focused largely on achieving *locally optimized* solutions, *i.e.*, they are restricted to analyzing and modifying/manipulating the middleware configuration space.
- Our approach showed that excessive overheads can be avoided by analyzing QoS configurations in the context of other application characteristics such as its component collocation/node placement heuristics.
- We have focused on combining the deployment decisions with the application QoS specification. However, as our results have indicated, further optimizations are possible by combining our technique with other design-, development-, run-time techniques, which merits further investigation. Additional investigations are also necessary to test our approach on larger DRE systems and different middleware platforms.
- Our approach indicated improvements in latencies. Significant research remains to be done to see how other QoS metrics can be improved as well. When multiple QoS metrics are considered together, simple heuristics may not work. Instead multi-objective optimizations [27] may be necessary. Additionally, we have not demonstrated the applicability of our work for ultra-large scale DRE systems. This will form part of our future research.
- Our approach is not necessarily restricted to real-time systems. There may be opportunities for applying these arguments to other domains including high performance computing and enterprise computing.

Acknowledgments. We thank the anonymous reviewers for the excellent feedback. We also thank the National Science Foundation. Although this work was not directly funded by NSF, it led to the CAREER award (number 0845789), which is pursuing research in the future work studies indicated above.

References

1. Hatcliff, J., Deng, W., Dwyer, M., Jung, G., Prasad, V.: Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In: Proceedings of the 25th International Conference on Software Engineering, Portland, OR, pp. 160–172 (May 2003)
2. de Niz, D., Rajkumar, R.: Partitioning Bin-Packing Algorithms for Distributed Real-time Systems. *International Journal of Embedded Systems* 2(3), 196–208 (2006)
3. Ghosh, S., Rajkumar, R., Hansen, J., Lehoczy, J.: Integrated QoS-aware Resource Management and Scheduling with Multi-resource Constraints. *Real-Time Syst.* 33(1-3), 7–46 (2006)
4. Stankovic, J.A., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: VEST: An Aspect-Based Composition Tool for Real-Time Systems. In: RTAS 2003: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, pp. 58–69. IEEE Computer Society, Los Alamitos (2003)
5. Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In: RTAS 2003: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, pp. 78–85. IEEE Computer Society, Los Alamitos (2003)
6. Rozenberg, G.: *Handbook of Graph Grammars and Computing by Graph Transformation*. Foundations, vol. 1. World Scientific Publishing Company, Singapore (1997)

7. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software* 20(5), 42–45 (2003)
8. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science* 9(11), 1296–1321 (2003), www.jucs.org/jucs_9_11/on_the_use_of
9. Object Management Group: Lightweight CCM RFP. *realtime/02-11-27* edn. (November 2002)
10. Balasubramanian, K., Schmidt, D.C.: Physical Assembly Mapper: A Model-driven Optimization Tool for QoS-enabled Component Middleware. In: *Proceedings of the 14th IEEE Real-time and Embedded Technology and Applications Symposium*, St. Louis, MO, USA, pp. 123–134 (April 2008)
11. Object Management Group: Real-time CORBA Specification. 1.2 edn. (January 2005)
12. Wang, N., Schmidt, D.C., Parameswaran, K., Kircher, M.: Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In: *24th Computer Software and Applications Conference (COMPSAC)*, Taipei, Taiwan, pp. 492–499. IEEE, Los Alamitos (2000)
13. Sharp, D.C.: Reducing Avionics Software Cost Through Component Based Product Line Development. In: *Software Product Lines: Experience and Research Directions*, vol. 576, pp. 353–370 (August 2000)
14. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. *Computer* 34(11), 44–51 (2001)
15. Subramonian, V., Shen, L.J., Gill, C., Wang, N.: The Design and Performance of Configurable Component Middleware for Distributed Real-Time and Embedded Systems. In: *RTSS 2004: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)*, Lisbon, Portugal, pp. 252–261. IEEE Computer Society, Los Alamitos (2004)
16. Krishna, A., Gokhale, A., Schmidt, D.C., Hatcliff, J., Ranganath, V.: Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In: *Proceedings of EuroSys 2006*, Leuven, Belgium, pp. 205–218 (April 2006)
17. Hunleth, F., Cytron, R.K.: Footprint and Feature Management Using Aspect-oriented Programming Techniques. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 2002)*, Berlin, Germany, pp. 38–45. ACM Press, New York (2002)
18. Demir, Ö.E., Dévanbu, P., Wohlstadter, E., Tai, S.: An Aspect-oriented Approach to Bypassing Middleware Layers. In: *AOSD 2007: Proceedings of the 6th international conference on Aspect-oriented software development*, Vancouver, British Columbia, Canada, pp. 25–35. ACM Press, New York (2007)
19. Zhang, C., Gao, D., Jacobsen, H.A.: Towards Just-in-time Middleware Architectures. In: *AOSD 2005: Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, Illinois, pp. 63–74. ACM Press, New York (2005)
20. Lee, S., Lee, K.W., Ryu, K.D., Choi, J.D., Verma, D.: ISE01-4: Deployment Time Performance Optimization of Internet Services. In: *Global Telecommunications Conference, 2006. GLOBECOM 2006*, pp. 1–6. IEEE, San Francisco (2006)
21. Diaconescu, A., Mos, A., Murphy, J.: Automatic performance management in component based software systems. In: *ICAC 2004: Proceedings of the First International Conference on Autonomic Computing (ICAC 2004)*, New York, NY, USA, pp. 214–221. IEEE Computer Society, Los Alamitos (2004)
22. Zinky, J.A., Bakken, D.E., Schantz, R.: Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems* 3(1), 1–20 (1997)
23. Gill, C.D., Cytron, R., Schmidt, D.C.: Middleware Scheduling Optimization Techniques for Distributed Real-time and Embedded Systems. In: *Proceedings of the 7th Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA. IEEE, Los Alamitos (2002)

24. Zhang, R., Lu, C., Abdelzaher, T.F., Stankovic, J.A.: ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In: ICDCS 2002: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, pp. 301–310 (2002)
25. Lu, C., Stankovic, J.A., Son, S.H., Tao, G.: Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Real-Time Syst* 23(1-2), 85–126 (2002)
26. Kavimandan, A., Gokhale, A.: Automated Middleware QoS Configuration Techniques using Model Transformations. In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), St. Louis, MO, USA, pp. 93–102 (April 2008)
27. Deb, K., Gupta, H.: Searching for Robust Pareto-Optimal Solutions in Multi-objective Optimization. In: Coello Coello, C.A., Hernández Aguirre, A., Zitzler, E. (eds.) EMO 2005. LNCS, vol. 3410, pp. 150–164. Springer, Heidelberg (2005)

Automated Architecture Consistency Checking for Model Driven Software Development

Matthias Biehl^{1,2} and Welf Löwe¹

¹ Software Technology Group, Växjö University, Sweden
Welf.Lowe@msi.vxu.se

² Embedded Control Systems, Royal Institute of Technology, Sweden
biehl@md.kth.se

Abstract. When software projects evolve their actual implementation and their intended architecture may drift apart resulting in problems for further maintenance. As a countermeasure it is good software engineering practice to check the implementation against the architectural description for consistency. In this work we check software developed by a Model Driven Software Development (MDSO) process. This allows us to completely automate consistency checking by deducing information from implementation, design documents, and model transformations. We have applied our approach on a Java project and found several inconsistencies hinting at design problems. With our approach we can find inconsistencies early, keep the artifacts of an MDSO process consistent, and, thus, improve the maintainability and understandability of the software.

1 Introduction

In a typical software development project several artifacts are created and changed independently, such as design documents and source code. Time pressure often leads to unresolved divergences between these artifacts. As a result, the system becomes harder to understand, making further maintenance tasks more complicated and costly [26]. Perry and Wolf are among the first to name and discuss this problem as architectural drift and architectural erosion [23]. New development methodologies and techniques such as Model Driven Software Development (MDSO) and Model Driven Architecture (MDA) seem at first glance to solve inconsistency problems. Typically, the development process starts with the manual design of a high-level artifact which is subsequently automatically transformed into a low-level artifact such as source code.

One might assume the transformation ensures that high-level artifacts are consistently transformed into low-level artifacts. However, the transformations do not create the entire set of low-level artifacts; they rather create skeletons that need to be extended and completed manually. Due to this semi-automation, projects developed with MDSO are also prone to the problem of architectural drift; inconsistencies may be introduced for the following reasons:

Incorrect transformations. Data of high-level models may be lost or misinterpreted during transformation to low-level models.

Manual additions. The implementation is usually not completely generated. Developers need to add code into the generated skeletons. Sometimes even new classes not even present in the high-level artifact need to be added. Thus code in manual additions may diverge from design documents.

Synchronization. Design documents and implementation may get out of sync, when the design documents are changed without subsequently generating the source code.

The detection of inconsistencies is a first step towards fixing these problems. Existing approaches [4,3] are general but semi-automated. We focus on software developed by MDSD and this reduction of generality allows for a fully automated solution. We contribute with an approach for consistency checking that is automated, thus requiring a minimum of user interaction and no additional user-supplied data. Our approach is flexible regarding the languages of the artifacts and the description of inconsistencies.

The remainder of this article is structured as follows: In section 2 we give a short introduction to the technology supporting our work. In section 3 we describe our approach for automated architecture consistency checking. We perform a case study with a Java software system and present the results in section 4. In section 5 we briefly describe related work and evaluate if and how these approaches suit our goals. We conclude with a summary and pointers to future work in section 6.

2 Terms and Technology

Model Driven Software Development (MDSD) is an approach for software development using models as primary artifacts of a software system [25]. During development a series of such models is created, specified, refined and transformed. Model transformations describe the relationship between models, more specifically the mapping of information from one model to another one. In this work we narrow the focus for software architecture consistency checking on software that has been developed using MDSD.

In theory the developer creates a model and automatically transforms it into an implementation, and, hence, never needs to touch the implementation. However, in practice the MDSD approach is semi-automated. Only parts of the implementation are generated, other parts require manual work. Manually created classes are not generated at all, but just added to the implementation. They cannot be mapped directly to any corresponding high-level model element. Other classes are only partly generated. The generated part is called skeleton, the manually created part of the implementation is called manual addition.

Aspect-oriented Programming (AOP) attempts to support programmers in the separation of concerns: functionality can be partitioned in cross-cutting concerns in an orthogonal way to the partitioning by modules [12]. Cross-cutting concerns, so-called aspects are selectively weaved into different places (join points) inside one or several software modules. In this work AOP is used for tracing in model-to-text transformations.

The **Software Reflexion Model** is a manual process for re-engineering and program understanding, designed to check a high-level, conceptual architecture for consistency against the low-level, concrete architecture extracted from the source code [17]. It checks the *vertical* consistency of artifacts on different stages of the development process, i.e., design and implementation.¹ Moreover, it checks *structural* consistency, more specifically, architectural consistency, which is based on the structure on an architectural level, i.e., modules and dependencies between those modules.²

For a first try assume that two artifacts are *architecturally consistent* if corresponding high- and low-level entities also have corresponding dependencies. Dependencies include relations like usage (access and invocation), aggregation, and delegation. A high-level entity corresponds to the low-level entities generated from it in the MDSD process. Besides these directly mapped entities, the corresponds also includes inheriting and auxiliary low-level entities. For an exact definition of consistency we refer to section 3.3.

According to the Software Reflexion Model, the analyst first creates a hypothesized high-level model based on information of the (architecture and design) documentation. Next, the analyst uses a fact extractor to create a low-level model of the software from the source code. Both models are graphs with nodes representing program entities and edges representing relations between them. To link the two models, the analyst manually maps low-level entities to their corresponding high-level entities. The mapping connects the two graphs to a single reflexion graph. Relational algebra is used to specify inconsistency rules between the high-level and the low-level model. The software reflexion model has been successfully used to perform design conformance tests [16]. A semi-automated approach has been proposed which uses clustering techniques (see below) to support the user in the mapping activity [4,3]. In this work, we extend the Software Reflexion Model to fully automate this process for MDSD applications.

Clustering for Architecture Recovery Clustering is a technique for finding groups of similar data elements in a large set of data. Architecture recovery attempts to reconstruct the high-level architecture of a software system based on information extracted from low-level artifacts such as the source code. In general, these approaches employ an abstract model of the structure of a software system, which consists of its basic entities such as functions, classes or files and relationships between them such as dependencies. Clustering is used to group related entities of the software system into subsystem [28,1,27]. Algorithms try to minimize the inter-cluster dependencies and maximizing the intra-cluster dependencies. The idea is to optimize low coupling and high cohesion for good subsystem decomposition [22]. The problem of finding an optimal decomposition is in NP-hard [10], and hence, heuristics are applied. In this work we use

¹ In contrast, *horizontal* consistency, is concerned with the consistency of documents of the same stage of a software development process, e.g., comparing UML sequence and class diagrams.

² In contrast, *behavioral* consistency compares two behavioral descriptions, for example UML sequence diagrams and execution states.

clustering for architecture recovery in order to group classes not generated under the MDSD process for relating them to model entities automatically.

3 Approach

In this work we specialize in checking architecture consistency for software developed with MDSD. In contrast to traditional software development, MDSD with its formalized process allows us to automate the checking for architectural violations. The MDSD process supplies us with: (1) a high-level artifact such as a UML class diagram, (2) a corresponding low-level artifact such as source code classes and (3) a transformation that maps high-level entities to low-level entities. Our solution is based on extending the Software Reflexion Model and tailoring it for the analysis of MDSD projects so we can automate it. As discussed, the MDSD process provides us with three different information sources, which can be associated with an input to the Software Reflexion Model: The high-level view of the Software Reflexion Model corresponds to the UML diagrams of MDSD, low-level view of the Software Reflexion Model corresponds to the source code of MDSD, and the mapping of the Software Reflexion Model corresponds to the transformation of MDSD.

The major part of the information needed for automated consistency checking can be extracted from the artifacts of the MDSD project. We can extract the low-level and high-level views from a program's source code and from the UML model, respectively.

Moreover, we can partially extract the mapping from the model transformation. The mapping relation captures the correspondence between low-level and high-level entities. This correspondence is established by the creation of the low-level elements during software development. In MDSD, low-level elements are created in two ways: (i) generation by a model transformation from a high-level entity (ii) manual creation by a developer. In an MDSD project the majority of the low-level entities are generated, manually created low-level entities are the exception. The two different ways of creating low-level entities entail two different approaches for automatic mapping creation. The mapping of generated low-level entities (i) is determined by the high-level entity it is transformed from. In order to extract the relevant information from the transformation, we have to study the type of the transformation and its properties. We describe this approach in section 3.1. The remaining unmapped low-level entities (ii) are manually created and there is no direct, explicit evidence about their mapping available in the artifacts of the system. This is why we rely on clues and heuristics to find their mapping. The clues are provided by the incomplete, extracted mapping and the extracted low-level dependency graph. We combine these clues to create a mapping by using a clustering technique for architecture recovery. We describe this approach in section 3.2.

Input to the consistency checking process is simply the MDSD project consisting of three parts: (1) A design document in form of a UML class diagram, (2) the source code including both skeletons and manually added code and a (3)

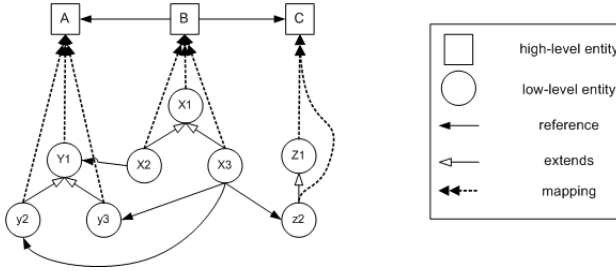


Fig. 1. Example of a complete analysis graph

model transformation, that generates parts of the source code from the design document. The desired output of the consistency checking process is a list of inconsistencies.

To get from the input to the output, we need to undertake three major steps: In the first step, we build a data structure, called analysis graph, that contains all the relevant information. We build the analysis graph according to the information contained in the MDSD project, cf. section 3.1. In the second step, we use a clustering technique to complete the information of the analysis graph, cf. section 3.2. In the third step, we use the complete analysis graph to find inconsistencies, cf. section 3.3.

3.1 Analysis Graph Extraction

The analysis graph is the central data structure of our approach for checking consistency, it is used in all major steps of the analysis process. The analysis graph contains only information relevant for solving the problem of consistency checking. It is a directed graph, consisting of two types of nodes and three types of edges. Nodes are either high-level, corresponding to the entities in a high-level design description such as UML class diagrams or low-level entities corresponding to source entities like compilation units or classes. The different edge types are: dependency edges representing references, hierarchical edges representing inheritance relationships and mapping edges representing the correspondence between high-level and low-level entities. Hierarchical and dependency edges only exist between nodes of the same level. The analysis graph is *complete*, if all low-level nodes are mapped to high-level nodes, i.e. the mapping function is defined over the whole domain of low-level nodes.

We can split the analysis graph construction into the following subtasks:

Fact Extraction from High-level Artifacts. Our fact extractor reads UML class diagrams from their XMI representation and delivers a high-level dependency graph.

Mapping Creation by Tracing. We log the execution of the transformation and find the mapping between high- and low-level model entities. Our AOP approach adds tracing code to the transformation program. Under transformation execution, information about the high-level entities is added to the

generated low-level entities in the form of annotations in the generated Java code. These annotations are later processed by the low-level fact extractor.

Fact Extraction from Low-level Artifacts. The low-level program structure is extracted from the Java source code. We reuse the existing fact extractor based on the VizzAnalyzer API. We extend the fact extractor to read the annotations produced under transformation by the tracing aspect. As output, the fact extractor delivers a low-level dependency graph partially mapped to the high-level graph.

Fact Extraction from High-Level Artifacts. In this step, we extract the relevant information from high-level artifacts. High-level artifacts such as design documents are expressed in UML. Thus we create a fact extractor for UML class diagrams. For the design of the UML fact extractor we set the following goals: (1) reuse of existing infrastructure, (2) support for different XMI versions, and (3) simple rules for fact extraction.

While standard fact extractors analyze the textual representation of a program with specialized parsers, we use a model transformation approach for fact extraction. Input of the transformation is a XMI representation of a UML class diagram, the output is a GML³ description of the analysis graph. This approach has several advantages regarding our goals for the UML fact extractor. It allows us to reuse the UML reader of the model transformation tool (1). This third party UML reader is mature, supporting different UML versions and different XMI versions (2). In the transformation we simply define a mapping between the UML elements as input and GML elements as output (3).

The UML diagrams we use as input contain more information than needed for our analysis. Thus we need to lift the extracted information to an appropriate level of abstraction, that only contains the relevant information used in later analysis. The table below shows the relevant UML elements and their counterpart in the analysis graph.

UML Element	Analysis Graph
uml::class	high-level node
uml::interface	high-level node
uml::usage	reference edge
uml::property	reference edge
uml::dependency	reference edge
uml::realization	hierarchy edge
uml::generalization	hierarchy edge

Mapping Creation by Tracing. Tracing keeps track of the relation between source elements and the created target elements of a model transformation. The result of tracing is a list of source and target element pairs of the transformation.

³ The Graph Modeling Language (GML) is a flexible and portable file format for graphs [11]. We use GML as an interface between the fact extractors and the rest of the analysis process. Thus the fact extractors can be exchanged easily.

Current template-based model-to-text transformations do not have built-in support for tracing [5], so we need to develop our own tracing solution. The goals for our tracing solution are automation and non-invasiveness. Automation of this subtask for consistency checking allows automation of the complete consistency checking process. Non-invasiveness ensures, that tracing does not change the transformation and thereby alters the object under study. Non-invasiveness also does not allow us to patch the transformation engine, so we are independent of any particular transformation engine implementation.

Possible practical solutions for tracing in model-to-text transformations are static analysis of the transformation code, instrumentation of the transformation and manipulation of the transformation engine. Since we extract the mapping from the trace of the transformation, our approach depends on the specific properties of the transformation: the transformation is (1) rule-based and (2) model-to-text⁴. These properties can be seen as constraints for the solution. Since we want our solution to be independent of the transformation engine, we regard the transformation engine as black box, thus ruling out the solution requiring manipulation of the source code of the transformation engine. The transformation code is rule-based (1), thus the static analysis of the transformation code is insufficient for providing exact mapping information. The actual mapping depends not only on the transformation rules, but also on the matching algorithm of the rules and the input. Thus, we extract the mapping information during the execution of the transformation and not just by static analysis of the transformation. To acquire this mapping information, we instrument the transformation code. However, the instrumentation of the transformation code has to be automated. The transformation is a model-to-text transformation (2). This means, that no parsing information (e.g., an abstract syntax tree) about the generated code is available during the execution of the transformation, even more so as we cannot change the transformation engine.

We solve the problems induced by (1) by using AOP. The instrumentation of the transformation with tracing code can be regarded as a cross-cutting concern, that needs to be woven into each transformation rule. We build a tracing aspect that contains code to log the transformation, i.e. it writes the name of the source model element as a comment into the target output stream. The same aspect can be applied automatically to arbitrary MDSD transformations without manually changing the transformation code.

Problems induced by (2) are solved by splitting up the tracing process in two phases: an annotation phase and a mapping extraction phase. The annotation phase takes place during the execution of the transformation. It weaves aspect code into every transformation rule. The aspect code writes the name of the current high-level element as a proper comment of the target programming language – Java in our case – into the output stream. In fact we use a Java Annotation as comment. The mapping extraction phase is part of the Java fact extraction. In this phase we read the annotations in the Java source code that were produced by the annotation phase. We connect the name of the high-level entity in the

⁴ In contrast to model-to-model transformations.

annotation to the closest low-level entity in the Java code. Since the source code is parsed, the low-level entity is now available during Java fact extraction.

Fact Extraction from Low-Level Artifacts. To obtain the dependency graph from the low-level artifacts we use a fact extractor for the appropriate programming language. A fact extractor processes the artifacts of the software system under study to obtain problem-relevant facts. These are stored in fact-bases and used to determine particular views of the program.

We keep our overall approach flexible enough to support fact extractors for any object-oriented programming language, the current implementation, however, is limited to Java. We achieve this flexibility by a well-defined interface between the fact extractor and the rest of the analysis. The interface is the file in GML format containing the dependency graph.

The goals for the low-level fact extraction are: (1) reuse of existing libraries for fact extraction, and (2) extensibility for the extraction of tracing information and (3) compatibility with graphs from high-level fact extraction which are represented in GML. A lot of fact extractors are available for the chosen implementation language Java. We choose the VizzAnalyzer fact extractor [14], since it fulfills all of our goal criteria: We can reuse it (1), since the source code is available, we can extend it (2), and it has export functions for GML (3).

The low-level fact extractor not only extracts the low-level dependency graph, but also the mapping information from the annotated source code. As explained before, annotations are inserted into the Java code whenever a transformation rule is executed. The content of the annotation includes the name of the high-level entity that is connected to this transformation rule. The full annotation consists of a comment in the appropriate low-level language and the name of the high-level entity. By putting the information in a comment, we ensure that the functionality of the source code is not modified. The fact extractor processes the annotated Java code and reads the tracing comments that are placed in front of each class. In this way the name of the high-level element from the comment and the name of the low-level element currently processed by the fact extractor are brought together defining the mapping.

3.2 Analysis Graph Completion by Clustering

For consistency checking we need a correspondence between low-level and high-level entities. This mapping needs to be complete, i.e. the mapping assigns a high-level entity to each low-level entity in the system. The mapping creation by tracing presented before can only provide such a complete mapping, if 100% of the source code was generated, especially if no new classes were added manually in the source code. In practice, only a fraction of the source code is generated, the rest is created manually, either in form of manual additions inside generated skeletons or completely manually developed classes. Manually created classes have to be either (1) excluded from the analysis or (2) need to be mapped to a high-level entity.

Option (1) circumvents the problem. We can configure the source code fact extractor in such a way that it excludes classes from the analysis. However we

only recommend this option for library code. All other source code elements should be treated according to option (2), which aims at solving the problem. We automatically map single, manually introduced source code classes to a high-level entity, i.e., we assume that these classes are part of the implementation of an abstract concept as defined with the high-level entity. Since information about the mapping is not explicitly provided, we need to rely on clues. Under a reasonable hypothesis, we then combine these clues to approximate the mapping as intended by the system developers.

For the mapping completion we follow the hypothesis of Koschke et al. [4,3]: *Source code entities that are strongly dependent on each other form a module, and all elements of a module are mapped to the same high-level entity.*

For each unmapped source code element, we need to find a mapped source code element that has a strong dependency relation to it. This is similar to the problem of automated software modularization, so we can use the techniques applied in this field, especially the ideas of clustering for architecture recovery (see section 2).

In the following we describe our clustering algorithm. It is important to distinguish mapped and unmapped source code entities. Each mapped source code entity is by definition related to a high-level entity; for unmapped source code entities no such relation exists.

1. Initially we cluster the mapped source code entities. All mapped source code entities with a mapping to the same high-level entity form a cluster.
2. We assign unmapped entities that have a mapped superclass to the same cluster as their superclass. We apply this rule recursively for all unmapped direct and indirect subclasses. Here we use inheritance information as clues for the clustering and apply our hypothesis on the inheritance hierarchy.
3. We terminate if there are no unmapped entities. Otherwise, we assign the still unmapped entities to one of the existing clusters. If several unmapped entities are available, we begin with those that are at the root of the inheritance tree. We choose their cluster based on the strength of the connection of the unmapped entity to already mapped hence clustered entities. We assign it to the cluster that has the strongest (accumulated over contained entities) connection. A connection between two entities is established by a dependency relation or reference. A connection between an entity and a cluster is the accumulated connection between the entity and the member entities of the cluster. The strength of a connection is determined by the number of connections between the entity and the cluster. Here we use the dependency relation as a clue and apply our hypothesis on the dependency relation.
4. When a new entity was mapped in step 3, we assign its unmapped subclasses according to step 2 or, otherwise, we terminate if there are no unmapped entities.

The above clustering algorithm assigns all source code entities to a cluster. Due to step 1 all clusters have exactly one high-level element that some of their elements are mapped to. We map all source code entities in the cluster to this high-level element. As a result we get the complete analysis graph.

3.3 Check Consistency Rules

Once the analysis graph is complete, we can perform consistency checks. A consistency check searches the analysis graph for patterns of inconsistency. In this section we discuss the chosen search mechanism and the search criteria. We refer to these search criteria for inconsistency patterns as *inconsistency rules*.

Inconsistency Rules. First we need to transform the analysis graph into a representation suitable for our search mechanism. For our relational algebra approach we transform the analysis graph into a set of binary relations. Below we list the types of relations representing facts from our analysis graph. Based on these facts, we can check our inconsistency rules efficiently.

$ll(X) \Leftrightarrow$ entity X is a low-level entity (e.g., extracted from Java)

$hl(A) \Leftrightarrow$ entity A is a high-level entity (e.g., extracted from UML)

$ref(A, B) \Leftrightarrow$ entity A references entity B

$inherit(A, B) \Leftrightarrow$ entity A extends entity B

$inherit^*(A, B)$ reflexive, transitive closure of $inherit(A, B)$

$map(X, A) \Leftrightarrow$ low-level entity X is mapped to high-level entity A

In the following we explain the most common definition of inconsistency patterns according to [13]: *absence* and *divergence* inconsistencies. An absence inconsistency is defined as a subgraph consisting of a high-level reference with no corresponding low-level reference. This can happen when a high-level model is updated by adding a reference but the existing source code based on the previous model is kept and not newly generated after the update. A divergence inconsistency is defined as a subgraph consisting of a low-level reference with no corresponding high-level reference. This may happen when the source code is changed without updating the model. Below we have formalized these informal descriptions of the two types of inconsistency in relational algebra.

$absence(A, B) \Leftarrow hl(A) \wedge hl(B) \wedge hlref(A, B) \wedge \neg llref(A, B)$

$divergence(A, B) \Leftarrow hl(A) \wedge hl(B) \wedge llref(A, B) \wedge \neg hlref(A, B)$

We use $hlref(A, B)$ as an auxiliary relation containing all pairs of high-level entities A and B with direct dependencies or dependencies between inheriting high-level entities. Similarly, $llref(A, B)$ denotes all pairs of high-level entities A and B where there is a corresponding low-level pair in a dependency relation:

$hlref(A, B) \Leftarrow \exists A', B' : hl(A') \wedge hl(B') \wedge$

$inherit^*(A, A') \wedge inherit^*(B, B') \wedge ref(A', B')$

$llref(A, B) \Leftarrow \exists A', B', X, Y : hl(A') \wedge hl(B') \wedge ll(X) \wedge ll(Y) \wedge ref(X, Y) \wedge$

$map(X, A') \wedge inherit^*(A, A') \wedge map(Y, B') \wedge inherit^*(B, B')$

The search criteria for inconsistency patterns may differ from project to project. Some projects may require a stricter definition of consistency than others. This

is why we required the inconsistency patterns to be user-definable. The inconsistency rules are kept separately and can be changed independently. The user does not need to recompile the analyzer, if a change of the inconsistency rules is necessary. A reasonable set of inconsistencies to check for is provided above. Executing the check results in a list of inconsistencies containing the type of inconsistency (absence or divergences) and the involved entities.

4 Evaluation

In this section we present a case study to demonstrate the feasibility of our approach. We choose to analyze an academic MDS project for a matrix framework. The developers of this MDS project are in house and can be consulted for evaluating the results.

4.1 Matrix Framework

The Matrix Framework is an academic project for self-adaptive matrix ADTs (abstract data types). The efficiency of matrix operations depends on the representation of the matrix and the choice of algorithm for this operation. The

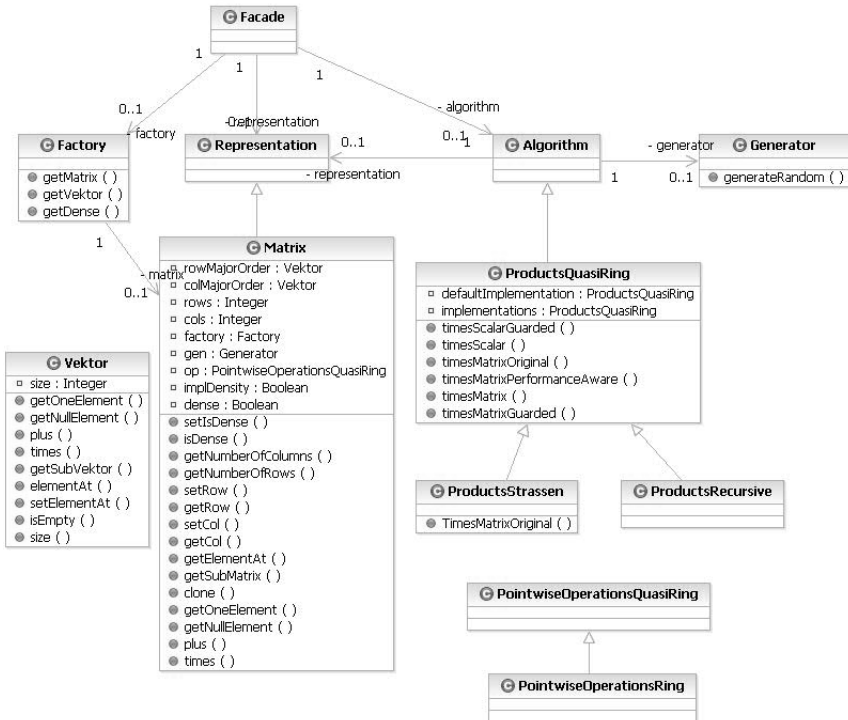


Fig. 2. UML Diagram of the Matrix Framework

matrix framework is designed such that the representations and the algorithms can be changed independently of each other. Automatically – using profiling – the most efficient choice for representation and algorithm is found depending on the actual problem size, density, and machine configuration.

The input to the consistency check is the matrix framework project, consisting of a UML design document, the Java implementation and a model transformation. In the following we introduce each of the three parts separately. The UML design document consists of a class diagram with 12 classes. It is written in UML Version 2 and serialized as XMI 2.0. It is depicted in figure 2. The Java implementation contains 18 classes. The code skeletons have been manually extended and new classes have been manually created. This allows us to see the mapping completion of our approach. The project contains a model-to-text transformation that transforms the UML class diagram into a Java implementation. The transformation consists of 11 rules written in the Xpand language of openArchitectureWare. Additionally there are several meta-model extensions written in the Xtend language of openArchitectureWare [6].

4.2 Execution

We have measured the runtime of the steps involved in checking for consistency. The measurement system is a Intel Pentium M 1.7 GHz with 1.5 GB RAM running Windows XP:

Process step	Runtime (in ms)
Model transformation with tracing	2684
Java Fact Extraction	6887
UML Fact Extraction	1482
Clustering	1654
Rule Checking	572
Total Runtime	13279

To evaluate the overhead for tracing we have made a runtime measurement with and without tracing: The model transformation without tracing takes 2483 ms, with tracing it takes 2684 ms, resulting in a tracing overhead of only 201 ms.

4.3 Results and Interpretation

In the following we discuss the results of our automated consistency check, in particular the results of the clustering algorithm and the detected inconsistencies.

As described earlier, the project contains more Java than UML classes as some classes have been manually created. For consistency checking, we need to assign these Java classes to a high-level entity of the UML design document. The clustering algorithm chooses this high-level entity based on hierarchy information and dependency information. The classes `VektorDense` and `VektorSparse` are not present in the UML class diagram. They are mapped to the high-level class `Vektor`. This makes sense, since `VektorDense` and `VektorSparse` are subclasses

of the class `Vektor`. The low-level Java classes `BooleanQuasiRing`, `DoubleRing`, `QuasiRing` and `Ring` are mapped to the high-level UML class `Factory`. This makes sense, since the `Factory` class heavily uses these representations.

The consistency check locates five inconsistencies in the case study. The first three inconsistencies are similar: They are divergence inconsistencies between `ProductsQuasiRing` and `Factory`, between `ProductsStrassen` and `Factory`, between `ProductsRecursive` and `Factory`. Since all the product operations produce a result and put it into a new matrix, a reference to the `Factory` class is actually required. The design does not reflect this and needs to be adapted.

We find another divergence inconsistency between `Factory` and `Generator`. The implementation of `Factory` has a reference to the `Generator`, however this reference is not present in the design documents. A closer look at the source code reveals that the reference is actually never used. It is thus safe and advisable for the sake of design clarity to remove the reference from the implementation.

The last discovered divergence inconsistency is between the `Matrix` and the `Generator`. A closer look at the `Generator` class on implementation level reveals that it contains functionality of two different purposes: (1) providing randomized input matrices for profiling and (2) providing the one element and the null element of a matrix. In the design documents the `Generator` has purpose (1), whereas in the implementation it has purpose (1) and (2). This may easily lead to misunderstandings. Since the implementations of the two purposes do not share any functionality or code, it is advisable to split up the `Generator` according to the two purposes, resulting in a cleaner and easier to understand design.

The five inconsistencies our consistency check discovered exist due to manual additions filling in the skeletons. All detected inconsistencies were inconsistencies indeed. They hint at potential design problems that, according to the developers of the matrix framework, need to be fixed.

5 Related Work

In this section we give an overview of existing techniques and approaches for *vertical* architectural software consistency checking. We briefly describe the approaches found in the literature and evaluate them w.r.t. our objective of fully automating the process. We acknowledged that there is orthogonal work on automated *horizontal* software consistency checking, e.g., [15,2], but exclude this from our discussion.

Most works follow a standard approach where an analyst first creates a high-level model based on information from documentation. Next the analyst uses a fact extractor to create a low-level model of the software based on the source code. The analyst manually maps low-level entities to their corresponding high-level entities, thus introducing new mapping edges. Relational algebra is used to specify inconsistency rules between the high-level model and the low-level model.

The most prominent example is the Software Reflexion Model [17]. It has been extended to support hierarchical models [13]. Another extension semi-automates the mapping creation, where a partial, manually created mapping is completed

by a clustering algorithm [4,3]. Postma et al. specialize in analyzing component based systems [24]. Egyed et al. analyze systems created with an architecture description language (ADL) [7,8,9]. The research of Paige et al. is targeted at finding a new definition for model refinement by using consistency rules written in OCL [21]. Our approach exploits the same correspondence between refining model transformation and consistency. However, the goals are different and there is no explicit consideration of the challenges of automation and the extraction of the mapping from model transformations. Nentwich et al. use an XML-based solution for checking consistency between arbitrary documents [20,19]. The approach of Muskens et al. nominates one of the two compared models as the gold standard, the so-called prevailing view. It deduces architectural rules from it and imposes these rules on the other model, the subordinate view [18]. Thus no external consistency definition is required. However, nominating subordinate and prevailing view requires manual work and cannot be automated. While all of these approaches tackle the problem of software consistency checking, none of them is automated completely as summarized in table below. The approaches provide only a partial solution of our problem.

Approach	Low-level Extraction	High-level Extraction	Mapping Extraction
[17]	×	-	-
[13]	×	-	-
[4,3]	×	-	semi-automated
[24]	×	×	-
[7,8,9]	×	×	-
[21]	only in theory	only in theory	only in theory
[20,19]	×	×	-
[18]	×	×	-
Present paper	×	×	×

Moreover, in our literature survey, we have not found any approach specifically designed for consistency checks of software developed by MDSD. Surely, the approaches for traditionally developed software can be applied for analyzing MDSD projects as well, but they do not use the advantageous possibilities for consistency checking provided by MDSD. These advantages include the ability to automate the high-level model extraction and the mapping extraction.

6 Conclusion

In this work we have developed the concept for a tool that automatically identifies architectural inconsistencies between low-level and high-level artifacts of an MDSD process. We were lead by two major realizations: (1) the major part of the information needed for automated consistency checking can be extracted from the artifacts of the MDSD project and (2) missing information can be completed using heuristic approaches.

We extract a low-level model from (Java) source code and a high-level model from the UML model using language-dependent fact extractors. We can extract a large part of the mapping from the model transformation using AOP-based tracing. The remaining unmapped entities from the source code have been manually created and we find a mapping for them using a heuristic. We collect the information in an analysis graph and subsequently use it to search for patterns of inconsistencies using relational algebra expressions.

We have demonstrated the practical use of our tool in a case-study. All detected inconsistencies have been acknowledged by the developer of the case study. The inconsistencies hint at actual design problems that need to be fixed.

The next step is to perform additional case studies and validating experiments on a larger scale of MDSD projects and assess the number of false positives and negatives found by our approach. This will show the accuracy of the clustering and whether the heuristic needs to be improved.

Thus far we have looked only at inconsistencies due to creation of new code. We plan to check for alternative types of inconsistencies, e.g. inconsistencies created by modification or deletion of generated classes. We need to research the robustness of our approach, especially the robustness of the mapping extraction in these situations.

References

1. Anquetil, N., Lethbridge, T.: Comparative study of clustering algorithms and abstract representations for software modularisation. In: IEE Proceedings - Software, vol. 150, pp. 185–201. Catholic Univ. of Brasilia, Brazil (2003)
2. Blanc, X., Mounier, I., Mougénot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE 2008, pp. 511–520. ACM, New York (2008)
3. Christl, A., Koschke, R., Storey, M.-A.D.: Equipping the reflexion method with automated clustering. In: WCRE. IEEE Press, Los Alamitos (2005)
4. Christl, A., Koschke, R., Storey, M.-A.D.: Automated clustering to support the reflexion method. *Information & Software Technology* 49(3), 255–274 (2007)
5. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3), 621–645 (2006)
6. Efttinge, S., Friese, P., Haase, A., Kadura, C., Kolb, B., Moroff, D., Thoms, K., Völter, M.: openarchitectureware user guide. Technical report, openArchitectureWare Community (2007)
7. Egyed, A.: Validating consistency between architecture and design descriptions, March 6 (2002)
8. Egyed, A., Medvidović, N.: A formal approach to heterogeneous software modeling. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 178–192. Springer, Heidelberg (2000)
9. Egyed, A., Medvidovic, N.: Consistent architectural refinement and evolution using the unified modeling language, March 6 (2001)
10. Garey, M.R., Johnson, D.S.: *Computers and Intractability*. W.H. Freeman, New York (1979)

11. Himsolt, M.: GraphEd: a graphical platform for the implementation of graph algorithms. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 182–193. Springer, Heidelberg (1995)
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
13. Koschke, R., Simon, D.: Hierarchical reflexion models. In: WCRE, pp. 36–45. IEEE Press, Los Alamitos (2003)
14. Lowe, W., Panas, T.: Rapid construction of software comprehension tools. In: International Journal of Software Engineering and Knowledge Engineering (2005)
15. Mens, T., Van Der Straeten, R., D’Hondt, M.: Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
16. Murphy, G.C., Notkin, D.: Reengineering with reflexion models: A case study. *Computer* 30(8), 29–36 (1997)
17. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 27(4), 364–380 (2001)
18. Muskens, J., Bril, R.J., Chaudron, M.R.V.: Generalizing consistency checking between software views. In: 5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005, pp. 169–180 (2005)
19. Nentwich, C., Emmerich, W., Finkelstein, A.: Static consistency checking for distributed specifications (2001)
20. Nentwich, C., Emmerich, W., Finkelstein, A.: Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology* 12(1), 28–63 (2003)
21. Paige, R.F., Kolovos, D.S., Polack, F.: Refinement via consistency checking in MDA. *Electr. Notes Theor. Comput. Sci* 137(2), 151–161 (2005)
22. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Communications of the ACM* (1972)
23. Perry, D., Wolf, A.: Foundations for the study of software architecture. *ACM SIG-SOFT Software Engineering Notes* 17(4), 40–52 (1992)
24. Postma, A.: A method for module architecture verification and its application on a large component-based system. *Information & Software Technology* 45(4), 171–194 (2003)
25. Stahl, T., Völter, M.: Model driven software development. Wiley, Chichester (2006)
26. Tran, J.B., Godfrey, M.W., Lee, E.H.S., Holt, R.C.: Architectural repair of open source software. In: IWPC 2000, p. 48 (2000)
27. Tzerpos, V., Holt, R.C.: Software botryology: Automatic clustering of software systems. In: DEXA Workshop, pp. 811–818 (1998)
28. Wiggerts, T.A.: Using clustering algorithms in legacy systems remodularization. In: Reverse Engineering, WCRE 1997, pp. 33–43 (1997)

Improved Feedback for Architectural Performance Prediction Using Software Cartography Visualizations

Klaus Krogmann¹, Christian M. Schweda², Sabine Buckl²,
Michael Kuperberg¹, Anne Martens¹, and Florian Matthes²

¹ Software Design and Quality Group
Universität Karlsruhe (TH), Germany

{krogmann, mkuper, martens}@ipd.uka.de

² Software Engineering for Business Information Systems

Technische Universität München, Germany

{schweda, buckl, matthes}@in.tum.de

Abstract. Software performance engineering provides techniques to analyze and predict the performance (e.g., response time or resource utilization) of software systems to avoid implementations with insufficient performance. These techniques operate on models of software, often at an architectural level, to enable early, design-time predictions for evaluating design alternatives. Current software performance engineering approaches allow the prediction of performance at design time, but often provide cryptic results (e.g., lengths of queues). These prediction results can be hardly mapped back to the software architecture by humans, making it hard to derive the right design decisions. In this paper, we integrate *software cartography* (a map technique) with software performance engineering to overcome the limited interpretability of raw performance prediction results. Our approach is based on model transformations and a general software visualization approach. It provides an intuitive mapping of prediction results to the software architecture which simplifies design decisions. We successfully evaluated our approach in a quasi experiment involving 41 participants by comparing the correctness of performance-improving design decisions and participants' time effort using our novel approach to an existing software performance visualization.

1 Introduction

Performance is a complex and cross-cutting property of every software system. If performance targets (e.g., maximum response times) are not met, a redesign or even a reimplementation of the system is needed, which leads to significant costs. Model-based software performance prediction approaches (cf. [1,31]) estimate the performance of software architectures¹ at design time, before fully implementing them. These approaches are also referred to as *Software Performance Engineering* (SPE, [26]), and several implementations of it exist. Making the correct design decisions using SPE requires proper understanding and interpretation of the prediction results.

¹ We will use the term *architecture* to capture static structure, behavior, and deployment of a software.

Nevertheless, many companies do not use SPE in software development [4]. Woodside et al. [31] highlight the *limited interpretability* in current SPE approaches: “Better methods and tools for interpreting the results and diagnosing performance problems are a future goal”. Mapping performance prediction results back to the analyzed software architecture is difficult because the underlying concept’s abstract entities (e.g., places in Petri Nets [15] or queue lengths in Layered Queuing Networks [8,23]) are not directly linked to a software architecture. Also, performance prediction results are insufficiently aggregated so that performance data is at a lower abstraction level than architectural elements (e.g. only resource demands of single steps of behavior specifications are available, while reallocation decisions require resource demands at the component level). Both aspects result in high time demands for the identification of performance problems like bottlenecks. For example, the SPE tool of [26] presents the resource demands of single steps in the behavior specification (“software execution graph”) as well as the overall demand of whole use cases or parts thereof (“scenarios”). Components are not reflected in the models. Thus, for decisions like reallocation, one needs to manually collect the results for all scenarios of the component.

The contribution of this paper is a novel integration of the approaches of *software performance engineering* (SPE) and *software cartography* to support design decisions by performance result visualizations at the level of software architectures. For SPE, our approach uses Palladio [2], a representative state-of-the-art model-based performance prediction approach. It is going to be integrated with software cartography [29], a scientific discipline which reuses cartographic techniques for visualizing large applications, i.e. software systems and their interconnections. Such a visualization technique is necessary as large applications form highly complex and interdependent systems, which are not easy to comprehend without graphical support. The visualizations are designed according to the viewpoints of individual stakeholders² to meet requirements of user convenience and usability.

The targeted benefits of the presented approach are a more intuitive result interpretation, a better usability, and thus the speedup of decision processes. These benefits are achieved by assisting the software architect through visualization of performance prediction results at an *architectural* level, which allows to map performance prediction results to the elements of an architectural software model (cf. Fig. 2 / 5). Also, multiple information layers are available to ease trade-off decisions and to help in identifying crosscutting design impacts. Our solution overcomes the limited interpretability of performance evaluation results as provided by Petri Nets, or Layered Queuing Networks. It transforms an architectural model, a prediction result model, and graph description models into a software architecture visualization which includes prediction results. The generality of our visualization approach allows different kinds of viewpoints.

We successfully evaluated the approach in a quasi experiment involving both experienced software performance analysts and less experienced computer science students. Our experiment shows an increased precision and effectiveness of design decisions processes, making it more likely to choose the right design options. The approach is applicable also for users with little experience in SPE.

² The terms *stakeholder* and *viewpoint* are used here in accordance to their definitions in [13].

In the remainder of this paper, Section 2 surveys related work, Section 3 describes the foundations of software cartography and visualizations while Section 4 introduces SPE. In Section 5, we present our novel integrated visualization approach, which is evaluated in Section 6 in a quasi experiment, before Section 7 concludes the paper.

2 Related Work

Model-based software performance prediction is surveyed by Balsamo et al. in [1]. In this area, the existing approaches either support (i) *result aggregation* or (ii) *links to the software architecture*, but no approach integrates both aspects, as described in the following. The commercial SPE-ED [26] tool by Smith et al. highlights critical actions in a behavioral model and highly utilized servers, and thus allows fast result interpretation for software architects. Still, there is no support in SPE-ED for high-level architectural constructs like composite components. In other commercial SPE tools like “Performance Optimizer” or “Capacity Manager” from Hyperformix [12], visualizations are mostly table-based or bar charts to illustrate impacts of changing deployment, but the visualizations are not connected to models of the software architecture. For Hyperformix products, only little information on visualization techniques is available, and the software itself is not freely available or affordable for universities to perform case studies or experiments. Of the approaches surveyed in [1], those that do link performance results back to architectural elements of the design model, like [30,14,5], still do not provide *aggregated* information on an architectural level, e.g. for components. The same is also true for similar newer approaches such as UML- Ψ [17].

Software model visualization in the field of software engineering is dominated by the unified modeling language (UML) [21], which provides the common basis for modeling single software systems. For performance modelling, the UML MARTE profile [20] has been suggested, which introduces performance-related tagged values for UML constructs. However, while the tagged values can be displayed in all UML diagram types, they do not provide a specialized visualization of the performance results (e.g. resource utilization visualization) to support fast insight and design decisions (e.g. reallocation of components to spread the load). Furthermore, the creation of different viewpoints according to the concerns of various stakeholders is not supported in UML (cf. [19]).

Visualization approaches of arbitrary models like *GMF* (Graphical Modeling Framework [6]) support the creation of graphical notations for arbitrary models (e.g., software or performance models), but put a special emphasis on creating graphical editors. In GMF, the user is granted a maximum amount of flexibility regarding the layout of visualizations – means for specifying layout rules are limited in the GMF approach. Also, GMF implies that the visualization of a specific concept uses a distinct unique type of visualization – to add other graphical elements for the visualization, GMF needs to create a *new* editor.

Other **software performance visualization approaches** [16,11,32,25,33] have already been proposed and they all support stakeholders in making design decisions to improve software performance. However, none of these approaches supports *architectural* design decisions by visualizations. Instead, they usually focus on parallel programs and often require executable implementations to monitor the software behaviour.

All of these approaches have in common that their visualization is not empirically evaluated. An additional limitation is that their outdated GUI techniques cannot be integrated into modern IDEs like Eclipse. General performance evaluation tools (e.g. profilers or performance monitors) are also implementation-centric and must execute the finished application.

3 Software Cartography

Conventional cartography (making geographical maps; see e.g. [10]) provides techniques well-suited for presenting complex information to a wide variety of people from different educational backgrounds. These techniques range from color-coding according to property values to the separation of the *base map* and layered additions.

Software cartography [29] re-uses cartographic techniques for visualizations of complex interconnected systems, called *software maps*. These maps are stakeholder-specific visualizations, especially designed as means for management support. In analogy to *urban planning*, a city and a software application landscape share a number of characteristics [18]:

- They form networked, open systems with autonomous and active constituents.
- They are constantly evolving and (mostly) have no designated end of lifetime.
- Many people are involved as stakeholders, with different educational backgrounds.
- Different stakeholders have different concerns regarding the system, such that a balance of interests has to be achieved.

Especially the people-centric characteristics motivate the idea of using cartographic techniques for visualizing systems. Existing stakeholder-specific graphical notations for certain aspects, e.g. the diagram types introduced in UML for the software development process, are not widely known outside the respective domain – neither business architects nor system administrators are likely to understand UML sufficiently. Map-like visualization, although having no such well-defined semantics, can be more easily understood by the various stakeholders.

One might argue that software maps do not provide a well-defined semantics, as the same type of symbol (e.g. a rectangle) might have different meanings in different visualizations (i.e. software maps). While this is true, the meaning of a specific symbol on one software map is clarified using another cartographic technique, the *legend*. A legend in *conventional* maps provides textual information on the meaning of the symbols used in the map. Such information is also contained in a *software map* legend, which also comprises information on the meaning of (relative) positioning, as different maps can employ different positioning rules to express certain underlying information.

Relative positioning rules are especially of interest in the context of the base map, leading to a distinction between different types of base maps – the so-called *software map types*. There are three distinct basic types, of which one – a cluster map – is shown in Figure 1 as an example. The cluster map uses the principle of *clustering* (i.e. nesting of symbols into other symbols) to visualize relationships, preferably hierarchical ones. On this map, a distance measure between the visualized concepts emerges from the assignment of the concepts to parenting clusters – concepts in the same cluster are

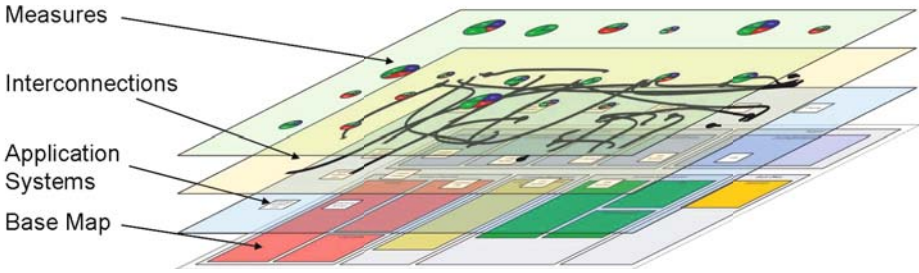


Fig. 1. Cluster map utilizing the layering principle

closer to each other than concepts in different clusters. Wittenburg [29] provides a more in-depth introduction to the software map types.

Regarding the involvement of various stakeholders, as for example in the context of performance predictions, the utilization of the so-called *layering principle* from cartography is especially beneficial. This concept allows the setup of a base map containing the logical constituents of the system under consideration, further utilizing relative positioning of symbols to represent certain types of associations as spatial relationships, which can be easily perceived. Additional information, only necessary for a certain stakeholder group, can be presented on an additional layer, which can be displayed or hidden on demand. Figure 1 illustrates this principle. In the context of software performance predictions, especially the layers showing *measures* and *interconnections* are of interest. The *measures* layer contains pie charts showing certain properties of the system under consideration on an aggregated, e.g. business-relevant, level. The actual interconnections, from which the properties of the system emerge, are contained in the *interconnections* layer.

In this setting, a stakeholder from a business group can identify a certain system based on the pie chart aggregation. Then the layer containing the interconnections can be overlaid to achieve a link to underlying technical information. By the utilization of the layering principle, it is ensured that the software system under consideration does not change its graphical position between the different viewpoints, which would not necessarily be the case, if stakeholder-specific visual notations were employed.

4 Software Performance Prediction and Visualization

It is significantly more expensive to deal with performance problems in an implemented software system than to *prevent* performance problems *before* they occur [26]. Thus, it is advantageous to *detect* performance deficiencies of a planned software system during the design phase. As actual performance measurements of the system are not possible during the design phase, the performance must be estimated (predicted) based on information available at that moment.

Performance prediction is also needed in other scenarios where measurements are impossible or cost too much effort. For example, to answer sizing questions on the server to be used in relation to an expected workload, performance prediction can help

to evaluate and to make design decisions (“is it more effective to buy server X or to replace components C and D with ones having higher performance, but remaining on an old server Y?”).

To answer such question on scalability, sizing etc., Software Performance Engineering (SPE [28]) is a *systematic* and well-studied approach for early estimation and prediction of software performance. As input, SPE approaches take a system’s *usage model* (i.e. workload and expected user behavior) and an *architectural model* (e.g. in UML) which includes a static part (components and connectors), a behavioral part (e.g., a strong abstraction of control and data flow), the resource environment (available server and networks), and component allocation (a mapping between component and resource environment). SPE is used to assess the feasibility of given performance requirements; achieving the optimal performance is usually not an objective because of costs.

Visualization in SPE. The performance of an executed software application depends on its usage of *resources* (e.g. CPU, HDD, network). *Resource contention* occurs when competing requests to a shared resource have to wait. Resource contention leads to a significant performance impact, which must be accounted for during performance prediction. After transforming the architectural models into analysis models such as Queuing Networks, SPE approaches analyze performance by employing simulation or analytical solutions. In the analysis, the resources maintain queues, with resource requests waiting if the resource is busy. Simulation or analytical solution of the modeled resources allows to estimate performance metrics such as utilization, waiting time in the queue, and response time. The length of such a queue allows to draw conclusions about resource utilization *over time*, and to identify which resources are bottlenecks. To “feed back” SPE results into their architecture, SPE users must be able to map SPE results to the architectural models – but in practice, there is a significant gap between (usually formal) analysis models of SPE (such as Queuing Networks and Petri Nets), and the architectural models.

Furthermore, the results returned by SPE are often aggregated: the response time for a coarse-grained application service does not allow to easily conclude which of the used components is consuming the largest amount of CPU power. Similarly, the textual output that reports the usage percentages of several CPUs still means that the SPE user must manually map these results to the (usually graphical) deployment model. Finally, to compare different design options, the respective SPE prediction results for them need to be visualized so that the SPE user can comfortably compare them in a unified way.

Visualization Requirements. Thus, results that need to be visualized are not only “end-user” performance metrics (e.g. service response time), but also “internal” metrics, such as the utilization of a resource. Visualization should allow to detect performance problems and, based on this, to make the right design decisions. To increase software architects’ effectiveness and correctness when designing with respect to performance, the visualization must be easy to understand. Apart from presenting single, isolated results, visualization of performance prediction results should consider *architecture relation*, *highlighting*, *correlation*, and *decomposition*:

- The visualization should offer the software architect the same components as in the design phase. Otherwise, the back-mapping of performance results gets ambiguous,

error-prone and as our studies in Section 6 show, it leads to less correct and more time-consuming design decisions. Consequently, results must be presented at the architectural level.

- To make it easier to match the performance prediction results and the architecture models, the aggregated results should be an overlay over the respective architecture models; an overlay can be textual (e.g., a tooltip showing the median utilization), or graphical (e.g., specific coloring of bottleneck resources).
- Visualizations that use multiple data dimensions (e.g., response times vs. more powerful resources) must be supported to allow trade-off analyses (e.g., cost of introducing cache vs. higher worst-case latency), and evaluation of cross-cutting concerns (e.g., enabling time-consuming security features).
- Multiple viewpoints (e.g., static architecture viewpoint vs. deployment viewpoint) of the same architectural prediction results should enable the stakeholder to delve into details. The software architect can then analyze an architecture with different focuses.

5 Integrated Approach

In our approach, we bring together visualizations from software cartography and performance predictions at the level of software architecture. This combination promises improved understanding of performance issues and the potential to easily optimize software design.

Palladio [2] is a model-based state-of-the-art SPE approach that features an integrated, Eclipse-based tool chain for modeling and evaluating software architectures. *Palladio* will be used in the quasi experiment (cf. [24, p. 4]) of our approach because we intend to study whether its visualization techniques must be enhanced, and which benefits the enhancements would provide. Internally, *Palladio* uses Queueing Networks for simulation-based performance prediction. Application models in *Palladio* are built on the *Palladio* Component Metamodel (PCM), following the paradigms of model-driven software development (MDSO).

In *Palladio*, raw performance prediction results are stored in a database. In the original visualization, data is basically accumulated to depict pie charts, bar charts/histograms, and line charts. However, the original visualization results are separated from the architecture model (the prediction results were simply *labeled* with service names, component names, and resource names). For example, the response time of a service is visualized as a general probability distribution of response times (cf. Fig. 2).

5.1 Chosen Visualizations in the New Approach

With the new, extended visualization, software architecture and performance prediction results are brought together in the visualization. For that, a new *decorator model* [9] annotates elements of a PCM model instance with prediction results. The decorator model contains prediction results for each component, server node, and network connection.

To realize the requirements listed at the end of section 4, we used four basic viewpoints to visualize performance results (these viewpoints are used in the evaluation

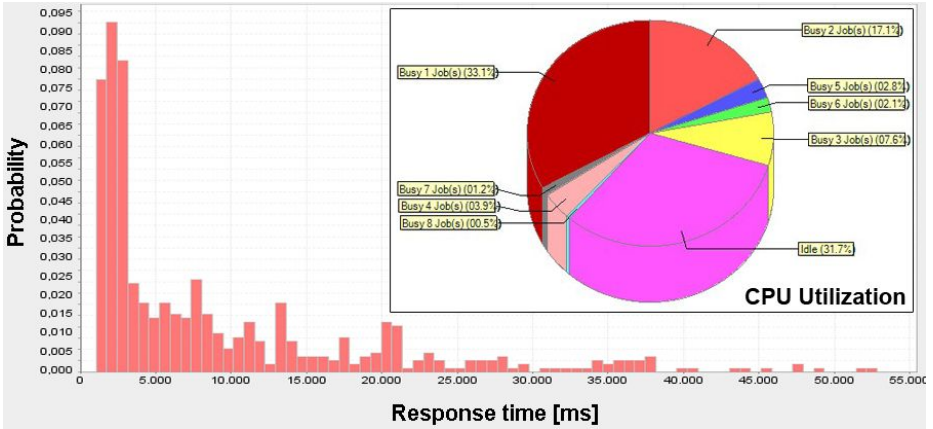


Fig. 2. Original result visualization in Palladio: Histogram of a general distribution function of service's response time and a pie chart showing the utilization of a resource (top right)

discussed in Section 6, where the actual visualizations are also shown; cf. Fig. 5) in our approach.

In the **black-box resource environment viewpoint**, server nodes and network connections are visualized together with their *utilization* (i.e. percentage of busy and idle times). In the visualization, strong network connections are thicker than slower ones. For the utilization of a server node, we use the weighted average of the utilization of its inner resources of one type (CPU, HDD, ...), weighting with their processing rate. Thus, we obtain a measure relative to the overall processing rate of all resources of this type. For passive resources such as thread pools, the utilization is analogously determined based on their capacity. If the server contains resources of different types (CPUs, HDDs, and thread pools), the utilization of the resource type with the highest average utilization is used, to allow fast bottleneck recognition.

In the **white-box resource environment viewpoint**, the first viewpoint is enhanced by visualizing server nodes and nested hardware resources (such as CPUs, hard disks, network connections, or thread pools) together with their respective utilization. Performance metrics here also include the *utilization per resource* (e.g. CPU or HDD) of a server node. In this viewpoint, also the average *wait time* (time for which requests are waiting in a queue; relative to the wait time of all resources in the model) and the average *demand absolute time* per resource can be visualized. The distinction between server nodes and resources allows users to delve into details. If a server has multiple CPUs, we can detect that, for example, only one CPU is under heavy load, while others are idle. For scenarios in which scheduling can be influenced or where different types of CPUs are used, one can then try to shift computations to another CPU.

In the **white-box allocation viewpoint**, the deployment of software components into the execution environment is included into the visualization. In addition to server nodes and network connections between them, also all components for each server, the connection of components, their individual *resource demands* as well as the *absolute response time* are visualized. The resource demand is shown relative to the summed-up

resource demands of all components on this server node (for composite components, also the internal resource demand is accumulated).

In the **software component viewpoint**, only software components, interfaces, and connectors between components are displayed. For each component the absolute response time is displayed.

Performance prediction result data is strongly aggregated with the intention to support understandability and to avoid overwhelming users with information. If the aggregated data is not sufficient, software architects and performance analysts can still delve into details using the original Palladio visualization, which provides access to specific results like the response time of individual services of components.

To ease the interpretation and speed of recognition, coloring indicates potential bottlenecks or critical architectural elements. *Components* with relative performance metrics (e.g., utilization) with more than 75% are colored red; while for *resources*, starting from 75% orange and starting from 85% red is used. Beyond coloring, also multiple data layers can be visualized for the same topology (same server nodes, network connections, and component allocation). In our approach, we use the data layers to visualize different usage scenarios to allow the estimation of impact for different usage scenarios (e.g., resource utilization with few vs. many users). Compared to the visualization requirements from Section 4, our visualization approach provides a) a relation between architecture and performance prediction results, b) highlights critical architectural elements, c) eases correlation analysis through multiple layers, and d) enables decomposition by multiple viewpoints at different levels of detail. We will detail on visualizations in the evaluation discussed in Section 6.

5.2 Applying the Software Cartography Approach to Visualizing Performance Information

Software maps, as introduced in Section 3, are consistent visualizations of information, i.e. a viewpoint on the information utilizing cartographic means. As creating these

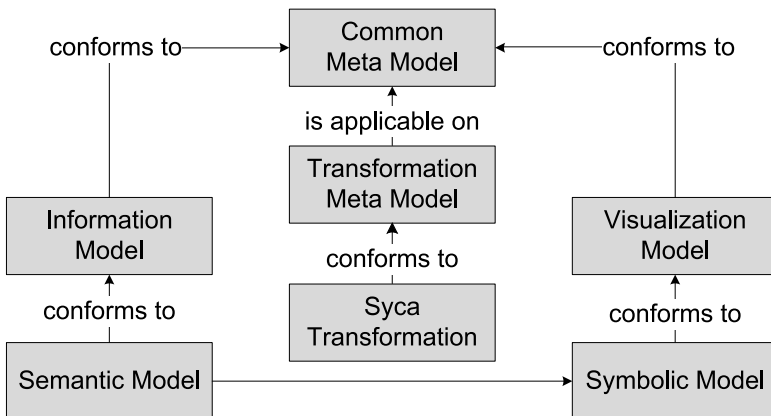


Fig. 3. Basic principle behind the Syca transformation approach to visualization generation

visualizations manually is both a time-consuming and error-prone process, an approach for generating software maps from input information is presented in [7]. This approach is based on object-oriented models of both the information and the visualization as well as a model-to-model transformation, the so-called *Syca transformation*. The core concepts of the approach are sketched in Figure 3 and lay the basis for the integration explained subsequently. Prior to details on the integration, the core models of the visualization generation approach are introduced:

Information model. This (meta) model sets up the language for describing the modeling subject, i.e. it introduces the core concepts, which are used to create a model of the subject's reality. In the context of software performance prediction, the information model defines concepts such as *components*, *connectors*, and attributes such as *average utilization*. The elements contained in an object-oriented information model are *classes*, *attributes*, and *relationships*.

Semantic model. The semantic model contains instance data modeled according to the respective information model, i.e. it contains *information objects*. In the context of software performance prediction, these objects are instances of components and interconnections, having assigned values for the attributes.

Visualization model. This (meta) model defines graphical concepts, either visual ones – so called *map symbols* – or *visualization rules*, which describe graphical relationships between the symbols. These rules can be used to specify relative positionings, size, or the overall appearance of symbols at the level of graphical concepts without having to supply all details of positioning or layout.

Symbolic model. This model contains visualization concept instances, i.e. map symbols and visualization rules, modeled according to the respective visualization model. By assigning visualization rules to the symbols, the relative positioning is determined, although no absolute positions are specified.

All aforementioned models are described using the Meta Object Facility (MOF) [22]. This language therefore provides the common meta model for both information model and visualization model.

The common meta model further lays the basis for expressing a mapping from information model to visualization model concepts in terms of a model-to-model transformation. This *Syca transformation* can be used to specify how a concept from the information model, e.g. a component, should be visualized, e.g. as a rectangle. Executing the thus specified transformation, instances from the semantic model can be mapped to instances in a symbolic model, which together describe a visualization. The transformed symbolic model is subsequently handed over to a layouting component, which computes the actual positions and sizes of the symbols in accordance with the respective rules. For details on the layouting mechanism, see e.g. [7].

The above approach can handle arbitrary object-oriented information models and corresponding instance data. Therefore, it is possible to apply it on the performance prediction data (Result Decorator) as computed by the Palladio approach, which is also represented in a decorated object-oriented model. The basic make-up of a respective integration is sketched in Figure 4. Selected technical details of the integration are described below.

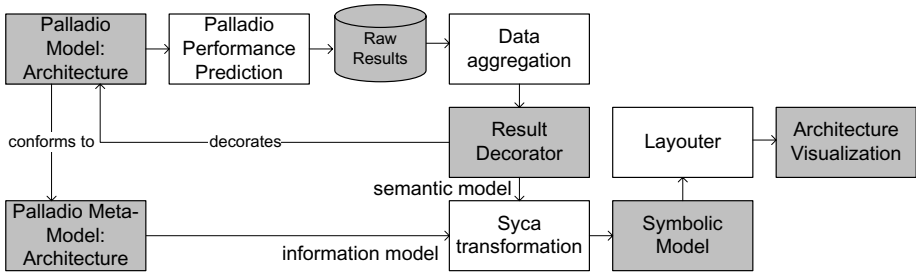


Fig. 4. Processing of models and prediction results

The model-transformation based approach for visualization generation has been implemented in a prototypic Eclipse based tool, the *SoCaTool* [3]. It allows the technical integration to the Palladio Bench, which is also realized based on the Eclipse platform. In this integration, the decorated results from the performance prediction are read as semantic models into the *SoCaTool*, which subsequently hands the data over to the Syca transformation component. From this, a generated symbolic model is passed to the layouter, which finally computes the absolute positioning of the symbols and thus creates a visualization. Central cornerstone of the technical integration is hence the realization of an appropriate Syca transformation. This transformation is built on the data contained in the decorated resource model.

For this paper, the existing Palladio and *SoCaTool* have been extended by decorator support, the Result Decorator itself, and the new Syca transformation. Screenshots of the realized visualization approach are available online³.

6 Evaluation

To evaluate the chosen visualization of performance results, we conducted a quasi experiment involving students and faculty members, all from the field of computer science. Only about half of the faculty members was familiar with Palladio.

In the quasi experiment, we investigated four research questions:

- Q1: Can participants make correct design decisions based on the visualization? To do so, the participants need to correctly identify performance bottlenecks and be able to solve performance issues (changing the architecture in the right way).
- Q2: How long does it take the participants to evaluate a scenario?
- Q3: How does the participants' experience in software performance influence the above metrics?
- Q4: Do users of the new visualization perform better than users of the original Palladio visualization?

To evaluate the quality of the improved visualization with respect to these questions, we conducted the experiment in two phases. In the first phase, we studied all four

³ <http://sdqweb.ipd.uka.de/wiki/SoCa-Palladio>

questions above on a mixed group of 21 participants, both faculty members and master students.

The participants filled out a questionnaire with overall six different scenarios for which they needed to choose one or more valid design decisions for optimizing a given scenario. The scenario itself was roughly described in text, whereas the performance results were present in the visualizations only. 14 of the participants used the original Palladio visualization (*P old*), the remaining 7 participants used the new visualization (*C new*). All participants needed to answer the same questionnaire, but were provided with the different visualizations. Participants using the new visualization *C new* were provided with result visualization as shown in Figure 5 and described in Section 6.1. Participants using the original Palladio visualization *P old* were provided pie charts for resource utilization, mean values and cumulated density functions for response time, and/or behavior specifications of component internals for resource demands. Each visualization diagram provided a short legend.

The questionnaires provided multiple design options for each scenario. There were correct and required design options (that would resolve the performance problems), wrong design options (that would actually worsen the performance) and optional design options (that might slightly improve performance or save cost). For each participant and each scenario, we evaluated whether all correct and required design options and no wrong design options were checked. In this case, the decision of the participant for this scenario was considered correct. The optional design options had no influence on this assessment. Thus, although we provided a multiple choice questionnaire, sheer guessing would have led to very few correct decisions.

We asked the participants to self-assess their experience in software performance. Based on the self-assessment, we can distinguish our results for participants with “low” or “high” experience.

For the comparison (Q4), we performed statistical hypothesis testing on Q1 and Q2. Our hypothesis were H_{1_1} : “Participants using *C new* make in average more correct design decisions over the evaluated scenarios than participants using *P old*” and H_{2_1} : “Participants using *P old* need longer for the questionnaire in average than participants using *C new*”. We decide whether to reject the opposite null hypotheses H_{1_0} and H_{2_0} based on Welch’s t-test [27] and a significance level $\alpha = 0.05$.

In the second phase, we studied how well participants totally untrained in both software development and software performance can handle the new visualization for further insight into Q3. We asked another 20 undergraduate students in their second year to interpret the new visualization and also fill out the questionnaire described above.

We do not claim to have conducted a controlled experiment, in which we have controlled all influencing factors (such as common knowledge of the participants). Rather, our study allows an initial evaluation of the improved, new visualization *C new*.

6.1 Scenarios: Performance Design Decisions

To illustrate the possible visualizations, we created six different performance scenarios in which changes in the software architecture design are required. All scenarios are minimalistic and basic. More complex scenarios would be a combination of the visualization of basic scenarios.

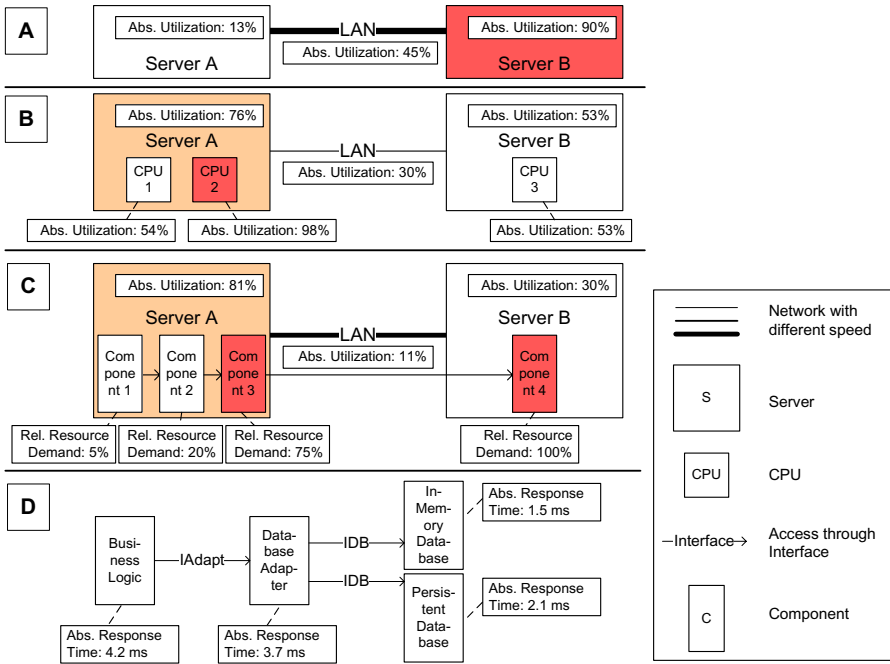


Fig. 5. New visualizations

1) *Server bottleneck.* In the first scenario (cf. Fig. 5 A), two servers are connected by a network connection. One of the servers is indicating a very high utilization ($>=90\%$), while the other has a normal utilization. In this case and at this level of information, there are two basic alternatives for increasing the performance of the system: a) use stronger server hardware for the highly utilized server, or b) consider changing component allocations (see scenario 4). For this scenario it is crucial to allow fast bottleneck recognition.

2) *Network bottleneck.* This scenario is comparable to the previous one (same topology; similar to Fig. 5 A), beside the fact that another resource – the network connection – is in an overload situation. Again, here two design alternatives are imaginable: a) increase the bandwidth of the network connection, or b) change the component allocations such that network traffic is reduced. Scenario 4 presents possible component relocations.

3) *Replicated resources.* If a single server has multiple CPUs or CPU cores, but just one or few of them are utilized, a lack of parallelism is indicated (cf. Fig. 5 B). In this case, a) other versions of utilized components with increased optimizations for parallel execution can be used, or b) low utilized server nodes can be down-sized to avoid wasting computational power, or c) further components can be allocated to the same server (and possibly bound to single CPUs) to improve the utilization of unused CPUs / cores. If the performance requirements are not fulfilled, also faster CPUs / cores can be used to avoid a bottleneck.

4) *Component's resource demand.* If server nodes are in an overload situation (cf. Fig. 5 C), it is desirable to also examine the component's allocation to server nodes for gaining insight on the causes of load. Therefore, each component's resource demand must be known. If the resource demand is normalized per server node, one can easily identify large demands which are likely to cause the overload: a) If at the same time other server nodes have only little load, it is a possible alternative to move the component with high resource demand to the other server nodes, b) if both, a server node and also a connected network connection have a high load, one needs to rethink the complete allocation and should also check whether for example the network connection could be improved to support solution a), c) for the server node with high utilization, also faster CPUs can be used.

5) *Compatible components.* In scenarios where two or more components offer the same interface, they become exchangeable (cf. Fig. 5 D). This is the case for databases with standardized interfaces, for example. By comparing the response times of exchangeable components, the faster implementation should be easily selectable. Such component selection decisions in general cannot be met, as component response time is sensible for the usage profile. If for example, a database is optimized for read requests, but fed with mostly writes, another database optimized for that purpose is much faster. Thus the decision cannot be met in advance, instead performance results (here: response time) must be visualized for a specific load situation.

6) *Multiple usage profiles.* As indicated in the previous case, components are subjected to different usage profiles. Sometimes, also for a deployed component, the usage profile still changes: Peak load situations, batch runs, or increasing users over time need to be handled. To estimate the scalability and sensitivity of a software architecture for changes in the usage profile, it is preferable to directly compare their impact on an architecture level. If the architecture does not satisfy the requirements for different anticipated load situations, the above scenarios 1-5 can be applied. Visualizations can apply multiple data layers which can be interactively hidden / unhidden. In the experiment, the participants compared two different usage profiles, each of which was visualized similarly to figure 5 C.

6.2 Phase 1 Results

All participants completed the entire questionnaire with the above scenarios in an average of 26.43 minutes (new visualization *C new*) or 40.17 minutes (original Palladio visualization *P old*).

Table 1 compares the results for question 1 (correctness) and question 2 (duration) for the different levels of experience (question 3) and for the two visualizations (question 4). Our first observation is that participants using the new visualization *C new* were able to make better design decisions (92.8% correct vs. 72.7% correct for *P old*, $p\text{-value} = 0.007 < \alpha$, thus we reject H_{10}). Both low and highly experienced participants performed similarly for their visualisation. It is remarkable that high experienced participants using *P old* performed worse than low experienced. On the questionnaires of some of these participants, we found manual calculations, which possibly lead to wrong results. In column "total # of decisions", we show the total number of decisions

Table 1. Results for cartography (*C new*) and the original Palladio visualization (*P old*)

Experience	% of correct decisions		total number of decisions		duration in min		standard deviation of duration	
	<i>C new</i>	<i>P old</i>	<i>C new</i>	<i>P old</i>	<i>C new</i>	<i>P old</i>	<i>C new</i>	<i>P old</i>
low	91.7%	75.0%	24	36	28.25	37.00	15.44	4.24
high	94.5%	70.8%	18	48	24.00	41.75	13.08	12.45
total	92.8%	72.7%	42	84	26.43	40.17	13.46	10.13

made by all participants in the corresponding group: Each of the 8 participants evaluated up to 6 scenarios.

For the duration, we again notice that participants using *C new* were able to finish the decision process (including interpretation and the decision itself) more quickly ($p\text{-value} = 0.03 < \alpha$, thus we reject H_{20}). Here, for low experienced participants using *C new*, the standard deviation of the duration is three times higher than for *P old*. Interestingly, more experienced participants needed longer than low experienced participants for interpreting *P old*. Possibly, they tried to delve deeper in the information available for *P old*, for example behavior specification (which did not lead to better decisions).

Overall, *C new* performed significantly better with respect to both correctness and duration. We can accept both hypotheses H_{11} and H_{21} stated above. The experience of the participants (with at least basic knowledge in software engineering) does not seem to have a strong impact on the results.

Table 2. Percentage of correct decisions for each scenario

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Scenario 6
<i>C new</i>	100.0%	100.0%	100.0%	85.7%	85.7%	85.7%
<i>P old</i>	92.9%	92.9%	78.6%	35.7%	64.3%	71.4%
Average	95.2%	95.2%	85.7%	52.4%	71.4%	76.2%

Table 2 shows the percentage of correct decisions for each scenario and the two visualizations. For the simpler scenarios 1 and 2, most subjects decided for the right design options. For the more complex scenarios 3, 5, and 6, the ratio of correct decisions is slightly lower for *C new*, whereas for *P old*, the ratio dropped about one quarter. For scenario 4, the differences between *C new* and *P old* are most pronounced, as the *P old* visualization seems to have been hard to interpret for most subjects: In *P old*, resource environment and software architecture are split into two distinct views, which might have been problematic to link. Here, most *P old* subjects could not correctly assess the consequences of their proposed re-allocation decision.

6.3 Phase 2 Results

In phase 2, the 20 participants untrained on both software development and software performance were able to make correct design decisions in 68.3% of a total number of 99 completed scenarios (some participants only answered 3 or 4 scenarios). The

duration was on average approx. 40 minutes. Thus, the new visualization *C_{new}* seems to have helped the participants, even though totally inexperienced, in gaining some insight in the performance problems of the described scenarios. However, the lower percentage of correct decisions shows that the visualization is probably not sufficient to allow these totally inexperienced participants to make correct design decisions in general and to reliably solve performance problems.

7 Conclusions

In this paper, we have presented an integrated approach for the visualization of software performance at an architectural level. With visualization means from software cartography, our approach depicts software performance prediction results in an software architecture visualization. As our quasi experiment involving 41 participants shows, the approach enables users with at least a basic level of experience in software development to correctly derive design decisions from visualizations of performance prediction. Compared to a reference group applying the original visualization of the Palladio approach, participants performed more correctly and faster, independent of in-depth experience with software performance prediction. Thus, our approach enables users to map performance prediction results back to a software architecture model and lets them decide for correct design alternatives.

Our approach meets demands of software cartography visualizations (cf. Section 4) as it a) relates architecture and performance prediction results, b) highlights critical architectural elements, c) eases correlation analysis through multiple layers, and d) enables decomposition by multiple viewpoints at different levels of detail. Nevertheless, our approach is not useful for completely untrained users, which have no experience with software development and software performance engineering. Currently, our visualizations are limited to four different views, although including support for an arbitrary number of data layers.

For our future work, we plan to further push the automation of the integration of the Palladio performance prediction approach with the SoCaTool software cartography. Besides adding further views to the approach, we also would like to integrate other quality attributes like reliability and maintainability. As our approach is not limited to Palladio, it would be beneficial to use the proposed visualizations with other SPE approaches.

References

1. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. on SE* 30(5), 295–310 (2004)
2. Becker, S., Koziolok, H., Reussner, R.: The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 3–22 (2009)
3. Buckl, S., Ernst, A.M., Lankes, J., Matthes, F., Schweda, C., Wittenburg, A.: Generating visualizations of enterprise architectures using model transformation (extended version). *Enterprise Modelling and Information Systems Architectures – An International Journal* 2(2) (2007)

4. Compuware. Applied performance management survey (October 2006), http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/APM_Survey_Report.pdf (last retrieved 2009-02-10)
5. de Miguel, M., Lambolais, T., Hannouz, M., Betgé-Bretetz, S., Piekarec, S.: UML extensions for the specification and evaluation of latency constraints in architectural models. In: Workshop on Software and Performance, pp. 83–88 (2000)
6. Eclipse Foundation. Graphical modeling framework homepage
7. Ernst, A.M., Lankes, J., Schweda, C.M., Wittenburg, A.: Using model transformation for generating visualizations from repository contents. Technical report, Technische Universität München, Munich (2006)
8. Franks, G., Hubbard, A., Majumdar, S., Neilson, J., Petriu, D., Rolia, J., Woodside, M.: A toolset for performance engineering and software design of client-server systems. *Perform. Eval.* 24(1-2), 117–136 (1995)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
10. Hake, G., Grünreich, D., Meng, L.: *Kartographie*. Walter de Gruyter, Berlin (2002)
11. Heath, M.T., Etheridge, J.A.: Visualizing the performance of parallel programs. *IEEE Software* 8(5), 29–39 (1991)
12. HyPerformix Inc. Hyperformix homepage (2007), <http://www.hyperformix.com> (last retrieved 2009-01-22)
13. IEEE. IEEE Std 1471-2000 for Recommended Practice for Architectural Description of Software-Intensive Systems (2000)
14. Kähkipuro, P.: UML-based performance modeling framework for component-based distributed systems. In: Dumke, R.R., Rautenstrauch, C., Schmietendorf, A., Scholz, A. (eds.) *WOSP 2000 and GWPESD 2000*. LNCS, vol. 2047, pp. 167–184. Springer, Heidelberg (2001)
15. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. on SE* 32(7), 486–502 (2006)
16. Lehr, T., Segall, Z., Vrsalovic, D.F., Caplan, E., Chung, A.L., Fineman, C.E.: Visualizing performance debugging. *Computer* 22(10), 38–51 (1989)
17. Marzolla, M.: *Simulation-Based Performance Modeling of UML Software Architectures*. PhD Thesis TD-2004-1, Università Ca' Foscari di Venezia, Mestre, Italy (February 2004)
18. Matthes, F.: *Softwarekartographie*. Informatik Spektrum 31(6) (2008)
19. Nassar, M.: VUML: a viewpoint oriented UML extension. In: 18th IEEE International Conference on Automated Software Engineering, pp. 373–376 (October 2003)
20. Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06) (2006)
21. Object Management Group (OMG). Unified Modeling Language: Superstructure Specification: Version 2.1.2, Revised Final Adopted Specification (formal/2007-11-02) (2007)
22. OMG. Meta Object Facility (MOF) Core Specification, version 2.0 (formal/06-01-01) (2006)
23. Petriu, D.C., Shen, H.: Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) *TOOLS 2002*. LNCS, vol. 2324, pp. 159–177. Springer, Heidelberg (2002)
24. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14(2), 131–164 (2009)
25. Sarukkai, S.R., Kimelman, D., Rudolph, L.: A methodology for visualizing performance of loosely synchronous programs. In: Scalable High Performance Computing Conference, 1992. SHPCC 1992. Proceedings, pp. 424–432. IEEE, Los Alamitos (1992)
26. Smith, C.U., Williams, L.G.: *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, Reading (2002)

27. Welch, B.L.: The generalization of student's problem when several different population variances are involved. *Biometrika* 34, 28–35 (1947)
28. Williams, L.G., Smith, C.U.: Making the Business Case for Software Performance Engineering. In: *Proceedings of the 29th International Computer Measurement Group Conference*, Dallas, Texas, USA, December 7-12, 2003, pp. 349–358. Computer Measurement Group (2003)
29. Wittenburg, A.: *Softwarekartographie: Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften*. PhD thesis, Technische Universität München (2007)
30. Woodside, C.M., Hrischuk, C.E., Selic, B., Bayarov, S.: Automated performance modeling of software generated by a design environment. *Perform. Eval.* 45(2-3), 107–123 (2001)
31. Woodside, M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. In: *Proceedings of ICSE 2007, Future of SE*, pp. 171–187. IEEE Computer Society Press, Washington (2007)
32. Yan, J., Sarukkai, S., Mehra, P.: Performance measurement, visualization and modeling of parallel and distributed programs using the aims toolkit. *Softw. Pract. Exper.* 25(4), 429–461 (1995)
33. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. *Int. J. High Perf. Comp. Appl.* 13(3), 277–288 (1999)

Predicting Performance Properties for Open Systems with KAMI

Carlo Ghezzi and Giordano Tamburrelli

Politecnico di Milano
Dipartimento di Elettronica e Informazione, Deep-SE Group
Via Golgi 40 – 20133 Milano, Italy
{ghezzi,tamburrelli}@elet.polimi.it

Abstract. Modern software systems are built to operate in an open world setting. By this we mean software that is conceived as a dynamically adaptable and evolvable aggregate of components that may change at run time to respond to continuous changes in the external world. Moreover, the software designer may have different degrees of ownership, control, and visibility of the different parts that compose an application. In this scenario, design-time assumptions may be based on knowledge that may have different degrees of accuracy for the different parts of the application and of the external world that interacts with the system. Furthermore, even if initially accurate, they may later change after the system is deployed and running. In this paper we investigate how these characteristics influence the way engineers can deal with performance attributes, such as response time. Following a model-driven approach, we discuss how to use at design time performance models based on Queuing Networks to drive architectural reasoning. We also discuss the possible use of keeping models alive at run time. This enables automatic re-estimation of model parameters to reflect the real behavior of the running system, re-execution of the model, and detection of possible failure, which may trigger a reaction that generates suitable architectural changes. We illustrate our contribution through a running example and numerical simulations that show the effectiveness of the proposed approach.

1 Introduction

In the past, software systems were mostly developed from scratch. The development was managed by a single coordinating authority, which was responsible for the overall quality of the resulting application. This authority had full ownership of the application: every single piece was under its control and full visibility of the process and product was possible. Component-based software development represented a major departure from this model. According to this paradigm, parts of an application are developed, tested, and packaged by independent organizations. The roles of the *components developer* and the *components integrator* correspond to different degrees of ownership and visibility of the various parts

that compose a componentized application. Today, however, service-oriented architectures (SOAs) [26,27] are pushing this situation to an extreme case. SOAs support the construction of systems whose parts may be discovered and changed dynamically. Furthermore, services are components deployed and run by *service providers*, while *service integrators* create new added-value services by simply invoking existing external services. The lack of a global coordinating authority, however, implies that the designers of an application that integrates external services must take into account the chance that its components may evolve independently and unpredictably.

We often refer to this situation as *open-world software* [3]. The world in which modern software applications are embedded is intrinsically open. It evolves continuously in terms of requirements to fulfill, properties and behaviors of the domain (e.g., usage profiles), components that can be aggregated to form the application, and bindings among these components. This situation is typical of (business) services offered and invoked through the Internet, as in the case of Web services. Pervasive computing applications are another typical case.

We investigate here how the open-world assumption affects the way applications are designed and constructed. Software *modeling* has long been advocated as a crucial activity in designing complex software systems, which may affect their overall quality. Modeling produces abstractions that represent particular aspects of interest of a system and frees engineers from distracting details. By reasoning on models, software architects can focus on specific system properties and check, even automatically, if the system under design meets the desired requirements. They can evaluate the effects of competing design alternatives on requirements satisfaction. Architectural analysis can thus support anticipation of flaws that could otherwise become manifest in later phases of the development process, leading to unscheduled and costly activities of re-engineering and corrective maintenance. Finally, according to a model-driven approach [24,25], all steps of software development can be described as model-to-model transformations. Some transformations take a model as a source and transforms it into another model as a target, which is more suitable for certain kinds of analyses. As an example, an (annotated) UML Activity Diagram may be transformed into a Markovian model to perform reliability analysis (see [16]). Model-to-model correctness-preserving transformations may also provide generation of the target system, which is thus *generated* rather than *implemented*.

The main goal of our research is to support model-driven development of open-world applications. In particular, we focus on supporting software engineers to evaluate the impact of their design decisions on non-functional quality attributes of a composite application. By reasoning on architectural models, it is possible to derive applications whose quality attributes can be analyzed and predicted at design time. For example, it is possible to predict certain performance characteristics (such as response time) of the architecture of an application that integrates a number of components, based on assumptions on the behavior of the components and on the structure of the integrating workflow. One may change

either the structure of the workflow or the expected behaviors of the components and get different performance figures for the integrated application.

Open-world assumptions introduce some peculiar features in modeling. Consider an application that is designed by composing a number of components, orchestrated via some workflow. As an example, in the case of Web services, the application may consist of a BPEL process that invokes a number of external services. Some of the components (or services) are deployed and run by the owner of the integrated application. Others instead are external components, deployed and run by independent providers. Obviously, the precision and detail of the model can be quite different for the two kinds of components. For instance, in the latter case, components can be viewed as black boxes: all one can say about them is what the component's provider declares—what is published in registries, in the case of Web services. In the former case, instead, a model is a white box, and detailed information is available about its deployment platform.

Our on-going research is focusing on KAMI—a methodology and its support tools—which aims at providing modeling support to open-world software over its entire lifetime, from inception to operation. In a previous paper [15], we investigated the use of models to support design-time reasoning on reliability-related properties, using Markovian models. This paper discusses another piece of the KAMI puzzle, which focuses on the use of *Queueing Networks (QNs)* to support analysis of performance attributes. We discuss how QNs can be used to model the different kinds of components. Furthermore, we discuss the interplay between design-time and run-time analysis, as we did in [15] for reliability attributes. In an open-world setting, the parameters of the models of the various components are likely to change at run time. They may change because the predicted values used at design time turn out to be inaccurate, or because of changes in the components performed by their respective owners. Operational profiles of the expected use of the application are also likely to change over time. All such changes can be detected by a monitoring component and detected changes may generate automatic updates of the values of the parameters. By keeping the updated model alive at run time, it is possible to check whether requirements are still met. In case of a violation, automatic, self-healing reactions may be triggered by the model.

In the sequel, Section 2 describes our general framework for run-time model evolution. Section 3 describes a systematic approach to modeling open systems with QNs at run time. Section 4 describes a concrete running example with numerical simulations that show the effectiveness of the approach. Section 5 discusses related works. Section 6 concludes the paper describing the current limitations of our approach and future work.

2 Model Evolution with KAMI

The KAMI methodology starts at design time, when software architects build models of their system. For example, they may want to assess the performance of a particular architecture or they may want to compare two architectural alternatives for a component based system. To analyze and predict the performance

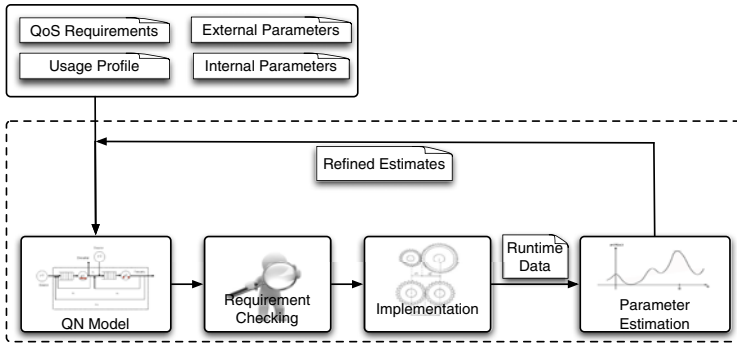


Fig. 1. Methodology scheme

of these competing solutions, they may build a QN model. Unfortunately, models for non-functional properties are characterized by and depend on numerical parameters. In our example, QNs require several parameters: (1) *Customer Interarrival Time Distribution (CITD)*, (2) *Service Time Distribution (STD)*, and (3) *Routing Probabilities (RP)*, which are usually unknown at design time. Consequently, model parameters depend on estimates provided by domain experts or extracted by similar versions of the system under design.

Software architects reason on their models iteratively, refining them to meet the desired requirements. Once a complete and satisfactory model is conceived and validated with respect to requirements, an implementation based on the model structure is built or generated. However, there is no guarantee that model parameters used at design time are correct or still hold in the target environment in which the system will be deployed. If they do not, the software may not exhibit the predicted performance, leading to possible unsatisfactory behaviors or failures.

KAMI addresses exactly this problem, which is crucial in the open-world case. KAMI is fed with run-time data provided by a monitor, which are passed to *estimators* that can produce new estimates of the numerical parameters of the non-functional models, by updating the initial estimate with the a-posteriori knowledge provided by run-time data. These continuous new estimates provide increasingly more accurate values and enable continuous automatic check of the desired requirements while the system is running. If the running system behaves differently from the assumptions made at design time and violates a requirement then a recovery action is triggered, aimed at managing the anomalous situation. Ideally, KAMI could trigger self-repairing strategies, according to an autonomic computing approach [17], which may synthesize an implementation that meets the requirements.

Figure 2 illustrates our ongoing implementation of KAMI. KAMI is a plugin-based software composed of: (1) *Model Plugins*, (2) *System Models*, (3) *Input Plugins*. *System Models* are text files describing the models on which KAMI operates. These files contain: (1) model descriptions, (2) the requirements the

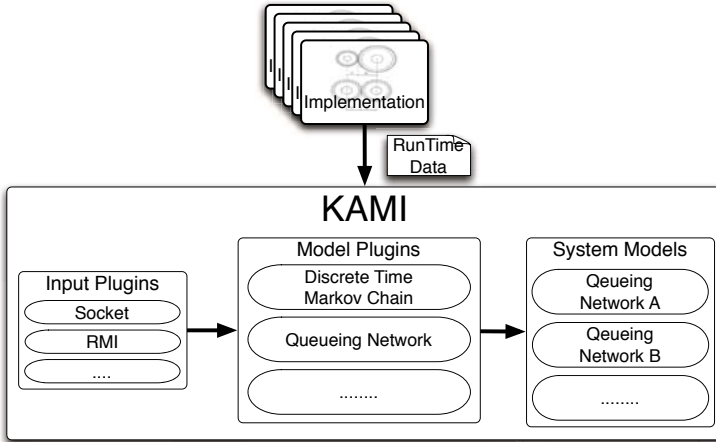


Fig. 2. Kami Architecture

user is interested in, and (3) a set of exceptions to be raised when a requirement is violated. For example, such file can contain a description of a QN and several requirements (e.g., a threshold on average residence time or on average queue length) with their associated exceptions aimed at managing the violations. *Model Plugins* provide to KAMI the ability to handle different and new models, by interpreting model files and their requirements. Finally, *Input Plugins* provide to KAMI the ability to connect models with the run-time world in which the implemented system is running. The purpose of such plugins is to handle different input formats and protocols for run-time data (e.g, socket, RMI, etc.).

By supporting a feedback loop, which allows estimates of model parameters to be tuned according to run-time data, models better capture real system behaviors. In addition, updated models evolve at run time following the changes in the environment. Moreover, the more data we collect from the running instances of the system, the more precise our models will be. Indeed, model parameters will eventually converge to real values characterizing the modeled system. Conceptually, KAMI establishes a *feedback control loop* between models and implementation, as shown in Figure 1.

3 Modeling with Queueing Networks in KAMI

In this section we discuss how QN models are incorporated in KAMI. We start with a succinct introduction to QNs, we then discuss modeling issues in the context of open-world software, and finally we focus on parameter estimation.

3.1 Introduction to Queueing Networks

QNs [6,20] are a widely adopted modeling technique for performance analysis. QNs are composed by a finite set of: (1) *Service Centers*, (2) *Links*, (3) *Sources*,

and (4) *Delay Centers*. Service centers model system resources that process customer requests. Each service center is composed by a *Server* and a *Queue*. Queues can be characterized by finite or infinite length. In this paper we focus on service centers with infinite queues. Service centers are connected through *Links* that form the network topology. Servers process *jobs*—hereafter we refer to requests interchangeably with the term *jobs*—retrieved from their queue following a specific policy (e.g., FIFO). Each processed request is then routed to another service center through connections provided by links. More precisely, each server, contained in a every service center, picks the next job from its queue (if not empty), processes it, and selects one link that routes the processed request to the queue of another service center. It is possible to specify a policy for the link selection (e.g., probabilistic, round robin, etc.). The time spent in every server by each request is modeled by continuous distributions such as exponential or Poisson distributions. Jobs are generated by source nodes connected with links to the rest of the QN. Source nodes are also characterized by continuous time distributions that model request interarrival times. Finally, delay centers are nodes of the QN connected with links to the rest of the network exactly as service centers, but they do not have an associated queue. Delay centers are described only by a service time, with a continuous distribution, without an associated queue. They correspond to service centers with infinite servers.

QNs may be solved analytically by evaluating a set of performance measures, such as:

- *Utilization*: It is the ratio between the server’s busy time over the total time.
- *Response Time*: It is the interval between submission of a request into the QN and output of results.
- *Queue Length*: It is the average queue length for a given service center.
- *Throughput*: It is the number of requests processed per unit of time.

The above measures are defined for a single service center, by they can also apply to the whole network.

As we mentioned, QNs are characterized by several model parameters (CITD, STD, RP). With specific values for these parameters it is possible to evaluate the performance measures of interest by solving equations.

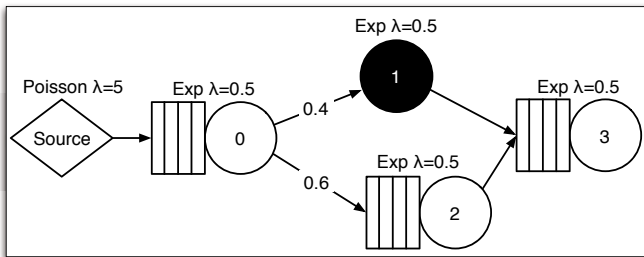


Fig. 3. Queuing Network Example

Figure 3 for example represents a QN with: (1) three service centers regulated by exponential distributions with $\lambda = 0.5$, which corresponds to an average service time equal to $1/\lambda = 2s$, (2) a source node regulated by Poisson distribution with $\lambda = 5$, which correspond to an average interarrival time equal to $5s$, (3) a delay center modeling an external component or service regulated by an exponential distribution with $\lambda = 0.5$, which corresponds to an average service time equal to $1/\lambda = 2s$ and (4) routing probabilities as shown in the figure. A complete description of QNs is beyond the scope of this paper and for a more complete introduction the reader is referred to [2,7,8,19,20].

3.2 Queueing Networks for Open Systems

We decided to use QNs to evaluate performance of open systems because they provide a good balance between a relative high accuracy in results and efficiency [20]. However there are pitfalls that software engineers must take into account while modeling open systems with QNs. Hereafter we discuss and categorize the most crucial ones.

In open systems, components may fall into different categories. First, they may differ in the way they are used (*use mode*). Their use may be *exclusive*; that is, the component is only used by the currently designed application. In this case, the component may be modeled as a service center, since we have full control of the flows of requests into its input queue. In other cases, the component is *shared* among different applications, which we may not know, although they concurrently access it. The component cannot be modeled as service center because other jobs which we cannot control also can access the service. In such a case, the component can be more simply—but less accurately—modeled as a delay center.

As an example of these two cases, consider a component which provides functionalities for video encoding and decoding. In case it is a component-off-the-shelf (COTS), which is deployed within the current application and it is used exclusively by it, then the designer has full control and visibility of its activations, and thus it can be modeled by a service center. If, however, the tool is offered by a provider as a Web service, it is potentially accessed by many clients, and the designer has no control nor visibility of the queues of requests.



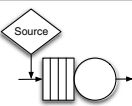

Another key factor that must be considered by the modeler is *visibility* of the internals of the component. Both accuracy of and trust of of the component's performance characteristics depend on how detailed is the designer's knowledge of the component's internals. If an accurate description of the component's architecture is available, its performance can be predicted quite accurately, for example using a design-time tool like Palladio [4]. If instead the component is a black-box, like in the case of Web services, the designer must rely on less trustable figures published by the service provider or inferred by past observations. Note that visibility is often related to *ownership*. If one owns a component, then normally one also has full access to its internals, and conversely. Furthermore, it is also related with *stability*. Whenever a component is owned, it only evolves under control of the owner. If an accurate model of the component is

available, there is no need to monitor the component at run time for possible deviations and, consequently, update of the model. The above discussion leads to the following main component categories:

- *White-Box (WB)* components. Their internal architecture is fully visible and understood by the designer; for example, they have been developed in-house. In addition, their use is exclusive by the current application.
- *Grey-Box (GB)* components. Their use is exclusive, but their internals are not known; only the executable version of the component is available. COTS are a typical example.
- *White-Box Shared (WBS)* components. The designer has full visibility of the component, which however is not used exclusively within the application being developed. An example is an in-house developed Web-service that is used by the current application, but is also exported for use by others.
- *Black-Box (BB)* components. The designer has no visibility of the internals of the component, whose use is shared with other unknown clients. An example is an externally developed Web service developed by third parties that is available on-line.

Table 1 summarizes the previous discussion by showing the main categories of components, the choices we made for modeling them via QNs, and the graphical notation we use.

Table 1. QN Notation for Open Systems

Notation	Name	Use Mode	Visibility	Description
	White-Box (WB)	exclusive	yes	service center
	Grey-Box (GB)	exclusive	no	service center
	White-Box Shared (WBS)	shared	yes	service center with source node
	Black-Box (BB)	shared	no	delay center

3.3 Parameter Estimation and Run-Time Update

QN models are based on numerical parameters (STD, RP, CITD). The current KAMI implementation supports QNs composed by service centers and delay centers characterized by exponential distribution of service time. In this section we review how the designer proceeds in KAMI to derive a model of the system at design time. We then show how model parameters may be automatically updated at run time to capture the possible run-time evolution of certain components or usage profiles, or even inaccurate design-time estimates.

At design time, the modeler specifies the exponential distribution for each STD through its parameter λ , where $1/\lambda$ corresponds to the mean value of the distribution and thus the *average service time* (the same applies for CITDs). For WB components the STD parameter is extracted through design-time analyses or simulations that produce the average service time. In the case of GB or BB components, we assume that the value is published by the component provider. For WBS components, besides the average service time (which is derived as for WB components), engineers must also specify the expected arrival rate of requests issued by external users. This value is equal to $1/\lambda$ where λ is the parameter of the exponential distribution representing the CITD of such requests. It is also necessary to specify CITD for the source nodes that represent the requests issued by users of the system under design and RP. The values of these parameters encode domain knowledge concerning usage profiles.

Let us now address the issue of how model parameters may be updated at run time. Refined estimates of routing probabilities (RP) are handled in KAMI through the Bayesian estimation technique illustrated in [15]. As for the other parameters, since KAMI supports exponential distributions to model service times and arrival rates we could adopt the Maximum Likelihood Estimator (MLE). In the case of exponential distribution it would be:

$$\hat{\lambda} = \frac{1}{\sum_{i=1}^N \frac{t_i}{N}} = \frac{1}{\bar{t}} \quad (1)$$

where t_i is the i^{th} execution time sampled at run-time over a total of N samples. By using this formula, it would be possible to estimate CITD and STD parameters, given runtime data representing interarrival times of requests and component execution times.

However, the MLE ignores the initial estimates provided by domain knowledge or published in specifications by components providers (*a-priori knowledge*) and considers only the information extracted by running instances of the system (*a-posteriori knowledge*). Consequently, we propose a different approach that takes into account both the contributions as shown in formula 2:

$$\hat{\lambda} = \alpha\lambda_0 + (1 - \alpha)\frac{1}{\bar{t}} \quad (2)$$

where λ_0 is the initial estimate and $0 \leq \alpha \leq 1$ is a parameter that the software architects use to express their confidence with respect to initial estimates. A reasonable value for α is $\frac{1}{N}$ which implies that for low values of N we trust more the initial estimates than the ones computed by run-time data. Conversely, as N becomes bigger our trust in estimates computed at run time grows consequently. It is important to notice that the extreme values $\alpha = 0$ and $\alpha = 1$ respectively correspond to ignoring a-priori knowledge and vice-versa.

4 A Running Example

This section illustrates an application of our approach to a concrete example, which falls into the Web service scenario. The example represents a typical e-commerce application that sells on-line goods to its users, by integrating the

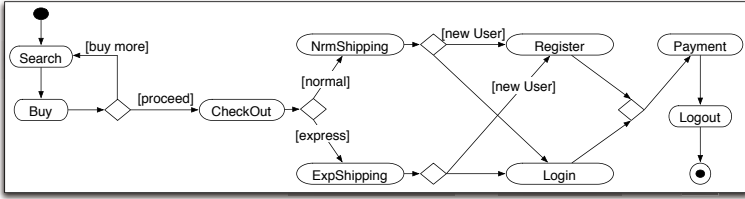


Fig. 4. Activity Diagram Example

following third-party services: a *Payment Service* and a *Shipping Service*. The Payment Service provides a safe transactional payment service. The Shipping Service handles shipping of goods to the customer’s address. It provides two different operations: *NrmShipping* and *ExpShipping*. The former is a standard shipping functionality while the latter represents a faster and more expensive alternative. The integrated service also comprises several internal components, such as: *Search*, *Buy*, *CheckOut*, *Register*, *Login*, and *Logout*, respectively in charge of searching in the database of goods, managing the shopping cart, computing the total cost of orders, managing user registration, authentication, and logout. The Search component is built on top of an externally developed library (a COTS component) that optimizes queries over a database (e.g., [28]). Figure 4 shows the workflow of the integrated service as an activity diagram. CheckOut is implemented as a service, which is shared among other e-commerce applications. It is managed by the same organization that is in charge of the current application. We assume that the integrated system must satisfy a number of performance requirements, such as:

- R1: “Average response time is $\leq 3s$ ”
- R2: “Search component utilization ≤ 0.7 ”
- R3: “System throughput ≥ 1.8 job/s”

In order to analyze performance requirements of the integrated service, starting from the activity diagram of Figure 4, we must build an analysis model based on QNs. This is an example of a model-to-model transformation that will be supported by KAMI; at this stage the environment only provides a tool that transforms an activity diagram (source model) into a Continuous Time Markov Chain as a target model, which is then used for reliability analysis. In this case, model transformation is done manually and produces the QN in Figure 5 built exploiting the taxonomy described in Section 3.2 as shown in Table 3.

Parameters of the model are then partly elicited from domain experts (about usage profiles), partly estimated by looking into WB components by using such tools as Palladio [4] or JMT [5], and partly obtained from external sources in the case of components/services with external visibility. These are collectively shown in Table 2. From estimated (or published) execution times and from domain knowledge it is straightforward to configure the exponential distributions characterizing the QN.

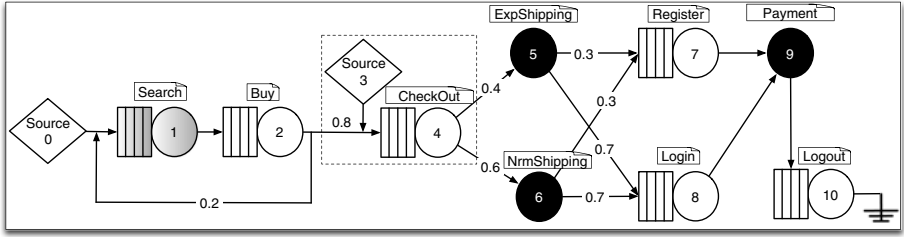


Fig. 5. Queuing Network Example

Table 2. Model Parameters

Usage Profile	Value
<i>% of users that choose an express shipping</i>	40%
<i>% of users that searches again after a buy operation</i>	20%
<i>Probability that an incoming user is a new user</i>	0.3
<i>Average Request Interarrival Time</i>	0.8s
<i>Average Request Interarrival Time for the CheckOut Web service</i>	1.5s
Internal Components	Expected Execution Time
<i>Buy</i>	0.15s
<i>CheckOut</i>	0.15s
<i>Login</i>	0.2s
<i>Logout</i>	0.2s
<i>Register</i>	0.2s
External Components	Expected Execution Time
<i>NrmShipping</i>	0.5s
<i>ExpShipping</i>	0.5s
<i>Payment</i>	0.35s
<i>Search</i>	0.43s

Table 3. QN Model

Node Number	Type	Classification	λ_0
0	source node	n/d	1.25
1	service center	GB	2.326
2	service center	WB	6.667
3	source node	WBS source node	0.667
4	service center	WBS service center	6.667
5	delay center	BB	2
6	delay center	BB	2
7	service center	WB	5
8	service center	WB	5
9	delay center	BB	2.857
10	service center	WB	5

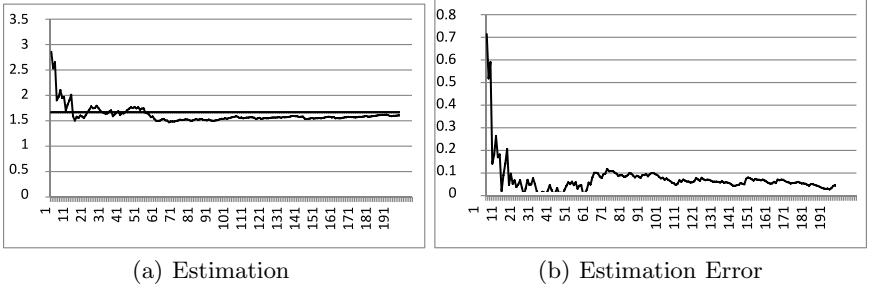


Fig. 6. Numerical Simulations

By running the resulting QN model, we obtain the following results, which show that requirements are satisfied:

- “Average response time = 2.87s”
- “Search component utilization = 0.67”
- “System throughput = 1.92 job/s”

Suppose now that we are at run time. KAMI can calibrate the values of model parameters by observing the real values at run time. Let us consider requirement R1 and let us suppose that the actual average execution time for Payment Service is 0.6s, which is higher than the published value (0.35s). If we knew this actual value, we could deduce an average response time of 3.12s, which violates requirement R1. KAMI can detect this violation thanks to its capability to estimate the value of parameters. In fact, suppose that the following trace of registered execution times for Payment Service has been observed: $T = (0.5, 0.5, 0.7, 0.6, 0.7)$, whose the average value is 0.6s. According to formula 2 (assuming $\alpha = \frac{1}{N}$), KAMI estimates the CITD for the payment service with $\hat{\lambda} = 1.9$, which produces a better approximation of the actual average response time:

$$\hat{\lambda} = \frac{1}{5}\lambda_0 + \left(1 - \frac{1}{5}\right) \frac{1}{\sum_{i=1}^5 \frac{t_i}{5}} = 1.9$$

After updating the QN model with the new estimated value, KAMI discovers at run time that the response time of the system is equal to 3.05s, which violates requirement R1. Having detected the failure, KAMI can activate a reaction associated with the violated requirement, which can perform recovery actions. Efficient and effective recovery strategies are still open issues that we plan to investigate in our future work as stated in Section 6.

Figure 6(a) shows estimates produced by KAMI for the STD of the Payment Service. Notice that the estimate for λ initially corresponds to the a priori knowledge provided by the service provider (i.e., $\lambda_0 = \frac{1}{0.35} = 2.857$ and gradually converges to the real value which is equal to $\frac{1}{0.6} = 1.667$). The effectiveness of the approach can be measured considering the estimation error (E) computed as:

$$E = \frac{| \textit{estimated paramter-real parameter} |}{\textit{real parameter}}$$

Figure 6(b) shows the estimation error for the estimate in Figure 6(a). The example illustrated in this section and its simulation were conceived for illustrative purposes and do not represent a complete validation of the approach which is beyond the scope of this paper and part of our future work.

5 Related Work

Many methods exist to support reasoning on non-functional properties of software, based on models that are analyzed at design time. The approach investigated by KAMI differs from these cases because it extends the lifetime of models to also cover run time. By collecting run-time data, it is possible to estimate parameters and progressively improve the accuracy of the models. Calibrated models, in turn, may produce more accurate verification of the requirements. Only few other similar approaches are described in the literature. For example, [32] describes a methodology for estimation of model parameters through a Kalman filter, which is fed with data produced by a monitor. This approach is similar to our proposal since it combines new data with initial estimates, weighting them by their confidence and it adapts to changes that occur in the modeled system considering constant the system structure. Their approach is quite general and effective, while in our proposal a specific statistical machinery has to be defined for every supported model and developed in KAMI through plugins. Indeed, we plan to integrate in KAMI a plugin for Kalman filtering, as described in [32].

A recent work [13] presents a framework for component reliability prediction whose objective is to construct and solve a stochastic reliability model allowing software architects to explore competing architectural designs. Specifically, the authors tackle the definition of reliability models at architectural level and the problems related to parameter estimation. The problem of correct parameter estimation is also discussed in [18,30], where shortcomings of existing approaches are identified and possible solutions are proposed.

Several approaches have been proposed in literature that deal with performance aspects of open systems and in particular of web services and their composition. In particular, [23] describes a framework for composed service modeling and QoS evaluation. Service composition is represented by a directed weighted graphs where each node corresponds to service and edge weights represent the transition probabilities of two subsequent tasks.

In [1], [11], [14], [21], and [22] queueing models have been used to identify performance problems in web services or web applications. All these papers describes approaches that work only at design time. Simulation-based approaches that analyze the performance of composite web services are described by [12], [29], and [31]. Besides service time, they explicitly take into account communication latency, which in our approach is hidden and encapsulated within other parameters. However these proposals rely on time-consuming simulations that are not suitable for run-time analysis.

An approach that works at run time and monitors the execution to trigger re-planning for composite Web services is proposed by [9]. This approach is not based on QNs but on a reduction-based mathematical model for QoS computation (a similar approach is also described by Cardoso et al in [10]) and it exploits genetic algorithms. Several current research efforts focus on performing self-healing reactions at run time, to automatically adapt the software to changes in the environment. These, however, are out of the scope of the work we described here.

6 Conclusion and Future Work

KAMI aims at providing a comprehensive support to quality-driven and model-driven development and operation of software systems that are situated in an open world. In particular, it focuses on non-functional quality attributes. The KAMI approach comes with estimation techniques that specifically support the update at run time of several non-functional models. The initial results obtained with the KAMI approach are described in [15], where a Discrete Time Markov Chain (DTMC) is used to assess the reliability of service compositions. A Bayesian technique is proposed by [15] for run-time estimation of the parameters of the model. This paper extended the approach to QNs, which may be used to reason about performance properties.

The main goal of our future research is to integrate KAMI with other modeling approaches that fit other kinds of analysis purposes. Different models may in fact be used to reason about different quality attributes of a complex system. They may provide complementary views which facilitate QoS analysis. The crucial problem, however, is to guarantee that the different views are sound, i.e., they do not contradict each other, so that the results of analysis can be composed. We plan to do so by defining a set of predefined model-to-model transformation rules. Moreover, we plan to enrich our current KAMI plugin for QNs to support shared resources and multi-class and layered QNs. A complete investigation and numerical validation of the approach described in this paper and the investigation on how it deals with phase transitions—a common scenario in modern software systems—is part of our future work. Finally, we plan to investigate effective and efficient recovery strategies aimed at closing the control loop among the model and its implementation established through KAMI.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom and by Project Q-ImPRESS (FP7-215013) funded under the European Union's Seventh Framework Programme (FP7).

References

1. Badidi, E., Esmahi, L., Serhani, M.A.: A Queuing Model for Service Selection of Multi-classes QoS-aware Web Services. In: Third IEEE European Conference on Web Services, 2005. ECOWS 2005, pp. 204–213 (2005)
2. Balsamo, S.: Product Form Queueing Networks. In: Reiser, M., Haring, G., Lindemann, C. (eds.) Dagstuhl Seminar 1997. LNCS, vol. 1769, pp. 377–402. Springer, Heidelberg (2000)
3. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. *Computer* 39(10), 36–43 (2006)
4. Becker, S., Koziolok, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP 2007: Proceedings of the 6th International Workshop on Software and Performance, pp. 54–65. ACM, New York (2007)
5. Bertoli, M., Casale, G., Serazzi, G.: The jmt simulator for performance evaluation of non-product-form queueing networks. In: Annual Simulation Symposium, Norfolk, VA, US, pp. 3–10. IEEE Computer Society, Los Alamitos (2007)
6. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing networks and Markov chains: modeling and performance evaluation with computer science applications. Wiley-Interscience, New York (1998)
7. Buzen, J.P.: Queueing Network Models of Multiprogramming (1971)
8. Buzen, J.P.: Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM* 16(9), 527–531 (1973)
9. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: QoS-Aware Replanning of Composite Web Services. In: ICWS 2005 Proc (2005)
10. Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web* 1(3), 281–308 (2004)
11. Catley, C., Petriu, D.C., Frize, M.: Software Performance Engineering of a Web Services-Based Clinical Decision Support Infrastructure. *Software Engineering Notes* 29(1), 130–138 (2004)
12. Chandrasekaran, S., Miller, J.A., Silver, G.S., Arpinar, B., Sheth, A.P.: Performance Analysis and Simulation of Composite Web Services. *Electronic Markets* 13(2), 120–132 (2003)
13. Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early prediction of software component reliability. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, pp. 111–120. ACM Press, New York (2008)
14. D’Ambrogio, A., Bocciarelli, P.: A model-driven approach to describe and predict the performance of composite services. In: WOSP 2007: Proceedings of the 6th international workshop on Software and performance, pp. 78–89. ACM, New York (2007)
15. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime adaptation. In: ICSE 2009: The 31th International Conference on Software Engineering, Vancouver, Canada (to appear, 2009), <http://home.dei.polimi.it/tamburrelli/icse09.pdf>
16. Gallotti, S., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Quality prediction of service compositions through probabilistic model checking. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281. Springer, Heidelberg (2008)
17. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. *IBM Systems Journal* 42(1), 5–18 (2003)

18. Gokhale, S.S.: Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. Dependable Sec. Comput.* 4(1), 32–40 (2007)
19. Lam, S.F., Chan, K.H.: *Computer capacity planning: theory and practice*. Academic Press Professional, Inc., San Diego (1987)
20. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River (1984)
21. Litoiu, M.: Migrating to web services: a performance engineering approach. *Journal of Software Maintenance and Evolution: Research and Practice* 16(1-2), 51–70 (2004)
22. Marzolla, M., Mirandola, R., Milano, I.: Performance Prediction of Web Service Workflows. In: Overhage, S., Szyperski, C., Reussner, R., Stafford, J.A. (eds.) *QoSA 2007*. LNCS, vol. 4880, pp. 127–144. Springer, Heidelberg (2008)
23. Menascé, D.A.: QoS Issues in Web Services. *IEEE Internet Computing*, 72–75 (2002)
24. Meservy, T.O., Fenstermacher, K.D.: Transforming software development: an mda road map. *Computer* 38(9), 52–58 (2005)
25. Moreno, G.A., Merson, P.: Model-Driven Performance Analysis. In: Becker, S., Plasil, F., Reussner, R. (eds.) *QoSA 2008*. LNCS, vol. 5281, pp. 135–151. Springer, Heidelberg (2008)
26. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engg.* 15(3-4), 313–341 (2008)
27. Papazoglou, M.P., Georgakopoulos, D.: Service-Oriented Computing. *Communications of the ACM* 46(10), 25–28 (2003)
28. Apache Lucene Project, <http://lucene.apache.org/>
29. Silver, G., Maduko, A., Jafri, R., Miller, J.A., Sheth, A.P.: Modeling and Simulation of Quality of Service for Composite Web Services. In: *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics* (2003)
30. Smith, C.U., Williams, L.G.: *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City (2002)
31. Song, H.G., Ryu, Y., Chung, T., Jou, W., Lee, K.: Metrics, Methodology, and Tool for Performance-Considered Web Service Composition. In: Yolum, p., Güngör, T., Gürgeç, F., Özturan, C. (eds.) *ISCIS 2005*. LNCS, vol. 3733, pp. 392–401. Springer, Heidelberg (2005)
32. Zheng, T., Woodside, M., Litoiu, M.: Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering* 34(3), 391–406 (2008)

Compositional Prediction of Timed Behaviour for Process Control Architecture

Kenneth Chan and Iman Poernomo

Department of Computer Science, King's College London
Strand, London WC2R 2LS UK

Abstract. The timing of properties is an essential consideration in the design, implementation and maintenance of embedded software development. In this paper, we present an approach to the prediction of timed and probabilistic nonfunctional properties of process control architectures. Our approach involves a novel compositional approach to model checking of statements in Probabilistic Computational Tree Logic (PCTL).

1 Introduction

The satisfaction of timing constraints is a central priority to the design, development and maintenance of process control systems. While control theory is a well understood branch of mathematics that enables us to determine timing requirements for a system, the task of guaranteeing that architectures satisfy these requirements is less understood. Furthermore, as process control systems become dependent on component-based paradigm, there is an increased need for effective, scalable, compositional tools and techniques for requirement verification.

This paper presents a methodology for predicting the likelihood of timed constraints holding over component-based process control software. We use Probabilistic Computational Tree Logic (PCTL) as a formal language for specifying required properties. PCTL enables a logic-based approach to specifying and verifying timed, probabilistic properties over dynamic models of system behaviour. PCTL has been used effectively for verifying such properties over small systems, as it provides a flexible language and is decidable. However, the model checking algorithms for PCTL involve a form of Markov-chain analysis that quickly results in state explosion for even moderately sized systems.

We solve this problem by developing a novel, compositional approach to verifying PCTL properties over process control architectures. The key aspect of this approach is the treatment of a component's behaviour as parametrized over the timing properties of a deployment context. The deployment context is described in terms of PCTL formulae which are interpreted to give best, average and worst case timing profiles for the component. Properties can then be checked over the client's behaviour instantiated by these profiles. If we are satisfied that a component has the required properties, it can then be added to a larger context within which other components may be analysed. When adding the component

to a larger context, we discard its behavioural information, apart from its PCTL contracts, avoiding state explosion in future analysis. It is in this way that our approach is compositional and scalable.

To the best of our knowledge, no existing approach has adapted PCTL to compositional verification of component-based process control systems, nor on relating compositional verification to runtime verification within the process control domain.

This paper is organized as follows:

- Section 2 discusses the aspects of process control systems that are of concern to our methods.
- Section 3 describes the syntax and semantics of PCTL.
- Section 4 sketches our compositional approach to the verification of PCTL constraints.
- Conclusions and related work are discussed in section 5.

2 Process Control Architectures

Process systems are typically built according to a hierarchical, bus-based architecture of the type depicted in Fig. 1. Process systems involve three layers. (1) *Field Management*. Smart field devices perform some work related to the domain and also provide a wide range of information on the health of a device, its configuration parameters, materials of construction, etc. Controller components communicate with and coordinate field devices, typically through some kind of field bus infrastructure. (2) *Process Management*. Distributed Control Systems and SCADA systems are used to monitor and control overall processes, communicating to devices and controllers. (3) *Business Management*. The information collected from the processes can be integrated into a wider business system that might, for instance, manage the financial aspects of the manufacturing process.

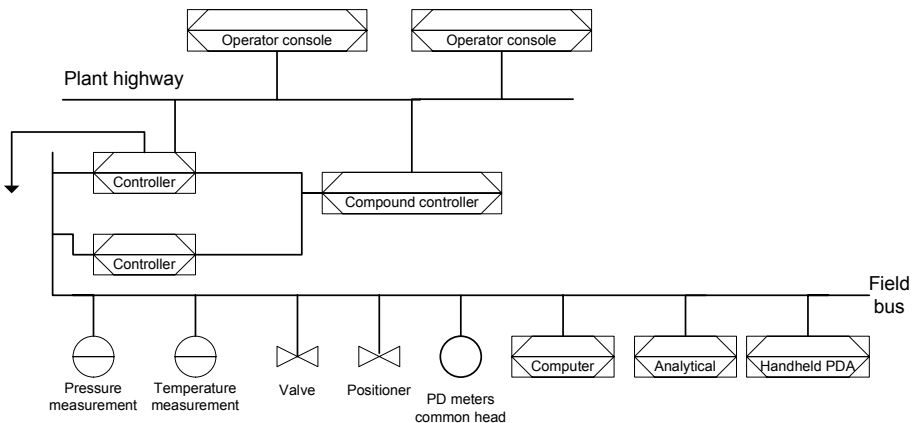


Fig. 1. A typical process system architecture

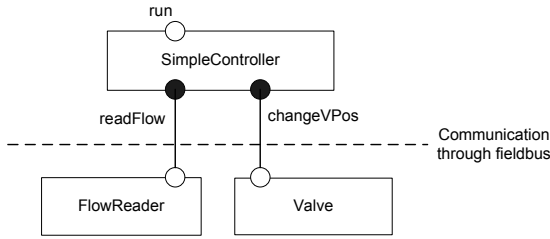


Fig. 2. A simple process control architecture

Providing this information in a consistent manner to client applications minimizes the effort required to provide this integration.

We are concerned the prediction of timing properties for a coarse grain view of an overall process control system architecture. In particular, we are concerned with verifying the timing properties of a client component operating at some level of the process control hierarchy with respect to its server components. Such concerns are key to developing controller software at the field management level and to developing controller and operator software at the process management level. This work is therefore in contrast to finer grain approaches to the development of particular components in a process control system (for instance, the PECOS methodology [3] is appropriate for developing component-based software for individual field devices). That work is complementary to ours, as we require that, at the lowest level, basic field devices provide some contractual guarantee of timing properties.

Example 1. Fig. 2 depicts a simple example of a lower level process control architecture. (The example is based upon that given in [3, 9-11]. It consists of two field devices, FlowReader and Valve. These devices are used by a process control component, SimpleController, that ensures an even flow of liquid in a pipe by controlling a valve. On detecting an deviation from the required flow by reading the current flow value from the flow meter (achieved by sending a signal to the meter's provided port `readFlow`), the control component should respond by altering the valves stem position by writing a new valve position to the valve positioner (achieved by sending a signal to the valve's provided port `changeVPos`). This response must occur within a 10 seconds if the equipment at the receiving end of the pipe is not to become overloaded.

Determining whether a component satisfies a required constraint will depend on two factors. (1) *The behavioural profile of the controller itself.* The response may on average involve quite a complex computation in order to calculate the new valve angle. (2) *The deployment context of the controller.* The response will depend on timing properties of the field components. As these components are typically built from composition of third-party elements, detailed behavioural specifications may not be known. However, the third-party vendors should provide some form of contractual specification of their components' timing behaviour. The combined effect of both factors must be understood in order to verify component requirements.

3 PCTL

Probabilistic Computational Tree Logic (PCTL) [4] enables us to specify requirements on the timing of when Boolean properties should hold over a system's execution. The language is probabilistic, in that requirements can be associated with a likelihood of occurrence.

3.1 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. 3. B is basic boolean statements T is timed statements and P are PCTL statements. The syntax is parametric over a set of atomic propositions $atom$. int ranges over integers and $float$ ranges over reals between 0 and 1.

The informal meaning of formulae as a description of component behaviour is understood with respect to the idea of a component that can execute a multiple number of runs. Each run is invoked by an external 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 represents a discrete time step of some length. 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.

Propositional formulae B should be interpreted as statements about the system that can be verified to be true or false at any state during an execution.¹ *Timed formulae* T 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 *the statement A until B steps: s* is true over the execution of a component 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* P are understood in terms of the probability of corresponding timing formulae being true for future runs. For instance, *A until B steps: s prob: p* is true over a number of completed system runs, if the corresponding nonprobabilistic formula *A until B steps: s* is likely to be true for the next run with a probability of p . The case is similar for *A leadsto B steps: s prob: p* .

3.2 Formal Semantics

PCTL formulae are decidable over models of system behaviour described by Probabilistic Kripke Frames. We describe how this is done.

We require the definition of a probabilistic deterministic finite state machine.

¹ Our work can be trivially adapted to consider these as built using boolean predicates over real valued system properties, but for reasons of space we will consider them as purely atomic.

$B := atom \mid not\ B \mid B\ and\ B \mid B\ or\ B$
 $T := B \mid B\ until\ T\ steps:\ int \mid F\ leadsto\ T\ steps:\ int$
 $P := B \mid B\ until\ F\ steps:\ int\ prob:\ float \mid F\ leadsto\ F\ steps:\ int\ prob:\ float$

Fig. 3. Our contractual specification language

Definition 1 (Probabilistic Deterministic Finite State Machine). A Probabilistic Deterministic Finite State Machine (PFSM) is a tuple

$$D = \langle Z_D, \delta_D, initial_D, final_D \rangle$$

where Z_D denotes the set of states. $initial_D \in Z_D$ is a designated initial state, while $final_D \in \mathcal{P}(Z_D)$ is a set of final states. $\delta_D : Z_D \times Z_D \rightarrow P$ is a total function, where P is the real numbers between 0 and 1. The sum of the probabilities of transitions exiting from a state z in Z_D is equal to 1.

Execution sequences are sequences of states (in terms of the component semantics to follow, they arise as the result of transitions caused by the input of events or the output of actions and the probabilities of the appropriate transition not failing). Given an execution sequence $\pi = s_1 \dots s_n$ we write $\pi(k)$ for the k -th state of π , s_k . Let $Path_{PM}$ denote the set of all the paths which a PFSM D can take, and $Path_D(s)$ denote the set of all paths starting at state s . We define $Prob(\Delta)$ to be a probability measure over paths such that

$$1) \text{ If } \Delta \subseteq \{\omega \in Path_D(s_0) \mid \omega \text{ extends } s_1, \dots, s_n\} \text{ then}$$

$$Prob(\Delta) = Prob(s_0, s_1, \dots, s_k) = P(s_0, s_1) * P(s_1, s_2) * \dots * P(s_{n-1}, s_n)$$

- 2) If $\Delta \subseteq \{\omega \in Path_D(s_0) \mid \omega \text{ extends } s_0\}$ then $Prob(\Delta) = 1$, and
 3) For any countable set $\{X_i\}_{i \in I}$ of disjoint subsets of $Path_D(s_0)$,

$$Prob\left(\bigcup_{i \in I} X_i\right) = \sum_{i \in I} Prob(X_i)$$

The semantics of our logic is given with respect to a *Probabilistic Kripke Frame*, essentially a PFSM where atomic propositions are associated with states:

Definition 2 (Probabilistic Kripke Frame). A Probabilistic Kripke Frame (PKF) for a PFSM is a tuple $S = \langle D_S, Prop_S, h_S \rangle$ where D_S is a PFSM, $Prop_S$ is a set of atomic propositions and $h : Z_{D_S} \rightarrow \mathcal{P}(Prop_S)$.

The truth value of PCTL formulae can be computed for a given PKF K , yielding the satisfaction relation

$$s \models_K f$$

which says that the formula f is true at state s inside the structure K . The same relation holds if f is path formula and s is a path inside the structure K .

Definition 3. The relation \models_K is defined as follows, for arbitrary PCTL formula f , state s and path π .

- $s \models_K \text{atom}$ ($\text{atom} \in \text{Prop}_K$) iff $\text{atom} \in h(s)$,
- $s \models_K \text{not } f$ iff $s \text{ not } \models_K f$,
- $s \models_K f$ and g iff $s \models_K f$ and $s \models_K g$,
- $\pi \models_K f$ until g steps: t iff there is an $i \leq t$ such that $\pi(i) \models_K g$ and $\pi(j) \models_K f$ for all $0 \leq j < i$,
- $\pi \models_K f$ leadsto g steps: t iff, given $\pi(0) \models_K f$, for some $i \leq t$, $\pi(i) \models_K g$.
- $s \models_K f$ until g steps: t prob: p iff

$$\text{Prob}\{\pi \in \text{Path}(s) \mid \pi \models_K f \text{ until } g \text{ steps: } t\} \geq p$$

- $s \models_K f$ leadsto g steps: t prob: p iff

$$\text{Prob}\{\pi \in \text{Path}(s) \mid \pi \models_K f \text{ leadsto } g \text{ steps: } t\} \geq p$$

The algorithms for deciding truth or falsity of PCTL statements against a PKF are given in, for example, [4]. However, because the algorithms are essentially based upon Markov chain analysis, the problem of checking PCTL formulae against a system quickly becomes intangible for even medium sized systems. A typical system description might consist of more than 500 states, which would lead to a severe state explosion in the algorithms. This is why it is not useful to employ PCTL as a requirements language if the architecture's behaviour is modelling using such a naïve approach.

4 Compositional Semantics

We propose a solution to this problem by means of a compositional approach to PCTL evaluation. Essentially we evaluate PCTL formulae against behavioural descriptions for individual components. We then discard the behavioural descriptions themselves when composing these components with a client component. In turn, we can evaluate PCTL formulae against a smaller behavioural description of the client component – this description is formed from a behavioural profile together with knowledge about the composed components obtained from their contracts. We can then discard this description when compositing the client with other clients, using only the PCTL formulae as contracts in this, and so on. At any point in the process of composition, the behavioural description's size should be of a comparable scale to any other point. As a result, there is no state explosion in the model checking and the problem of verification becomes tangible over larger architectures.

The approach is suited to coarse-grain process control style architectures, due to their hierarchical nature: hierarchy means that controllers control field devices, but field devices do not control controllers, and similarly for higher levels. The approach is not as well suited for other domains, such as enterprise software development, which generally requires a greater circularity of dependence.

Our approach to predicting properties about an architecture relies on considering two kinds of components: grey box components, which provide some behavioural description outlining a behavioural profile and describing how the

component calls other components via its required ports; and black box components, that only provide PCTL statements as their behavioural description. Our approach involves taking a grey-box description of a component and checking required behaviour of its provided ports parametrized a *deployment context*, consisting of black-box component descriptions.

4.1 Grey-Box Components

A grey-box component specification consists of provided and required interfaces (given as a list of ports) and a probabilistic behavioural semantics, defining what happens when a provided port is invoked, calls to components connected to the required ports.

A behavioural profile explains the relation between provided port invocations and resulting calls to required ports.

Definition 4 (Behavioural profile). *A behavioural profile is a tuple*

$$D = \langle A_D, Z_D, Prop_D, \gamma_D, initial_D, final_D, h_D \rangle$$

satisfying the following conditions:

- A_D is called the action alphabet and must include a unique element τ ,
- Z_D denotes the set of states.
- $initial_D \in Z_D$ is a designated initial state, while $final_D \in \mathcal{P}(Z_D)$ is a set of final states.
- $Prop_D$ is a set of atomic proposition names.
- The transition function $\gamma_D : Z_D \times Prop \times A_D \times Z_D \rightarrow [0, 1]$ is total, such that, for each $t \in Z_D$, either of the following cases hold
 - There is no $s \in Z_D$ such that $\gamma_D(t, P, a, s) > 0$ for $P \neq true$, and $\sum_{s \in Z_D} \gamma_D(t, true, a, s) = 1$.
 - There is no $s \in Z_D$ such that $\gamma_D(t, P, a, s) > 0$, except for the cases where, for some $s_1, s_2 \in Z_D$, $\gamma_D(t, P, \tau, s_1) = 1$ and $\gamma_D(t, \text{not } P, \tau, s_2) = 1$. In this case, we call s a decision state. P is called the guard condition for the decision state.
- h is a valuation function $h : Z_D \rightarrow \mathcal{P}(Prop_D)$.

A behavioural profile is essentially a PKF with binary guarded choice transitions. Transitions specify three kinds of internal process, each taking one time step: $\gamma(s, P, a, s') = p$ ($a \neq \tau$) should be interpreted meaning that the component will, with probability p call required port a to move from state s to s' , $\gamma(s, P, \tau, s') = p$ means that the component will, with probability p , perform some internal computation to move from state s to s' , and $\gamma(s, P, a, s') = 1$ means that, if P is true for the component at state s (subject to deployment in a particular context) then action a should be performed to go to state s' .

Definition 5 (Grey-box component). *A grey-box component*

$$\langle P, R, U \rangle$$

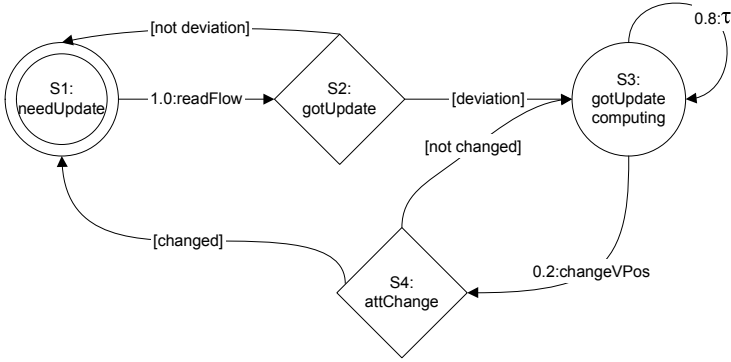


Fig. 4. Visual representation of the behavioural profile for SimpleController.run

is a pair of provided and required port names, P and R , together with behavioural profiles $U_p \in U$ for each provided port $p \in P$, such that the actions for U_p are a subset of the required ports R .

Example 2. The controller component SimpleController can be represented as a grey-box component $\langle \{\text{run}\}, \{\text{readFlow}, \text{changeVPos}\}, U \rangle$, where U is the behavioural profile for SimpleController.run, depicted in Fig. 4. The initial state is represented by two circles, decision states are represented as diamonds. A proposition is shown inside the state if it is associated with the state according to the valuation function. Transitions between non-decision states are represented by arrows labelled by actions and probabilities.

In our ongoing example, we will interpret a transition as representing a 1 second time step. In general, the timing interpretation of a transition depends on the problem domain.

We will find the following operations useful in our modelling of component composition. If S_1 , S_2 and S are PKFs, p is a real number between 0 and 1 and f is a state, and U is a behavioural profile. 1) We define $Join(S_1, S_2)$ to be the PKF formed from taking the union of states from S_1 and S_2 , retaining all transitions, and adding a τ transition of probability 1 from any final state of S_1 to the initial state of S_2 . 2) We define $Join(S_1, S_2, p, f)$ to be the PKF formed from taking the union of states from S_1 and S_2 , retaining all transitions, and adding a τ transition of probability p from any final state of S_1 to the initial state of S_2 , and a τ transition of probability $1 - p$ from any final state of S_1 to the state f . The final states of S_2 become the final states of the new PKF.

Definition 6 (Joining PKFs). Take two PKFs

$$S_1 = \langle \langle A_1, Z_1, \delta_1, \text{initial}_1, \text{final}_1 \rangle, Prop_1, h_1 \rangle \text{ and}$$

$$S_2 = \langle \langle A_2, Z_2, \delta_2, \text{initial}_2, \text{final}_2 \rangle, Prop_2, h_2 \rangle$$

Let p be a real number between 0 and 1 and f a set of state names. We define $Join(S_1, S_2, p, f)$ to be the PKF

$$Join(S_1, S_2, p) = \langle \langle A_1 \cup^* A_2, Z_1 \cup^* Z_2 \cup \{f\}, \delta^*, initial_1, final_2 \rangle, Prop_1 \cup Prop_2, h^* \rangle$$

where

$$\delta^*(s, a, s') = \begin{cases} \delta_1(s, a, s') & \text{if } s, s' \in Z_1, a \in A_1 \\ \delta_2(s, a, s') & \text{if } s, s' \in Z_2, a \in A_2 \\ p & \text{if } s \in final_1, a = \tau, s' = initial_2 \\ 1 - p & \text{if } s \in final_1, a = \tau, s' = f \\ 0 & \text{otherwise} \end{cases}$$

and

$$h^*(s) = \begin{cases} h_1(s) & \text{if } s \in Z_1 \\ h_2(s) & \text{if } s \in Z_2 \\ \emptyset & \text{otherwise} \end{cases}$$

3) $Insert(U, s, s', S)$ is the behavioural profile formed from taking U and “inserting” the PKF S between s and s' and “evaluating” any choices that can be made if s' is a decision state, by verifying truth of guards for transitions coming from s' against the final states of S . The operation involves “redirecting” the transition between s and s' to a behavioural profile transition of the same probability and with guard *true* between s and the initial state of S . If s' is a decision state with a transitions to s_2 and s_3 , guarded by propositions P and not P respectively, then the new behavioural profile will set a final state f of S to a behavioural profile transition with guard *true* to s_2 if $f \Vdash P$ and to s_3 if $f \Vdash \neg P$. The full definition of $Insert$ is slightly more involved but we omit it for reasons of brevity.

4.2 Black-Box Components

A black-box context is meant to represent a (possibly compound) executable component, without revealing details of its composition or dependencies. No dynamic behavioural information is provided about the effect of calling a provided port, save that supplied by the port’s contract.

Definition 7 (Black-box component). *A black-box component*

$$B = \langle P, C \rangle$$

is a set of provided port names P a set of PCTL statements, $C = \{C_p \mid p \in P\}$, called the contracts of B .

P	$BPKF(P)$
<i>any Boolean proposition</i>	The PKF $\langle\langle\{\tau\}, \{s\}, \delta, s, \{s\}\rangle, Prop, h\rangle$ such that $h(s) = \{P\}$ and δ is 0 over all arguments.
A until B steps: t prob: p	$BPKF(B)$
A leadsto B steps: t prob: p	$BPKF(B)$

Fig. 5. Best case time PKF for a provided method contract P

The contract C_p is meant to specify how a call to port p will work:

- If C_p is a boolean proposition, then p will only require one state to perform its computation, resulting in C_p being true.
- If C_p is of the form A until B steps: t prob: r , then, with a probability of r , p will require at most t time steps to perform its computation, which will terminate in B being true, with A holding until termination.
- If C_p is of the form A leadsto B steps: t prob: r , then, with a probability of r , p will require at most t time steps to perform its computation, which will begin with A being true, and will terminate in B being true.

Example 3. We will assume that our example field devices have known timing that permit them to have the following representation as black-box components:

$$\text{FlowReader} = \langle\{\text{readFlow}\}, \{\text{true until deviation steps: 2 prob: 0.5}\}\rangle$$

and

$$\text{Valve} = \langle\{\text{changeVPos}\}, \{\text{true until changed steps: 6 prob: 0.99}\}\rangle$$

The idea is that a PCTL contract can be used to provide information about how a black-box component is likely to behave when a provided port is invoked. For our purposes, this information can be understood by obtaining “canonical” PKFs that satisfy the contract. Given any contract, there are a range of possible PKFs that will satisfy the contract, each with different timing. We define best-, average- and worst-case timing PKFs for a contract C , $BPKF(C)$, $APKF(C)$ and $WPKF(C)$, respectively. The definition of $APKF$ is given in Fig.6.

These provide a “spectrum” of possible response behaviours for the component.

Example 4. Fig. 7 (a) depicts

$$APKF(\text{true until deviation steps: 2 prob: 0.5})$$

the $APKF$ for the contract of `FlowReader.readFlow` and (b) the contract for `Valve.changeVPos`,

$$APKF(\text{true until changed steps: 6 prob: 0.99})$$

P	$APKF(P)$
<i>any Boolean proposition</i>	The PKF $\langle\langle\{\tau\}, \{s\}, \delta, s, \{s\}\rangle, Prop, h\rangle$ such that $h(s) = \{P\}$ and δ is 0 over all arguments.
A until B steps: t prob: p	The PKF M_k defined by the recursive equations $M_0 = APKF(B)$ $M_1 = Join(APKF(A)^0, M_0, p, \{f\})$ $M_{i+1} = Join(APKF(A)^i, M_i, 1, \{f\}) \quad 1 < i < k$ where $k = t/2$ if t is even and $(t+1)/2$ if t is odd, $APKF(A)^i$ is $APKF(A)$ with all states renamed to be disjoint from M_i , fa is the set of final states of $APKF(A)^i$ and f is a unique stand disjoint from $Join(APKF(A)^i$ and M_i .

Fig. 6. Average time PKF for a provided method contract

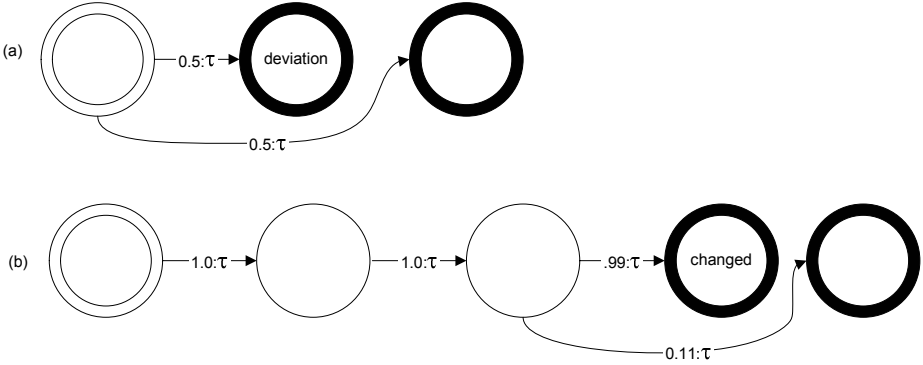


Fig. 7. Visual representation of APKFs for the black-box components' contracts

Theorem 1. *Given any PCTL formula A , if $BK = BPKF(A)$, $AK = APKF(A)$ and $WK = APKF(A)$ then $initial_{BK} \Vdash_{BK} A$, $initial_{AK} \Vdash_{AK} A$ and $initial_{AK} \Vdash_{WK} A$.*

4.3 Component Deployment

Definition 8 (Deployment context). *Let A be a grey-box component $\langle P, R, U \rangle$. Then a deployment context for A , c , is a mapping from R to black-box components of the form*

$$c(r) = \langle P_{c(r)}, C_{c(r)} \rangle$$

such that $r \in P$. The deployment contracts for A are defined to be

$$\{C_{c(r)} \mid r \in R\}$$

For example, it should be clear that the two black-box representations of our field components above form a deployment context for the SimpleController grey-box component.

Given a grey-box client component and a deployment context, we can determine a PKF for the composition of the grey-box component with the context's black-box components. Essentially, this PKF is obtained by 1) inserting a behavioural description of a black-box component wherever its port was invoked by an action call in the behavioural profile and 2) determining what transitions must necessarily be made from a decision state, given the knowledge at that state (that might be informed by previous black-box evaluation).

For example, in the behavioural profile for SimpleController, we need to add the PKF that describes FlowReader.readFlow after the readFlow port has been invoked. Similarly, if the result of the FlowReader device is that a deviation from the norm has occurred, the controller *must* move to computation of the correct valve position – but if the device does not detect a deviation, the controller *must* return to its original state.

The development of the PKF depends on the choice of PKF used for the context's components: a best-case time will result in a faster PKF for the grey-box component, worst-case will result in a slower PKF. Depending on the domain, these three forms of PKF should be examined, in order to get a fair evaluation of the architecture.²

The average-case time PKF for a deployed component is given below – the cases for the other timings are similar.

Definition 9 (Average-case PKF for a deployed component). *Let $A = \langle P, R, U \rangle$ be a grey-box component, $m \in P$ a provided method with profile U_m and let c be a deployment context for A with $\{C_{c(r)} | r \in R\}$ the deployment contracts for A . Assume the set of required ports used by A is $R = \{a_1, \dots, a_n\}$.*

The average-case PKF for m in A deployed within C , $APKF(A, p, C)$, is defined as follows.

First we define the behavioural profile M_n recursively as follows.

1. Let $M_0 = U_m$.
2. Let $\{(s_j, s'_j)\}$ ($j = 0, \dots, k_i$) be the set of state pairs satisfying $\gamma_{M_{i-1}}(s_j, true, a_i, s'_j) > 0$. Each pair consists of a state s'_j that can be reached from a non-decision state s_j by sending a request to a required port a_i . Then let M_i be the result N_{k_i} of the following recursion:
 - (a) Define $N_0 = Insert(M_{i-1}, s_0, s'_0, APKF(C_{c(a_i)})^*)$.
 - (b) $N_v = Insert(N_v, s_v, s'_v, APKF(C_{c(a_i)})^v)$
 where $APKF(C_{c(a_i)})^*$ and $APKF(C_{c(a_i)})^v$ are $APKF(C_{c(a_i)})$ with states renamed to be disjoint from the states of M_{i-1} and N_v respectively.

It can be shown that M_n does not contain any transitions involving actions other than τ .

² Other forms of PKF can be defined to give alternative estimations of how the black-box components behave.

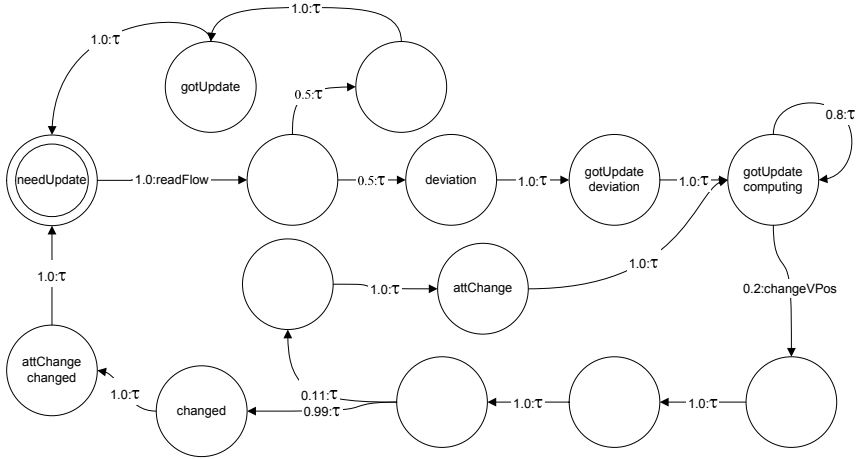


Fig. 8. Visual representation of $APKF(\text{SimpleController}, fv)$

Then,

$$APKF(A, m, c) = \langle \langle Z_{M_n}, A_{M_n}, \delta, initial_{M_n}, final_{M_n} \rangle, Prop_{M_n}, h_{M_n} \rangle$$

where δ is defined so that, for any $s_1, s_2 \in Z_{M_n}$, $a \in A_{M_n}$, $\delta(s_1, a, s_2) = \gamma(s_1, P, a, s_2)$ for some P .

Theorem 2. Given any grey-box component A with provided port m and context c , $APKF(A, m, c)$ is a well defined PKF.

Proof. Using the definition of behavioural profile, by induction on C . \square

Example 5. The architecture of Fig. 2 can be understood as an example of deployment. The SimpleController component, as a grey-box component, is deployed within a context fv defined

$$fv(\text{readFlow}) = \text{FlowReader} = \langle \text{readFlow}, true \text{ until } deviation \text{ steps: } 2 \text{ prob: } 0.5 \rangle$$

$$fv(\text{changeVPos})\text{Valve} = \langle \text{changeVPos}, true \text{ until } changed \text{ steps: } 6 \text{ prob: } 0.99 \rangle$$

The context should be understood as prescribing the required ports of SimpleController to be connected by a behavioural relation with the provided ports of FlowReader. The resulting average-case PKF for A in this context is depicted in Fig. 8.

4.4 Parametrized and Compositional Verification

The approach to verifying required timing constraints against a grey-box component C is then as follows.

1. Take a constraint P that is required to hold after invoking provided method p on C .
2. Choose a particular deployment environment D for the component.
3. Build $BPKF(C, p, D)$, $APKF(C, p, D)$ and $WPKF(C, p, D)$.
4. The constraint P can then be verified against these PKFs to give a picture of how the constraint applies to the system assuming the best, average and worst case timing behaviour for components in the environment.
5. The acceptance or rejection of the component as satisfying the constraint should be informed by these results, together with some domain-specific heuristic.

Example 6. We require that, during a call to `run`, `SimpleController` enables a change in valve position within 10 seconds of discovering a deviation in the required flow. It is acceptable if this timing requirement is met 99% of the time. This condition is formally specified as the PCTL statement:

true until deviation until attChange steps: 10 prob: .99 steps: 60 prob: 1

Assuming the deployment context described above, the statement is met by the best case time PKF, but is not satisfied by the average or worst case time PKFs. For this particular domain, we need at least average case time PKF to satisfy the requirement. The architect therefore needs to make a decision to either improve the timing of parts of the deployment context or else improve on the design of the grey-box component.

Compositional verification of components within a hierarchy is as follows. Take a grey-box component C whose provided ports P have all been accepted as satisfying a set of constraints $\{C_T\} T \in P$ with respect to a deployment context D within a particular architecture. The component can then be treated as a black-box component $\langle P, \{C_T\} \rangle$ and used as part of a new deployment context over which a grey-box component higher up in the process control hierarchy can be evaluated. This successive evaluation will not result in a state space explosion, as the previous deployment context is not used.

5 Related Work and Conclusions

Most work involving PCTL for specification of systems involves static verification of state transition designs. Younes and Simmons [9] consider probabilistic verification that is model independent. They used Continuous Stochastic Logic (CSL), which considers continuous time, in contrast to our use of PCTL with discrete time. They used acceptance sampling – the approach also used upper and lower bounds to decide if the formula truth value, which might could be a useful addition to our approach.

PCTL has been used by Jayaputera, Schmidt and the current authors [2], as a language for runtime verification of enterprise applications written in .NET [5]. However, that work was not compositional – the larger the architecture, the more difficult it became to do model checking.

Our approach to contractual specification complements previous work pioneered by Reussner [6] that has since evolved to become the Palladio component model [1]. That work also involves a parametrized approach to component reliability models, where a component's probabilistic finite state machine was determined according to a behavioural profile that was parametrized according to the behaviour of a deployment context. Markov chain analysis is used to arrive at a reliability measure for a component that depends on the context. Schmidt's group at Monash University has also developed this approach further with the idea of parametrized finite state machines, and had success in application of this to embedded systems [7,8].

It is envisaged that compositional methods similar to these will become a useful tool in coarse-grained process system design.

Our analysis becomes useful in the transition from design to implementation, when used in conjunction with model-driven approaches to synthesize appropriate monitoring infrastructure to check that our predictions hold in reality. This idea is currently being developed by the Predictable Assembly Laboratory³ at King's College London.

References

1. Becker, S., Koziolk, H., Reussner, R.: The palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82(1), 3–22 (2009)
2. Chan, K., Poernomo, I.H., Schmidt, H., Jayaputera, J.: A model-oriented framework for runtime monitoring of nonfunctional properties. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) *QoSA 2005 and SOQUA 2005*. LNCS, vol. 3712, pp. 38–52. Springer, Heidelberg (2005)
3. Genssler, T., Christoph, A., Schulz, B.: *PECOS in a nutshell* (2002)
4. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6(5), 512–535 (1994)
5. Poernomo, I., Schmidt, H.W., Jayaputera, J.: Verification and prediction of timed probabilistic properties over the dmtf cim. *International Journal of Cooperative Information Systems* 15(4), 633–658 (2006)
6. Reussner, R., Schmidt, H., Poernomo, I.: 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)
7. Schmidt, H.W., Krämer, B.J., Poernomo, I.H., Reussner, R.: Predictable component architectures using dependent finite state machines. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) *RISSEF 2002*. LNCS, vol. 2941, pp. 310–324. Springer, Heidelberg (2004)
8. Schmidt, H.W., Peake, I.D., Xie, J., Thomas, I., Krämer, B.J., Fay, A., Bort, P.: Modelling predictable component-based distributed control architectures. In: *Proc. Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pp. 339–346. IEEE, Los Alamitos (2004)
9. Younes, H., Simmons, R.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) *CAV 2002*. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)

³ <http://palab.dcs.kcl.ac.uk>

Timed Simulation of Extended AADL-Based Architecture Specifications with Timed Abstract State Machines

Stefan Björnander¹, Lars Grunske², and Kristina Lundqvist³

¹ School of IDE, Mälardalen University, Box 883, 72123 Västerås, Sweden
Tel.: +46 21 101 689

`stefan.bjornander@mdh.se`

² Faculty of ICT, Swinburne University of Technology Hawthorn,
VIC 3122, Australia
Tel.: +61 3 9214 5397

`lgrunske@swin.edu.au`

³ School of IDE, Mälardalen University, Box 883, 72123 Västerås, Sweden
Tel.: +46 21 101 428

`kristina.lundqvist@mdh.se`

Abstract. The Architecture Analysis and Design Language (AADL) is a popular language for architectural modeling and analysis of software intensive systems in application domains such as automotive, avionics, railway and medical systems. These systems often have stringent real-time requirements. This paper presents an extension to AADL's behavior model using time annotations in order to improve the evaluation of timing properties in AADL. The translational semantics of this extension is based on mappings to the Timed Abstract State Machines (TASM) language. As a result, timing analysis with timed simulation or timed model checking is possible. The translation is supported by an Eclipse-based plug-in and the approach is validated with a case study of an industrial production cell system.

Keywords: AADL, Behavior Annex, TASM, Translation.

1 Introduction

Time-critical embedded systems play a vital role in, e.g. aerospace, automotive, air traffic control, railway, and medical applications. Designing such systems is challenging, because the fulfillment of real-time requirements and resource constraints has to be proven in the development process. The architecture design phase is of specific practical interest, as the timing behavior and resource consumption of systems depend heavily on the architecture chosen for these systems. Furthermore, architectural mistakes that cause a system not to fulfill certain real-time requirements are hard to correct in later development phases. As a result, a development process for time-critical embedded systems should include verification techniques in the architecture design phase to provide evidence that a system architecture has the potential to fulfill its real-time requirements [1,2,3].

In this paper, timed simulations are used for evaluating real-time requirements. The Architecture Analysis and Design Language (AADL) [4] has been chosen, due to the sound specification language and its industrial use for the development of embedded systems in the automotive and avionic area. To allow for a specification of timing behavior in AADL, we have extended AADL's behavior annex [5] with time annotations that provide a minimum and maximum time for each behavioral transition in the model. To provide a formal semantics of these extensions, Timed Abstract State Machines (TASMs) [6] are used as the formal foundation. In detail, this paper describes a translational semantic that maps the extended behavior specifications of AADL into a network of timed abstract state machines. As a result, existing evaluation and verification techniques, such as timed simulations [7] and timed model checking [3] defined for the TASM specification formalism could also be applied to extended AADL specifications. The translation of time-extended AADL specifications into TASM is supported by a tool called AADLtoTASM and the overall approach is validated with a case study of an industrial production cell.

The rest of this paper is organized as follows: Section 2 introduces the running example of a production cell system and gives an overview of the used specification formalisms: AADL with its behavior annex as well as TASM. Section 3 presents the time extension of AADL's behavior annex. Furthermore, the translational semantics and tool support are also presented in this section. The results of timed simulations of the case study are presented in Section 4. Finally, in Section 5 the approach is compared with related work and Section 6 concludes the paper and gives an outlook to future work.

2 Background

2.1 The Production Cell - a Running Example

In order to explain and validate the approach of this paper, a running example of a production cell system is used. This case study is based on an automated manufacturing system which models an industrial plant in Karlsruhe (Germany). The industrial production cell system was first described by Lewerentz and Lindner in [8]. Ouimet et al. defined it in TASM in [9] as depicted in Figure 1.

The overall purpose of the system is to attach two bolts to a metal block. The system is not controlled by a central unit. Instead, the production cell components communicate with each other through ports and bus connections. The components work concurrently; when a component is ready to accept a new block it notifies the preceding component, which in turn acknowledges that it has loaded the block. There is also a signal acknowledging that the loading location of the production cell component is free.

The system is composed of the robot arms *Loader*, *BeltToPress* (Arm A), and *PressToBelt* (Arm B), the conveyer belts *FeedBelt* and *DepositBelt* as well as the *Press*. The system input are sets of blocks arriving in crates and the output are the same blocks with bolts attached to them. Once a block has been loaded, it is "dragged" through the system. See Figure 2 for a schematic description.

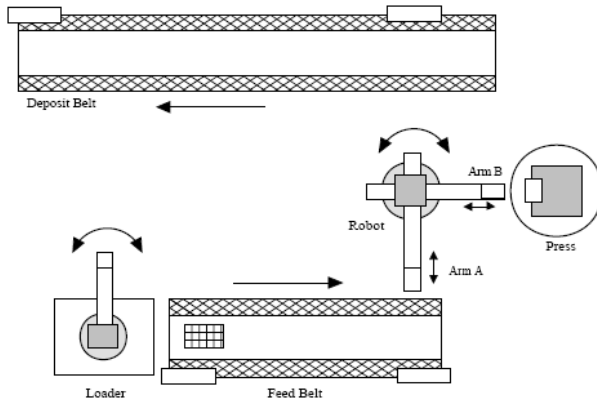


Fig. 1. The Production Cell System as presented in [9]

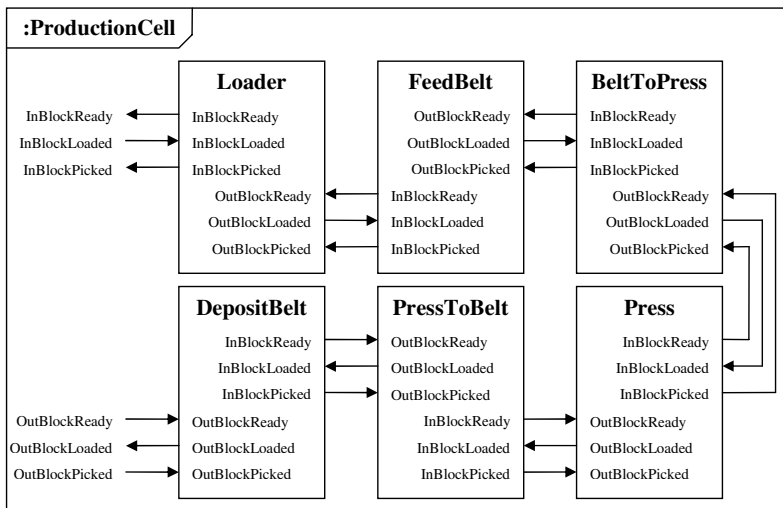


Fig. 2. Schematic Description of the Production Cell System

To describe the behavior of the production cell components let us look at the loader. While waiting, it is parked at the crate with the magnet turned off. When it receives a signal from the feed belt that it is ready to receive a new block, the loader turns the magnet on, moves the arm onto the beginning of the feed belt, turns the magnet off, and signals the feed belt that it has loaded a block. The rest of the production cell components work in similar fashions.

2.2 The Architecture Analysis and Design Language

AADL (aadl.info) is a language intended for the design of system hardware and software. It is a Society of Automotive Engineers (www.sae.org) standard based on MetaH [10] and UML 2.0 [11,12].

The AADL standard [4] includes runtime semantics for mechanisms of exchange and control of data, including message passing, event passing, synchronized access to shared data, thread scheduling protocols, and timing requirements.

AADL can be used to model and analyze systems already in use as well as to design new systems. AADL can also be used in the analysis of partially defined architectural patterns. Moreover, AADL supports the early prediction and analysis of critical system qualities, such as performance, schedulability, and reliability.

In AADL, a model consists of syntactical *elements*. There are three categories of elements. The first category is the application software:

- **Thread.** A thread can execute concurrently and be organized into thread groups.
- **Thread Group.** A thread group is a component abstraction for logically organizing threads or thread groups within a process.
- **Process.** A process is a protected address space whose boundaries are enforced at runtime.
- **Data.** A data component models types as well as static data.
- **Subprogram.** A subprogram models a callable block of source code.

The second category is the execution platform (the hardware):

- **Processor.** A processor schedules and executes threads.
- **Memory.** A memory component is used to store code and data.
- **Device.** A device represents sensors and actuators that interface with the external environment.
- **Bus.** A bus interconnects processors, memory, and devices.

The third category contains only one element: the system components. System components can consist of software and hardware components as well as other systems.

Component definitions are divided into *types* holding the public (visible to other components) features, and *implementations* that define the non-public parts of the component.

The components interact through defined interfaces, which consist of directional flows through event and data ports. It is possible to define physical port-to-port connections as well as logical flows through chains of ports.

Listing 1 shows the AADL ProductionCell system of section 2.1 (in AADL, two hyphens introduce a comment).

2.3 The Timed Abstract State Machines

TASM is specification language for reactive real-time systems. It has been developed at the Embedded Systems Laboratory (esl.mit.edu) at the Massachusetts Institute of Technology as a part of the Hi-Five project [13].

Formally, a TASM specification is a pair $\langle E, ASM \rangle$ [14] where E is the *environment*, which is a pair $E = \langle EV, TU \rangle$ where EV denotes the *environment variables* (a set of typed variables) and TU is the *type universe*, a set of types that includes real numbers, integer, boolean constants, and user-defined types.

Listing 1. The ProductionCell System

```

system ProductionCell
end ProductionCell;

system implementation ProductionCell.impl
  subcomponents
    loader :system Loader;
    feedBelt :system FeedBelt;
    robot :system Robot;
    press :system Press;
    depositBelt :system DepositBelt;
  connections
    event port feedBelt.FeedBeltReady -> loader.FeedBeltReady;
    event port loader.BlockDropped -> feedBelt.BlockDropped;
    event port feedBelt.BlockPicked -> loader.BlockPicked;
    — ...
end ProductionCell.impl;

```

ASM is the machine, which is a triple $\langle MV, CV, R \rangle$ where MV is the set of read-only *monitored variables*, CV is the set of *controlled variables* (both MC and CV are subsets of EV) and R is the set of *rules* (n, r) where n is a name and r is a rule of the form "if C then A " where C is an expression that evaluates to an element in BVU and A is an action. It is also possible to attach an else rule on the form "else A ".

Technically, a TASM specification is made up of *machines*. A specification must hold at least one *main machine* with its set of rules. Beside main machines, it is also possible to define sub machines and function machines. A sub machine accesses¹ the variables of the model and work as a procedure in a programming language, with the difference that it does not accept parameters. A function machine is equivalent to a function in a programming language; it accepts parameters and returns a value. However, it cannot access the global variables of the model.

A transition between two states in a TASM machine can be annotated by a time interval defining the minimal and maximal time to perform the transition.

The TASM Toolset [14] is an integrated development environment, composed of a project manager, an editor, an interpreter, and a simulator. The specification machines can be simulated in the Toolset.

3 Time Extension of the AADL Behavior Annex

3.1 The AADL Behavior Annex Extension

In order to increase the expressiveness of AADL, it is possible to add *annexes*. One of them is the Behavior Annex [15] that models an abstract state machine [16].

Each component of the model describes its logic by defining a behavior model, which consists of three parts [5]:

¹ The term *access* is a comprehensive term for *inspect* and *modify*.

- **States.** The states of the machine, one of them is the initial state.
- **Transitions.** The condition for a transition from one state to another (or the same state) is determined by a guard: an expression that evaluates to **true** or **false**. It is also possible to attach a set of actions to be executed when the transition is performed.
- **State Variables with Initializations.** The variables are similar to variables in programming languages. They can be initialized, inspected, and assigned.

In this paper, we add the concept of *time* to the transitions. In addition to the guard, each transition also has a time interval defining the minimal and maximal time for the transition performance. The grammar of the extended behavior model is given in Table 3, with the extensions underlined. The time annotations can be an interval, a single integer value representing both the lower and upper limit of an interval, or the word **null**, representing the absence of a time annotation.

Listing 2 defines the behavior model of the ProductionCell system that has been described earlier in Listing 1. The model is extended in relation to the standard in two ways:

- **Channel Initialization.** When the model starts to execute, a trigger message is sent to the OutBlockReady channel.
- **Time.** When the transitions occur, a time period is recorded. It is one time unit in all four transitions of Listing 2.

Listing 2. The Behavior Model of the ProductionCell System

```
annex ProductionCell {**
  state variables
    LoadedBlocks : integer;
    StoredBlocks : integer;
  initial
    OutBlockReady!;
    LoadedBlocks := 0;
    StoredBlocks := 0;
  states
    Waiting : initial state;
    Sending, Receiving : state;
  transitions
    Waiting -[(LoadedBlocks < 10) and InBlockReady?, 1]-> Sending
      {InBlockLoaded!;}
    Sending -[InBlockPicked?, 1]-> Waiting
      {LoadedBlocks := LoadedBlocks + 1;}
    Waiting -[OutBlockLoaded?, 1]-> Receiving
      {OutBlockPicked!;}
    Receiving -[true, 1]-> Waiting
      {OutBlockReady!; StoredBlocks := StoredBlocks + 1;}
**};
```

3.2 Translation Semantics

The translation to TASM has two phases: analysis and generation. The analysis phase follows the syntax of Table 3 to achieve a full coverage of the language.

The aim of this phase is to translate an extended AADL specification into a tuple consisting of a state variable map, a state variable set, a set of state variables to become initialized, the state set, the initial state (it can only be one), and the transaction set. The generation phase takes the tuple defined in the analysis phase as an input and returns for each transition (if present) a time specification and an if-statement testing the current state together with the guard expression and updating the new state value as well as the action list. See Tables 1 and 2 for pseudo code samples describing the analysis of an AADL behavior model and the generation of the corresponding TASM main machine, respectively.

In plain English the translation steps can be described as follows:

- For each model, its states are translated into an enumeration type that has the states as its possible values. Moreover, for each model, its states are also translated into a global variable of the enumeration type above. The variable is initialized with the enumeration value corresponding to the model's initial state. For the production cell system of Section 2.1, the generated types and variables is shown in Listing 3 (ProductionCellStateSet is the type and ProductionCellState is the variable).

Listing 3. The AADL behavior annex states of Listing 2 translated into TASM global type and variable.

```
ProductionCellStateSet := {Waiting, Sending, Receiving};
ProductionCellStateSet ProductionCellState := Waiting;
```

- Each state variable (please note the difference between state variables and states as described above) is translated to a global variable with the instance name attached to the variable name in order to avoid name clashes. The variables are initialized with the values given in the initial part of the model. Since a variable must be initialized in TASM, it must also be initialized in the model. Even though TASM supports the possibility to add variables local to a single machine, we do not use that option due to the fact that global variables are visible during simulation, local variables are not, consequently it would become more difficult to simulate the model with local variables. In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has two state variables LoadedBlocks and StoredBlocks, see Listing 2. They are translated into global TASM variables as shown in Listing 4.

Listing 4. The AADL behavior annex state variables of Listing 2 translated into TASM global variables.

```
Integer ProductionCell_LoadedBlocks := 0;
Integer ProductionCell_StoredBlocks := 0;
```

- For each connection between two subcomponent instance ports, a boolean variable representing the connection between the ports is defined. When a signal is sent through the output port, the boolean variable is set to true. The reception of the signal occurs when the boolean variable is read. After it is read, the variable is set to false, so that a new signal can be sent again. In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has several connections, see Listing 2. They are translated into global boolean TASM variables as shown in Listing 5 (in TASM, two slashes introduce a comment). The first boolean variable is initialized to true since the signal is triggered in Listing 2.

Listing 5. The AADL connections of Listing 1 translated into TASM boolean global variables.

```

Boolean Loader_InBlockReady_to_InBlockReady := true;
Boolean InBlockLoaded_to_Loader_InBlockLoaded := false;
Boolean Loader_InBlockPicked_to_InBlockPicked := false;
// ...

```

- For each subcomponent in the AADL system, a main machine with the subcomponent’s name is created in the TASM environment. Since the machines are generated from subcomponent instances, one component can be translated into several machines. Then the rules regarding time annotation are applied:
 - The timed feature of the transition is translated into the assignment of the predefined TASM variable t .
 - Each transition of the behavior models is translated into a TASM rule in its main machine. The rule is translated into an if-statement with an expression that is a logical conjunction between two subexpressions. The first subexpression tests whether the machine is in the source state, and the second expression is the guard of the transition (omitted if it consists solely of the value true). The actions of the rule is the action list connected to the transition. However, the first action is the assignment of the state variable to the target state. Moreover, if port signals are received in the guard, their matching boolean variables are set to false at the end of the action list.
 - A machine in TASM is active as long as one of its rule is satisfied, otherwise it becomes terminated. In order to keep the machine active even though no rule is satisfied, an else-rule in accordance with Section 2.3 is added.

In the production cell system of Section 2.1, the behavior annex of the ProductionCell system has several transitions, see Listing 2. They are translated into TASM rules as shown in Listing 6. The if-statement transacts the machine state from Waiting to Sending if the number of loaded blocks is less than ten and it receives the InBlockReady signal see Figure 2. Note that the signal is set to false, so that it can be received again in the future.

Listing 6. The AADL behavior model of the ProductionCell of Listing 2 translated into TASM rules.

```

R1:
{
  t := 1;
  if (ProductionCellState = Waiting) and
    ((ProductionCell_LoadedBlocks < 10) and
     Loader_InBlockReady_to_InBlockReady) then
    ProductionCellState := Sending;
    InBlockLoaded_to_Loader_InBlockLoaded := true;
    Loader_InBlockReady_to_InBlockReady := false;
}

// ...

R5:
{
  t := next;
  else then
    skip;
}

```

The main translation rule is that each AADL subcomponent is translated into a TASM instance. However, there is an additional rule. In many cases, when the subcomponents are translated into TASM machines, it required to establish an environment. Therefore, if a component has subcomponents and a behavior model of its own, it is assumed to hold the system's environment and initialization information, and an instance of the model is translated into a TASM main machine. The two rules fulfill different purposes. The main rule generates the components of the system, and the additional generates its initialization.

3.3 Tool Support

There is a number of tools developed for AADL. One of them is the Open Source AADL Tool Environment (OSATE, aadl.info), which is a plug-in for the Eclipse environment (www.eclipse.org).

AADLtoTASM², a contribution of this paper, is an OSATE plug-in that analyzes an AADL model and generates the equivalent TASM specification. It reads the subsystems, ports, and connections of each component as well as its behavior model. The model is parsed in accordance with the grammar of Table 3 with the extension of Section 3.1. Moreover, due to the fact that uninitialized variables are not allowed in TASM, all state variables in the AADL behavior model have to be initialized. In order to properly initialize the TASM variables representing the AADL behavior model states, each model must have exactly one initial state.

AADLtoTASM translates an AADL model extended with time annotations into a TASM model, in order for the model to be simulated in the TASM Toolset Simulation environment. However, the tool does not perform any analysis in addition to the translation.

² The tool is available for download, please contact one of the authors.

Table 1. AnalyseModelParts: $\text{Input} \Rightarrow \text{Map} \times \text{Set} \times \text{Set} \times \text{State} \times \text{Set}$

<code>/* Empty */ \Rightarrow return (emptyMap, emptySet, emptySet, null, transSet);</code>
<code>featureList name : type ; \Rightarrow (varMap, initSet, stateSet, initState, transSet) = AnalyseModelParts(featureList); return (varMap \cup (name, type), initSet, stateSet, initState, transSet);</code>
<code>featureList name ! ; \Rightarrow (varMap, initSet, stateSet, initState, transSet) = AnalyseModelParts(features); return (varMap \cup (name, send), stateSet, initState, transSet);</code>
<code>featureList name : state ; \Rightarrow (varMap, initSet, stateSet, initState, transSet) = AnalyseModelParts(features); return (varMap, initSet, stateSet \cup name, initState, transSet);</code>
<code>featureList name : initial state ; \Rightarrow (varMap, initSet, stateSet, -, transSet) = AnalyseModelParts(features); return (varMap, initSet, stateSet \cup name, name, transSet);</code>
<code>featureList source -[guard, time,] \rightarrow target { actionList } \Rightarrow (varMap, initSet, stateSet, initState, transSet) = AnalyseModelParts(features); return (varMap, initSet, stateSet, initState, transSet \cup (source, guard, time, target, actionList);</code>

Table 2. GenerateModelParts: $\text{Set} \Rightarrow \text{Output}$

<code>/* Empty */ \Rightarrow generate("");</code>
<code>transitionSet transition \Rightarrow GenerateModelParts(transitionSet); (if tr.time != null then generate("t := " + tr.time + ";;"); (generate("if (" + modelName + "State = " + tr.source + ") and (" + tr.guard + ") then"); (generate(" " + modelName + "State := " + tr.target + ";;"); (generate(" " + tr.actionList"); (generate(" " + extractPorts(tr.guard));</code>

4 Simulation

As the ProductionCell system of Section 2.1 has the subcomponents Loader, FeedBelt, BeltToPress, Press, PressToBelt, and DepositBelt, TASM main machines with the same names are generated for each of the subcomponents. Due to the additional rule of Section 3.2, an instance of the ProductionCell is translated into an TASM main machine with the same name. See Listing 3 for the definition of the states and state variables of the production cell system behavior model.

Furthermore, see Listing 2 for the behavior model of the ProductionCell. Extracts of the translated TASM machines are shown in Listings 3, 4, 5, and 6. The system that we want to investigate loads three (the number three is just chosen as an example) blocks into the system and picks three processed blocks from the system. Note, that the signals of input ports are received with the question mark (?) operator and the output ports signals are sent with the exclamation mark (!) operator. Even though it is possible to send data through the ports, only trigger signals are used in this case study. Each transition has a guard as well as a time interval.

In order to evaluate an architecture and find its minimal and maximal time, the production cell system was translated from AADL to TASM by the tool as described above. The minimal and maximal time for each production cell component of the system are defined in Table 4.

Table 3. The Extended Behavior Annex Grammar

<i>annex_specification</i>	⇒ <i>optional_state_variables optional_initialization optional_states optional_transitions</i>
<i>optional_state_variables</i>	⇒ state variables <i>variable_declaration_list</i> /* Empty. */
<i>variable_declaration_list</i>	⇒ <i>variable_declaration</i> <i>variable_declaration_list variable_declaration</i>
<i>variable_declaration</i>	⇒ <i>identifier_list</i> : <i>variable_type</i> ;
<i>identifier_list</i>	⇒ identifier <i>identifier_list</i> , identifier
<i>variable_type</i>	⇒ { <i>identifier_list</i> } integer boolean
<i>optional_initialization</i>	⇒ initial <i>initial_list</i> /* Empty. */
<i>initial_list</i>	⇒ <i>send</i> ; <i>assignment</i> ; <i>initial_list send</i> ; <i>initial_list assignment</i> ;
<i>send</i>	⇒ identifier ! ;
<i>assignment</i>	⇒ identifier := <i>expression</i>
<i>interval</i>	⇒ [integer_constant , integer_constant]
<i>optional_states</i>	⇒ states <i>state_list</i> /* Empty. */
<i>state_list</i>	⇒ <i>state</i> <i>state_list state</i>
<i>state</i>	⇒ <i>identifier_list</i> : <i>optional_initial state</i> ;
<i>optional_initial</i>	⇒ initial /* Empty. */
<i>optional_transitions</i>	⇒ transitions <i>transition_list</i> /* Empty. */
<i>transition_list</i>	⇒ <i>transition</i> <i>transition_list transition</i>
<i>transition</i>	⇒ identifier - [<i>expression</i> , <i>time</i>] → identifier <i>optional_action_list</i>
<i>time</i>	⇒ null integer_constant <i>interval</i>
<i>optional_action_list</i>	⇒ ; { } { <i>action_list</i> }
<i>action_list</i>	⇒ <i>expression</i> ; <i>action_list expression</i> ;
<i>expression</i>	⇒ <i>expression binary_operator expression</i> not expression (<i>expression</i>) <i>constant</i> <i>assignment</i> <i>send</i> identifier identifier ?
<i>binary_operator</i>	⇒ or and + - * / = != < <= > >=
<i>constant</i>	⇒ integer_constant boolean_constant real_constant character_constant string_constant

Table 4. Transition Time

System	Transition	Minimum Time	Maximum Time
ProductionCell	Sending	1	1
	Receiving	1	1
	Waiting	1	1
Loader BeltToPress PressToBelt	Magnet On	1	1
	Rotating Arm Forwards	3	6
	Magnet Off	1	1
FeedBelt DepositBelt	Rotating Arm Backward	3	6
	Receiving Block	1	1
	Moving Block	4	8
Press	Leaving Block	1	1
	Move Block to Press Position	1	2
	Pressing	5	10
	Move Block from Press Position	1	2

Furthermore, the process was simulated one hundred times with random time intervals. Figure 3 describes the results of this simulations. It is also possible to analyze the best and worst case performance by setting the simulator to use the minimum or maximum time, respectively, of each interval. With this method, we could conclude that the minimum time to put three blocks through the system was 72 units and the maximum time was 131 units.

The TASM Toolset does also have the capability to perform a number of different kinds of analysis, such as average time consumption and minimal and maximal time consumption regarding a specific rule. However, due to limited

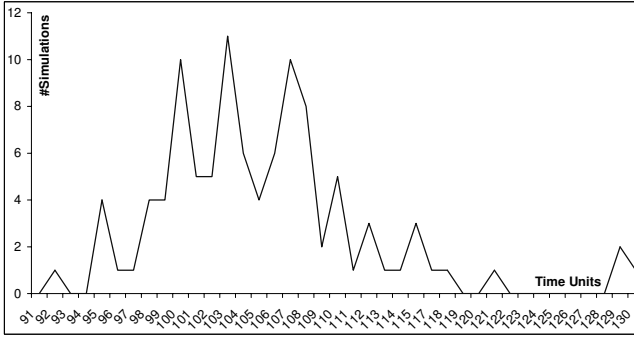


Fig. 3. Simulation of the Generated TASM Model

space, we do not describe them in detail in this paper. For further timing analysis of TASM we refer to [14].

The simulation of the case study shows based on the AADL behavior annex extension and the translator AADLtoTASM automatic reasoning about timing behavior of AADL models is possible. As the simulation of the machines are executed in parallel, it would be difficult to perform such simulations manually. Another benefit is that the TASM model can be further translated into a UPPAAL [17] model, where timed model checking can be performed.

5 Related Work

Due to the importance of real-time systems, a considerable number of languages have been formally extended to tackle the problem of verifying real-time requirements and properties. Consequently, in the following we would like to compare the specific contributions of this paper, namely (a) the extension of the AADL notation and its behavior model with timing annotation and (b) the transition of timed AADL models into timed abstract state machines to allow for tool automated simulation of the model with related approaches.

A majority of the related approaches focuses on timed extensions of visual specification formalisms. Known examples are timed Automata [18,19], Timed Petri Nets [20] and Timed Behavior Trees [21,22,23]. Timed specification formalisms come with a variety of tools and methods to correctly specify a system and to verify its timing requirements and properties. An example of approaches that help with the correct specification is based on the recently proposed timed patterns [24]. The verification focuses on timed model checking [1] with tools like UPPAAL [17] and KRONOS [25].

Since the use of these model checking tools is also possible for the timed AADL extension as TASM has been chosen as the underlying formal specification formalism in the presented approach. In [26], a formal transition from TASM to timed automata, especially UPPAAL automata, is introduced.

There are also textual notations that have timed extensions, e.g. timed CSP [27], and timed versions of Object-Z [28,29]. However, the specification of a system in these languages requires expert knowledge in formal methods and practitioners are often discouraged by the strict mathematical formalism. In contrast, the foundation for the approach presented in this paper is with AADL, a well accepted specification language [4]. Furthermore, we argue that the introduced concepts and syntax elements are easy to understand and do not require a massive amount of training for the practitioners that are already familiar with AADL and its behavior model.

Beside the timed extension described in this paper and the use of timed simulations of the underlying TASM model, AADL currently also supports scheduling analysis [30,31] as a second type of real-time analysis. This scheduling analysis assumes different scheduling strategies (e.g. rate monotonic scheduling) and allows verifying schedulability and end-to-end deadlines. The approach described in this papers is based on a fully concurrent implementation of the architectural elements. Consequently, both approaches are complementary, but we believe that an integration of the two approaches is an interesting topic for future research.

6 Conclusions and Further Work

The main contribution of this paper is an extension of AADL's behavior model to allow for the specification of timed behavior. This extension is based on the formal language of timed abstract state machines (TASM) and consequently techniques like time simulations could be performed to check if an architecture specification meets its real-time requirements and resource constraints.

This paper has furthermore presented a novel tool called AADLtoTASM for transition from AADL with its behavior annex to TASM. OSATE is an Eclipse plug-in and the tool is an OSATE plug-in that reads an AADL file (including its behavior models) and generates the corresponding TASM file.

To evaluate the approach, the production cell case study has been translated from AADL into TASM. This shows that the tool works and that it seems to be valuable when reasoning about AADL models.

There are some features that can be changed in the future. One of them is the stopping criteria for the simulation could be improved. One possible approach could be confidence intervals.

One possible extension of this work is to further translate the TASM model into UPPAAL [17] in order to perform timed model checking. In that case, it would be possible to, in detail, formally define best-case and worst-case time behavior of the AADL model.

Furthermore, an extension to specify probabilistic behavior in AADL's behavior annex, jointly with the specification of the AADL Error Annex [32], would be interesting. As a result also the probabilistic behavior (e.g. probabilistic safety properties [33]) could be analyzed. As an example, Monte Carlo simulation could be performed on these models.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 414–425. IEEE Computer Society Press, Los Alamitos (1990)
2. Grunske, L.: Early quality prediction of component-based systems - A generic framework. *Journal of Systems and Software* 80, 678–686 (2007)
3. Yovine, S.: Model checking timed automata. In: Rozenberg, G. (ed.) *EEF School 1996*. LNCS, vol. 1494, pp. 114–124. Springer, Heidelberg (1998)
4. Feiler, P.H., Gluch, D.P., Hudak, J.J.: *The Architecture Analysis and Design Language (AADL): An Introduction*. Technical Report CMU/SEI-2006-TN-011, Society of Automotive Engineers (2006)
5. Feiler, P., Lewis, B.: *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1*. Technical Report AS5506/1, Society of Automobile Engineers (2006)
6. Ouimet, M., Lundqvist, K.: The TASM Toolset: Specification, Simulation, and Formal Verification of Real-Time Systems. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 126–130. Springer, Heidelberg (2007)
7. Lynch, N.: Modeling and verification of automated transit systems, using timed automata, invariants, and simulations. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) *HS 1995*. LNCS, vol. 1066, pp. 449–459. Springer, Heidelberg (1996)
8. Lewerentz, C., Lindner, T.: Formal development of reactive systems, case study production cell. In: Lewerentz, C., Lindner, T. (eds.) *Formal Development of Reactive Systems*. LNCS, vol. 891, pp. 21–54. Springer, Heidelberg (1995)
9. Ouimet, M., Lundqvist, K.: *Modeling the Production Cell System in the TASM Language*. Technical Report ESL-TIK-00209, Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA (2007)
10. Vestal, S.: Formal verification of the metaH executive using linear hybrid automata. In: *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS 2000)*, pp. 134–144. IEEE, Washington (2000)
11. Miles, R., Hamilton, K.: *Learning UML 2.0*. O’Reilly Media, Sebastopol (2006)
12. Pilone, D., Pitman, N.: *UML 2.0 in a Nutshell, 2nd edn*. O’Reilly Media, Sebastopol (2005)
13. Ouimet, M., Lundqvist, K.: The TASM Language and the Hi-Five Framework: Specification, Validation, and Verification of Embedded Real-Time Systems. In: *Asia-Pacific Software Engineering Conference* (2007)
14. Ouimet, M., Lundqvist, K.: *The TASM Language Reference Manual, Version 1.1*, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA (2006)
15. França, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: *ICECCS*, pp. 377–382. IEEE Computer Society, Los Alamitos (2007)
16. Börger, E., Stärk, R.: *Abstract State Machines - A Method for High-level System Design and Analysis*. Springer, Heidelberg (2003)
17. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) *SFM-RT 2004*. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
18. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* 126, 183–235 (1994)
19. Bengtsson, J., Wang, Y.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets*. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)

20. Winkowski, J.: Processes of Timed Petri Nets. *TCS: Theoretical Computer Science* 243 (2000)
21. Colvin, R., Grunske, L., Winter, K.: Probabilistic timed behavior trees. In: Davies, J., Gibbons, J. (eds.) *IFM 2007*. LNCS, vol. 4591, pp. 156–175. Springer, Heidelberg (2007)
22. Colvin, R., Grunske, L., Winter, K.: Timed behavior trees for failure mode and effects analysis of time-critical systems. *Journal of Systems and Software* 81, 2163–2182 (2008)
23. Grunske, L., Winter, K., Colvin, R.: Timed Behavior Trees and their Application to Verifying Real-time Systems. In: *Proceedings of the 18th Australian Conference on Software Engineering (ASWEC 2007)*, pp. 211–220. IEEE Computer Society, Los Alamitos (2007)
24. Dong, J.S., Hao, P., Qin, S.C., Sun, J., Wang, Y.: Timed patterns: Tcoz to timed automata. In: Davies, J., Schulte, W., Barnett, M. (eds.) *ICFEM 2004*. LNCS, vol. 3308, pp. 483–498. Springer, Heidelberg (2004)
25. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The Tool KRONOS. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) *HS 1995*. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)
26. Ouimet, M., Lundqvist, K.: A Mapping between the Timed Abstract State Machine Language and UPPAAL's Timed Automata. Technical Report ESL-TIK-00212, Embedded Systems Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA (2007)
27. Schneider, S.: An operational semantics for timed CSP. *Information and Computation* 116, 193–213 (1995)
28. Dong, J.S., Duke, R., Hao, P.: Integrating object-z with timed automata. In: *Int. Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pp. 488–497. IEEE Computer Society, Los Alamitos (2005)
29. Smith, G., Hayes, I.: An introduction to real-time object-z. *Formal Aspects of Computing* 13, 128–141 (2002)
30. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Embedded Systems Architecture Analysis Using SAE AADL. Technical report, CMU/SEI-2004-TN-005 (2004)
31. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with AADL. *ACM SIGADA Ada Letters* 25, 1–10 (2005)
32. Feiler, P., Rugina, A.: Dependability modeling with the architecture analysis and design language (AADL). Technical Report CMU/SEI-2007-TN-043, Carnegie Mellon University (2007)
33. Grunske, L., Han, J.: A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models. In: *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008*, pp. 283–292. IEEE Computer Society, Los Alamitos (2008)

Achieving Agility through Architecture Visibility

Carl Hinsman¹, Neeraj Sangal², and Judith Stafford³

¹ L.L.Bean, Inc.

Freeport, Maine USA

chinsman@llbean.com

² Lattix, Inc.

Andover, Massachusetts USA

neeraj.sangal@lattix.com

³ Tufts University

Medford, Massachusetts USA

jas@cs.tufts.edu

Abstract. L.L.Bean is a large retail organization whose development processes must be agile in order to allow rapid enhancement and maintenance of its technology infrastructure. Over the past decade L.L.Bean's software code-base had become brittle and difficult to evolve. An effort was launched to identify and develop new approaches to software development that would enable ongoing agility to support the ever-increasing demands of a successful business. This paper recounts L.L.Bean's effort in restructuring its code-base and adoption of process improvements that support an architecture-based agile approach to development, governance, and maintenance. Unlike traditional refactoring, this effort was guided by an architectural blueprint that was created in a Dependency Structure Matrix where the refactoring was first prototyped before being applied to the actual code base.

Keywords: architecture, dependency, agility.

1 Introduction

This paper reports on L.L. Bean, Inc.'s experience in infusing new life to its evolving software systems through the increased visibility into its system's architecture through the extraction and abstraction of code dependencies. Over years of software development the information technology infrastructure at L.L. Bean had become difficult to maintain and evolve. It has long been claimed that visibility of architectural dependencies could help an organization in L.L. Bean's position [9][12][17]. In this paper we provide support for these claims and demonstrate the value of applying these emerging technologies to a large, commercial code-base. We explore the strengths and weaknesses of the application of the Dependence Structure Matrix (DSM) as implemented in the Lattix LDM [16], to improve the agility of the L.L. Bean code base, and propose avenues for follow-on research to further improve tool support for architecture-based refactoring in support of software agility.

L.L.Bean has been a trusted source for quality apparel, reliable outdoor equipment and expert advice for nearly 100 years¹. L.L.Bean's software is used to manage its sales, which include retail, mail-order catalog, as well as on-line sales, inventory, and human resources. More than a 100 architects, engineers, and developers work on continual improvement and enhancement of the company's information technology infrastructure, which for the last 8 years, has suffered the typical problems associated with rapid evolution such as increased fragility and decreased intellectual control resulting in increased difficulty in building the system. While the company's software development processes have long included a number of good practices and coding rules to help avoid these issues, in the end the speed of evolution overwhelmed the development teams and maintenance and evolution of the software infrastructure were recognized as chief concerns by upper management of the company. Investigation into the core cause of the problems pointed to the fact that the code had become a complex entanglement of interdependencies. It was decided that the code must be restructured and software engineering process must be enhanced to prevent the web of dependencies from recurring in the future.

Refactoring, as described by Fowler et al. [4] and others in the object community is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Refactoring is generally practiced as a series of small changes to classes and collections of classes that modify the structure in a way that improves the code or moves it in an intended direction so that the code is better structured for future enhancements, improved comprehensibility, easier unit testing etc. There are a number of tools that provide support for refactoring (e.g. Eclipse and IntelliJ). These tools provide a variety of capabilities such as generating 'getters/setters' and 'constructors' etc that simplify code modifications. However, the approach of localized modifications was too limited and L.L.Bean recognized the need to approach solving the problem from a global perspective.

Because the software architecture of a system is a model of software elements and their interconnections that provides a global view of the system it allows an organization to maintain intellectual control over the software and provides support for communication among stakeholders [2][18]. As such, it seemed an architectural approach to "refactoring" L.L.Bean's code base would be appropriate. A variety of approaches for exploring architectural visibility were explored. Ultimately, an approach based on a Dependency Structure Matrix (DSM) [15] representation was selected because of its innate ability to scale and the ease with which alternative architectural organizations could be explored.

L.L.Bean's strategic approach to refactoring required few code changes but rather a code consolidation followed by a series of structural modifications. Unlike current approaches to refactoring, L.L.Bean's method is driven by overall visibility of the architecture and includes five steps: define the problem, visualize the current architecture, model the desired architecture in terms of current elements, consolidate and repackage the code base, and automate governance of the architecture through continuous integration.

This approach to software development employs the Lattix Architecture Management System as the primary tool for architectural analysis and management. It

¹ <http://www.llbean.com>

also uses custom tools developed at L.L. Bean for automating changes to the code organization, and for maintaining visibility of the state of evolving dependencies on a continuing basis.

The remainder of the paper recounts the L.L. Bean experience in “refactoring” and describes the architecture-based approach to software development that has been created and adopted as an organizational standard at L.L. Bean. The new standard was welcomed by all development teams and provides a mechanism for continuous improvement as the technology infrastructure evolves to meet ever-growing business demands of this increasingly popular brand.

We begin our report with a description of the problem facing L.L. Bean’s software developers. This is followed by a recounting of research toward identifying a viable solution and the basis for the decision to apply an approach based on a Dependency Structure Matrix. We then provide an overview of this approach in enough detail to support the reader’s understanding of this report, and follow that with description of our experience using and extending the Lattix tools at L.L. Bean. We then summarize lessons learned through this experience and propose avenues for future research in providing additional mechanisms to maintain architecture-based agility.

2 Background

2.1 IT Infrastructure

A significant part of L.L. Bean’s information technology infrastructure is written in Java and runs on Windows, UNIX, or Linux based servers. The system has gone through a rapid evolution over the last eight years due to several massive development efforts undertaken in response to increased demand from multiple business units. New front-end systems, strategic web site updates, regular infrastructure improvements, external product integration, and security have been among the key drivers.

L.L. Bean develops software primarily in Java and follows object oriented programming principles and patterns [5]. Development teams normally consist of ten or fewer developers grouped by business domains such as product, order capture, human resources, and IT infrastructure. Package names are chosen to represent behavior and/or responsibility of groups of Java classes within the package. Aligning development structure with naming conventions facilitates reuse and helps avoid duplication of effort by increasing visibility. Although it does not address interdependencies among modules, this alignment was an important contributor to the success of this project.

The current system has more than one million lines of code assembled into more than a 100 jar files. In turn, the actual code is organized into nearly 1,000 Java packages and more than 3,000 Java classes. Despite the use of good software development practices and standards, normal code evolution created a complex entanglement of interdependencies, increasing software development and maintenance costs, and decreasing reuse potential. Multiple code bases diverged over time, which increased complexity significantly.

Initially, ad-hoc approaches were tried to deal with these problems. Interdependency issues were mainly identified by configuration managers while

attempting to compile and assemble applications for deployment. These were then fixed one dependency entanglement at a time. The process was slow and correcting one problem often led to a different set of problems. One significant effort for resolving core dependency entanglements consumed three man weeks of effort and was not entirely successful. Moreover, there was nothing to prevent entanglements from recurring.

Business needs continued to drive new development, and interdependency entanglements continued to grow. Software development and configuration management costs increased in stride. IT management understood the economic significance of reuse and created a small team of software engineers focused on creating and implementing a comprehensive packaging and reuse strategy. This team quickly identified the following key issues:

- Too many interdependencies increased testing and maintenance costs
- Multiple Code Bases (segmented somewhat by channel) resulted from the rapid evolution and could not be merged. A goal of the effort was to consolidate into a single code base and transition the development life cycle to a producer/consumer paradigm.
- Architecture was not visible and no person or group in the organization maintained intellectual control over the software
- There was no mechanism to govern system evolution

A solution was needed that supported immediate needs while providing the framework for refactoring the architecture to prevent costly entanglements from recurring.

2.2 Preliminary Research and Tool Selection

There were two key tasks. First, research the abstract nature of software packaging from various viewpoints. Second, create a clear and detailed understanding of the existing static dependencies in L.L. Bean's Java code base. What dependencies actually existed? Were there patterns to these dependencies?

In addition to the major goals of eliminating undesirable dependencies and governing packaging of future software development, the resulting packaging structure needed to accomplish other goals. First, provide a single, consolidated code base to support a producer/consumer paradigm (where development teams consume compiled code instead of merging source code into their development streams) while helping to define code ownership and responsibility. Next, dynamically generate a view of the interdependencies of deliverable software assets. Last, minimize the cost and effort required to compile and assemble deliverable software assets. An unstated goal held by the team was to increase the level of reuse by fostering a Java development community and increase communication between development teams.

It was important to build confidence in the new strategy. The business, and more specifically the development teams supporting the various business domains, would not entertain undertaking a restructuring effort without evidence of the soundness of the strategy. The team understood that the way to earn this trust was through research, communication and prototyping.

Literature Search

As a starting point, the team sought articles and academic papers primarily through the Internet. Managing dependencies is not a new problem, and considerable research and analysis on a wide range of concepts and approaches was available to the strategy development team [3][6][7][10][11][14][15][17]. Another effort was underway at L.L.Bean to create a strategy for reuse metrics; there was overlap between these two efforts [7][8][13][14]. Much of the research suggested that code packaging in domain-oriented software could promote reuse and facilitate metrics. Exploring these metrics, and tools to support them, provided additional focus. Transition of research into practice would happen more quickly with increased effort on both sides to bridge the researcher/practitioner communication chasm.

Analysis Tools

There are many tools available for detecting and modeling dependencies in Java. The team wanted to find the most comprehensive and easy to understand tool. Initially, open-source tools were selected for evaluation. These included Dependency Finder² and JDepend³ (output visualized through Graphviz⁴), among others^{5,6}. Each of these tools were useful in understanding the state of dependencies, but none of them offered the comprehensive, easy to understand, global view needed nor did they provide support for restructuring or communication among various stakeholders, which included IT managers, architects, and developers.

Graphing the analysis was cumbersome, requiring the use of a combination of tools to produce illustrations of problem spaces. One solution was to use JDepend to analyze the code base, which outputs XML. This output was then transformed into the format required by Graphviz for generating directed graphs. The process was computationally intensive, and there was a limit to the amount of code that could be analyzed collectively in this fashion.. Furthermore, when this view was generated it was nearly incomprehensible and of little practical value in either communicating or managing the architecture. Using these tools was inefficient and less effective than had been anticipated. After a problem was identified, it was necessary to code or compile a potential solution and then repeat the entire analysis to illustrate the real impact. Given the extent of the interdependency entanglement, identifying and fixing problems through this approach was found to be too cumbersome to be practical.

L.L.Bean's research identified the Lattix matrix-based dependency analysis tool as promising and, through experience, found it to be effective in that it offered a comprehensive easy to understand interface as well as mechanisms for prototyping and applying architecture rules, and supporting "what if" analysis without code modification.

² <http://depfind.sourceforge.net/>

³ <http://clarkware.com/software/JDepend.html>

⁴ <http://www.graphviz.org/>

⁵ <http://java-source.net/open-source/code-analyzers/byecycle>

⁶ <http://java-source.net/open-source/code-analyzers/classycle>

The Lattix Architecture Management System

Lattix has pioneered an approach using system interdependencies to create an accurate blueprint of software applications, databases and systems. To build the initial Lattix model, the LDM tool is pointed at a set of Java jar files. Within minutes, the tool creates a “dependency structure matrix” (DSM)⁷ that shows the static dependencies in the code base. Lattix generated DSMs have a hierarchical structure, where the default hierarchy reflects the jar and the package structure.

This approach to visualization also overcomes the scaling problems that L.L.Bean encountered with directed graphs. Furthermore, Lattix allows users to edit system structures to run what-if scenarios and to specify design rules to formalize, communicate, and enforce architectural constraints. This means that an alternate structure, which represents the desired architectural intent, can be manipulated and maintained even if the code structure is not immediately a true reflection. Once an architecture is specified Lattix allows that architecture be monitored in batch mode and key stakeholders are notified of the results.

The Lattix DSM also offers partitioning algorithms to group and re-order subsystems. The result of this analysis shows the layering of the subsystems as well as the grouping of subsystems that are coupled through cyclic dependencies.

3 Refactoring the Architecture

With tool support and good development practices in place, L.L.Bean created a five-step approach to architecture-based maintenance that increased the agility of our software development process.

STEP 1: Mapping the Initial State

The first step in the architecture-based refactoring process was to illuminate the state of the code base. An initial DSM was created by loading all Java jars into a Lattix LDM. Then the subsystems in the DSM were organized into layers [1]. The magnitude of the problem became apparent once this DSM was created. It was possible to see numerous undesirable dependencies where application code was being referenced by frameworks and utility functions. The example shown in Fig. 1 illustrates a highly complex and cyclic dependency grouping.

A DSM is a square matrix with each subsystem being represented by a row and column. The rows and columns are numbered for ease of reference and to reduce clutter. The results of DSM partitioning, the goal of which is to group subsystems together in layers, can be evidenced by the lower triangular nature of the upper left-hand portion of the matrix shown in Fig. 1. Each layer in turn is composed of subsystems that are either strongly connected or independent of each other. In this figure, the presence of dependencies above the diagonal in the lower right-hand grouping shows us that subsystems 30..37 are circularly connected. For instance, if you look down column 31, you will see that subsystem 31 depends on subsystem 30 with strength of ‘3’. Going further down column 31, we also note that subsystem 31 depends on subsystems 33 and 34 with strengths of ‘16’ and ‘85’, respectively. By

⁷ <http://www.dsmweb.org>

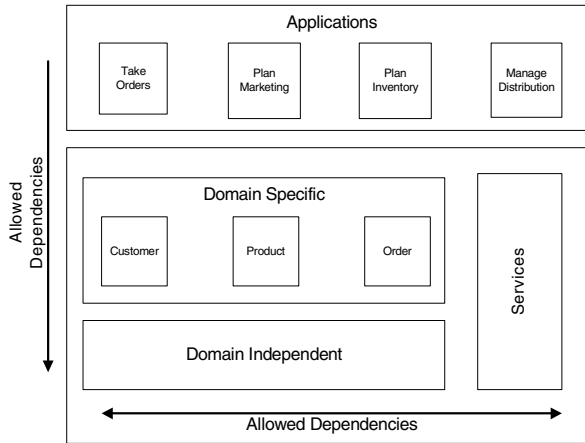


Fig. 2. Example Layered Architecture

STEP 3: Validating the Architecture

The next step was prototyping and transforming the current state into the intended architecture.

First, subsystem containers were created at the root level of the model for each of the high-level organizational layers defined in the architecture; initially these were the familiar domain-independent, domain-specific, and application-specific layers. The next step was to examine each jar file, and move them into one of the defined layers.

Here, the benefits of L.L.Bean's early package naming approach became clear; behavior and responsibility were built into package naming clarifying the appropriate layer in most cases. In a small set of specialized cases, developers with in-depth knowledge of the code were consulted. Here again, the well defined and documented layered architecture facilitated communication with software engineers and simplified the process of deciding on the appropriate layer. With the architecture already well understood, it took only two working days to successfully transform the initial model into the intended architecture, simply by prototypically moving Java classes to their appropriate package according to both their generality/specificity and their behavior/responsibility. At the end of that time, we were surprised to observe that nearly all undesirable dependencies at the top level had been eliminated. The DSM shown in Fig. 3 captures the state of the prototyping model near the end of the two-day session. The lower triangular nature of the DSM shows the absence of any top-level cycle.

		application	domain	commons
\$root		1	2	3
+ application	1	26%		
+ domain	2	469	35%	
+ commons	3	14182360	38%	

Fig. 3. DSM of layered architecture, no top-level cycles present

STEP 4: Identifying Sources of Architecture Violation

Three key packaging anti-patterns were identified that were at the core of the interdependency entanglement. This is illustrated by the following examples (note: the arrows in the figures show “uses” dependencies [2]):

Misplaced Common Types: Many types (i.e. Value Object Pattern, Data Transfer Object Pattern, etc.) were packaged at the same hierarchical level as the session layer (Session Façade) to which they related. This approach widely scattered dependencies creating a high degree of complexity, and a high number of cyclic dependencies. This issue was resolved as shown in Fig. 4, by moving many of these common types from their current package to an appropriate lower layer. This resolved an astounding 75% of existing circular dependencies.

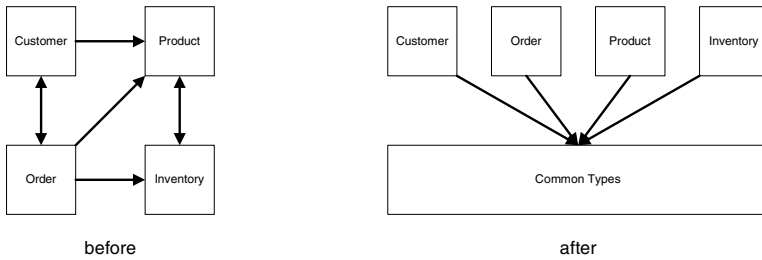


Fig. 4. Repackaging Common Types

Misplaced Inheritable Concrete Class: When a concrete class is created by extending an abstract class, it is packaged according to its behavior. However, when a new concrete class is created by extending this class it then creates a coupling between components that were normally expected to be independent. Moving the concrete class to the shared layer where its parent was located solved the problem as

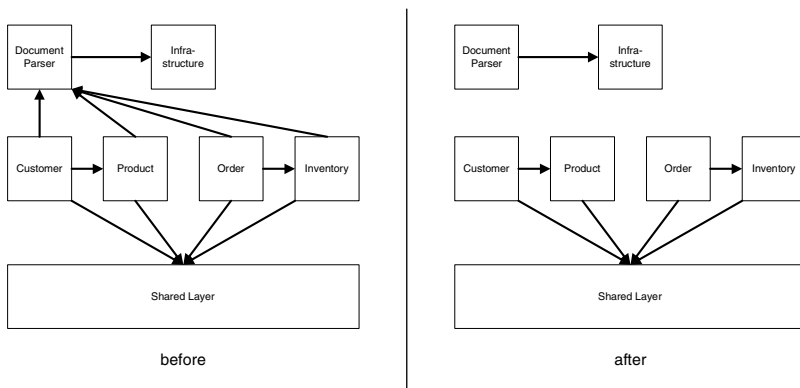


Fig. 5. Moving Descendants into Ancestor's Package

shown in Fig. 5. This also supports the notion that concrete classes should not be extended [19]. Instead, whenever the need arises to extend a concrete class, the code should be refactored to create a new abstract class, which is then used as a base class for the different concrete classes. This problem also illustrates that as code bases evolve it is necessary to continuously analyze and restructure the code.

Catchall Subsystems: The behavior/responsibility focus of the early package naming approach produced a subsystem dedicated to IT infrastructure. This became a disproportionately large “catch-all” subsystem that represented a broad array of mostly unrelated concepts. It also included a small set of highly used classes supporting L.L.Bean exception handling. It generated a large number of dependencies making it brittle and costly to maintain. To manage this problem, the exception classes were moved into their own subsystem and the remaining parts were reorganized into multiple subsystems of related concepts as shown in Fig. 6. This problem also illustrates how analyzing usage can be used to identify and group reusable assets.

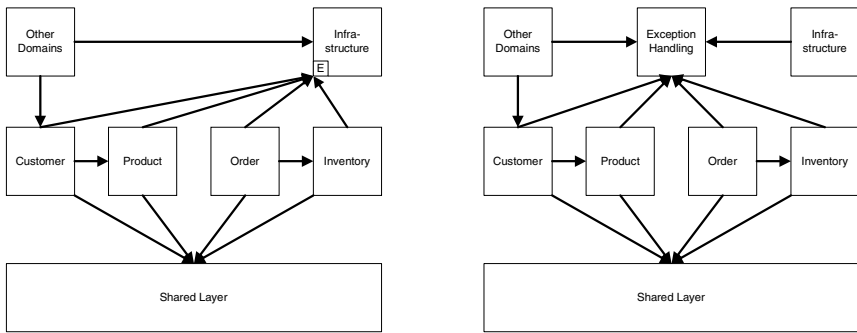


Fig. 6. Breaking up Catchall Subsystems

STEP 5: Refactoring the Code

With the right tools and a well-defined architecture, prototyping packaging change was relatively simple. Fortunately, L.L. Bean’s code restructuring effort was primarily limited to changing packages (e.g. Java package and import statements), and did not affect code at a method level.

The initial goal was to take a “big bang” approach by re-architecting the entire system at once. The large commitment to cut over to a consolidated and restructured code base in one step proved costly and did not mesh with the various iterative development cycles across development teams. Instead, an incremental approach is being used where new development and refactored code are packaged according to the principles of the layered architecture. Development teams and configuration engineers use DSM models to analyze static dependencies as well as to prototype new packages and package changes during the normal development cycle. This has proven to be a sustainable approach for continuous improvement of the code base.

A few key standards in place at L.L. Bean have helped facilitate this approach. First, using a standard IDE, developers can easily organize import statements such that fully qualified class names are not embedded within methods. Second, wildcards

are not allowed in import statements. Automated configuration management processes enforce standards⁸. As a result, there existed a relatively consistent state of package and import statements. The last important standard here is unit tests. L.L. Bean standards require a unit tests for every class, and most software development teams employ test-first development methodologies⁹. After restructuring, each unit test is exercised, providing an immediate window into the impact of the changes.

4 Evolving and Improving the Architecture

A software engineering process was needed that would prevent architectural drift and the need for large scale refactoring in the future. A set of rules were created that could be applied to any of L.L. Bean's DSM models, and visibility of maintenance processes was increased.

4.1 Rules

Design rules are the cornerstone of architecture management. L.L. Bean developed a simple set of architecture enforcement rules. These rules enforce a layered architecture and essentially state that members of a given layer may only depend on other members in the same level, or in layers below it.

Rules also help software engineers identify reuse candidates. When violations occur, the nature of the dependencies and the specific behavior of the Java code are closely analyzed. If there are multiple dependencies on a single resource that break an allowed dependency rule, then the target resource is a candidate for repackaging. The analysis is followed by a discussion with the appropriate project manager, architect or software developer.

Governance reduces software maintenance cost, improves quality, and increases agility, by enabling architectural remediation during ongoing development.

4.2 Maintaining Visibility

Architectural governance offers several benefits. A DSM model provides consistent visibility and supports on-going communication between development teams, configuration engineers and project leaders. It also facilitates change impact analysis. L.L.Bean creates DSM models at different organizational levels from application-specific to a comprehensive "master model". Application modeling during normal development cycles enables configuration engineers to determine what dependencies are missing, what dependencies are using an outdated version, whether unused component libraries that are included should be removed, and to report on changes between development iterations. As of the writing of this paper, this analysis and on-going communication have resulted in a 10% reduction in the number of Java jar files being versioned and dramatically improved understanding about the true dependencies of the applications and the jars they consume.

⁸ <http://pmd.sourceforge.net/>

⁹ <http://www.junit.org/index.htm>

L.L.Bean creates multiple dependency structure matrices for various purposes. One is designated as “master model”, which maintains visibility to the current state of the overall architecture as new development continually introduces new dependencies. The master model is integrated with and updated through automated configuration management processes, and is designed to support dependency governance. Each time a new version of a software element is created, the master model is updated, design rules are applied and when violations are detected, they are reported to the appropriate stakeholders (project managers, configuration managers, architects, and reuse engineers) who determine whether each violation is a programming error or reflect change in architectural intent. Violations also “break the build”, forcing software development teams to correct problems before the new version is permitted to move forward in its lifecycle.

For additional analysis, L.L.Bean created an analysis and configuration tool leveraging DSM metadata designed to address two long-standing questions. First, given a class, which jar file contains that class? Second, given a jar file which other jar files does it depend upon? This information is then stored in query optimized database tables that are refreshed with each update. The query operation is exposed as a service for use with automated configuration management processes. For example, dependent components defined in build scripts are updated with new dependencies, including the order of those dependencies as code is modified during the course of normal development.

5 Lessons Learned

L.L. Bean’s experience has been successful in demonstrating the value of using an architecture dependency analysis based approach to improve software agility. There were several lessons learned along the way that should be kept in mind.

While good tool support is essential, without good development practices, use of established coding standards, active configuration management, and consistent unit testing, tool use would be much more time-consuming and less effective.

Dependency information must be visible to be managed, but that alone is not enough to reduce maintenance costs and effort. It must be supported by the ability to try “what if” scenarios and creating prototypes to explore change impact.

Business drives requirements and ultimately the need to be agile. The “big bang” approach wasn’t viable in an environment with multiple development teams that had various and, often, conflicting development cycles, each with different business drivers. Moreover, it became evident that the difficult and costly attempt at using a “big bang” approach was not necessary. Following the incremental approach described in Section 3 development teams remain agile and refactor to use the layered architecture as part of their normal development cycles.

Beyond consolidating and repackaging code, there are often implications with respect to external (non-Java) component coupling. In some cases fully qualified Java packages were specified in scripts and property files. In other cases, Java classes referenced external components, which presented issues when consolidating code. The lesson learned was that change impact is often greater than what is originally estimated.

6 Limitations and Future Work

While it is believed that DSMs can be applied to systems implemented in other languages and databases, the L.L.Bean experience is only with Java based software. Therefore, the results of this experience may not generalize to other types of systems.

While the experience reported here has made a substantial impact on L.L. Bean's ability to maintain its code-base, we believe this is just one of many benefits that architecture analysis can provide. This report validates the DSM's support for evolvability, we are continuing to explore the potential for extracting other relationships from code, in particular run-time relationships, which can be used to identify the existence of Component and Connector (run-time) architectural styles and the application of the DSM to support analysis of a variety of run-time quality attributes.

7 Summary

The key to L.L. Bean's code restructuring success was increasing visibility of both system's architecture and the process. L.L. Bean has found that increasing the visibility of architecture greatly reduces architectural drift as the system evolves and at the same time reduces ongoing maintenance costs. Architectural visibility provides guidance for large-scale refactoring.

L.L. Bean also discovered that changing the structure of the system can sometimes be achieved without substantial code modifications and that large scale reorganization is a complex process that, when done with proper tool support and in a disciplined software development environment, can be effective.

The results of this experience demonstrate that architecture-based analysis can improve the productivity of software development. It is hoped that future research and practice will produce continued advancement in architectural support for improved software quality.

Acknowledgements. Sincere thanks to Doug Leland of L.L. Bean for his skillful mentorship and guidance, to David Smith and Tom Gould of L.L. Bean for their many insightful contributions to the Reuse Team and code dependency analysis.

References

1. Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester (1996)
2. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Reading (2003)
3. Ducasse, S., Ponisio, L., Lanza, M.: *Butterflies: A Visual Approach to Characterize Packages*. In: *Proceedings of the 11th International Software Metrics Symposium (METRICS 2005)*, Como, Italy (September 2005)
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading (1999)

5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley, Reading (1995)
6. Hautus, E.: Improving Java Software Through Package Structure Analysis. In: Proceedings of the 6th IASTED International Conference Software Engineering and Applications (SEA 2002), Cambridge, Massachusetts (September 2002)
7. Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*. Addison-Wesley, Reading (1999)
8. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture. In: *Process and Organization for Business Success*. Addison-Wesley, Reading (1997)
9. Kruchten, P.: Architectural Blueprints: The “4+1” View Model of Software Architecture. *IEEE Software* 12(6), 42–50 (1995)
10. Melton, H., Tempero, E.: An Empirical Study of Cycles among Classes in Java, Research Report UoA-SE-2006-1. Department of Computer Science, University of Auckland, Auckland, New Zealand (2006)
11. Melton, H., Tempero, E.: The CRSS Metric for Package Design Quality. In: Proceedings of the thirtieth Australasian conference on Computer science, Ballarat, Victoria, Australia, pp. 201–210 (2007)
12. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17(4) (October 1992)
13. Poulin, J.S.: *Measuring Software Reuse*. Addison-Wesley, Reading (1997)
14. Poulin, J.S.: Measurements and Metrics for Software Components. In: Heineman, G.T., Councill, W.T. (eds.) *Component-Based Software Engineering: Putting the Pieces Together*, pp. 435–452. Addison Wesley, Reading (2001)
15. Sangal, N., Waldman, F.: Dependency Models to Manage Software Architecture. *The Journal of Defense Software Engineering* (November 2005)
16. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, San Diego, California, pp. 167–176 (October 2005)
17. Stafford, J., Richardson, D., Wolf, A.: Architecture-level dependence analysis in support of software maintenance. In: Proceedings of the Third International Conference on Software Architecture, Orlando, Florida, pp. 129–132 (1998)
18. Stafford, J., Wolf, A.: Software Architecture. In: Heineman, G.T., Councill, W.T. (eds.) *Component-Based Software Engineering: Putting the Pieces together*, pp. 371–388. Addison Wesley, Reading (2001)
19. Lieherherr, K.J., Holland, I.M.A., Riel, A.J.: Object-oriented programming: an objective sense of style. In: OOPSLA 1988 Conference Proceedings, San Diego, California, September 25–30 (1988); *ACM SIGPLAN Not.* 23(11), 323–334 (1988)

Successful Architectural Knowledge Sharing: Beware of Emotions

Eltjo R. Poort¹, Agung Pramono², Michiel Perdeck¹,
Viktor Clerc², and Hans van Vliet²

¹ Logica, P.O. Box 159, 1180 AD Amstelveen, The Netherlands
{eltjo.poort,michiel.perdeck}@logica.com

² VU University, Amsterdam, The Netherlands
agungpramono@yahoo.com, {viktor,hans}@cs.vu.nl

Abstract. This paper presents the analysis and key findings of a survey on architectural knowledge sharing. The responses of 97 architects working in the Dutch IT Industry were analyzed by correlating practices and challenges with project size and success. Impact mechanisms between project size, project success, and architectural knowledge sharing practices and challenges were deduced based on reasoning, experience and literature. We find that architects run into numerous and diverse challenges sharing architectural knowledge, but that the only challenges that have a significant impact are the emotional challenges related to interpersonal relationships. Thus, architects should be careful when dealing with emotions in knowledge sharing.

Keywords: Software Architecture, Architecture Knowledge, Software Project Management.

1 Introduction

In recent years, Architectural Knowledge (AK), including architecture design decisions, has become a topic of considerable research interest. Management and sharing of AK are considered to be important practices in good architecting [10, 17, 5]. There has not been, however, much published research into the usage of AK related practices in industry.

In the beginning of 2008, the members of the architecture community of practice in a major Dutch IT services company¹ were surveyed. The main reason for this survey was to establish a baseline of current practice in Architectural Knowledge Sharing (AKS), and to gain insight into the mechanisms around AKS and related challenges in projects. The ABC company was interested in these mechanisms because they saw Architectural Knowledge management as a way to improve IT project performance. The architects were asked about the content, manner, reasons and timing of the AK sharing they did in their latest project; both obtaining and sharing knowledge towards others. They were also asked about the challenges they faced. Furthermore, they were asked to identify various properties of their latest project's context, such as project size and success factors.

¹ In this paper, this company will be identified as ABC.

Even though the architects surveyed all work for the same IT services company, according to the survey 64% of them is doing so mostly at customers' sites. As a consequence, the survey results represent a mix of AK sharing practices in ABC and in ABC's customer base, which includes major Dutch companies and government institutions.

2 Survey Description

The invitation to participate in the survey was sent out by e-mail to 360 members of the Netherlands (NL) Architecture Community of Practice (ACoP) of the ABC company. The ACoP consists of experienced professionals practicing architecture at various levels (business, enterprise, IT, software, and systems architecture) in project or consultancy assignments. The survey was closed after 3 weeks. By that time, 142 responses were collected. 97 respondents had answered the majority of the questions (93 had answered all). The other 45 responses were discarded because no questions about AK sharing had been answered. The survey consisted of 37 questions: 20 directly related to AK sharing, and 17 related to the context in which the AK sharing took place.

3 Analysis

The analysis of the 97 valid survey responses was performed in three phases: first, the current state of AK practice and challenges was established by comparing the respondents' answers to the 20 AK related questions. The analysis of four of these questions is presented in section 3.1 of this paper: three questions about AK practices and one about challenges in AK sharing. In phase one, we examined the responses by ordering and grouping them.

Second, the relationship between the AK practices and challenges and their context was analyzed by determining significant correlations between the AK-related responses and some of the 17 context-related questions. In this paper, the two context factors of project success and project size are analyzed systematically in section 3.2. The result of phase two is a set of statistically significant correlations between responses to AK related questions, and the size and success of the projects they pertained to.

In the third phase of the analysis, we reasoned and discussed about the results from the first two phases. Two of the authors have been practicing architects in the ABC company for more than a decade. Based on reasoning, literature and their experience we deduced causality and impact mechanisms from the correlations, leading to an observed impact model that is presented in section 3.3. Further discussions are presented in section 4.

3.1 State of AK Sharing Practice

In this section, the responses to four of the AK related questions are analyzed, presenting the results of phase 1 of the analysis.

The four questions are:

- What type of architectural knowledge have you provided to or acquired from ABC in your latest assignment?
- Why did you share architectural knowledge to your colleagues in ABC?

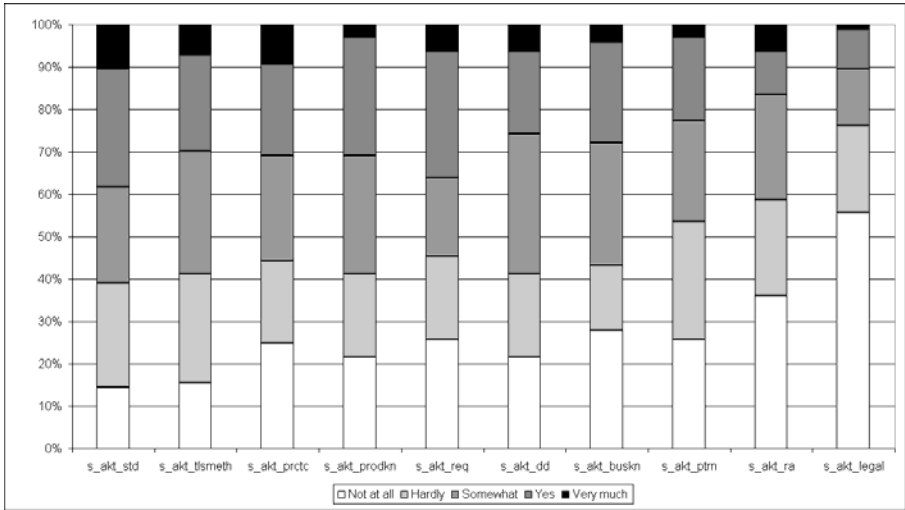


Fig. 1. Architectural Knowledge Types

- When did you share architectural knowledge in your latest assignment?
- What challenges in architectural knowledge sharing did you experience in your latest assignment?

Each question was provided with a set of predefined responses, determined in consultation between two experienced architects and two researchers. There was also the possibility for open text for missing answers. Respondents were asked to signify the applicability of those responses on a 5-point Likert scale. Table 1 lists the predefined responses to the questions, sorted by their average response values, which are listed in the third column. Each question is further analyzed in the following subsections. The two rightmost columns in the table list the Spearman’s rho correlations between the responses and the project context factors, which will be analyzed in section 3.2 below. We will start with the analysis of the responses without taking into account their contexts.

Architectural Knowledge Types *What type of architectural knowledge have you provided to or acquired from ABC in your latest assignment?*

The distribution of the response values is visualized in Fig. 1.² With the exception of reference architectures and legal knowledge, all types of architectural knowledge appear to be shared more or less equally. The least shared type of AK is legal knowledge: over 75% indicate they do not or hardly share it with ABC.

AK Sharing Motivation *Why did you share architectural knowledge to your colleagues in ABC?* The distribution of the response values is visualized in Fig. 2. These data tell us that most architects are either impartial to or agree with almost all motivation responses.

² The figures in this paper use the codified response IDs of the ID column in Table 1.

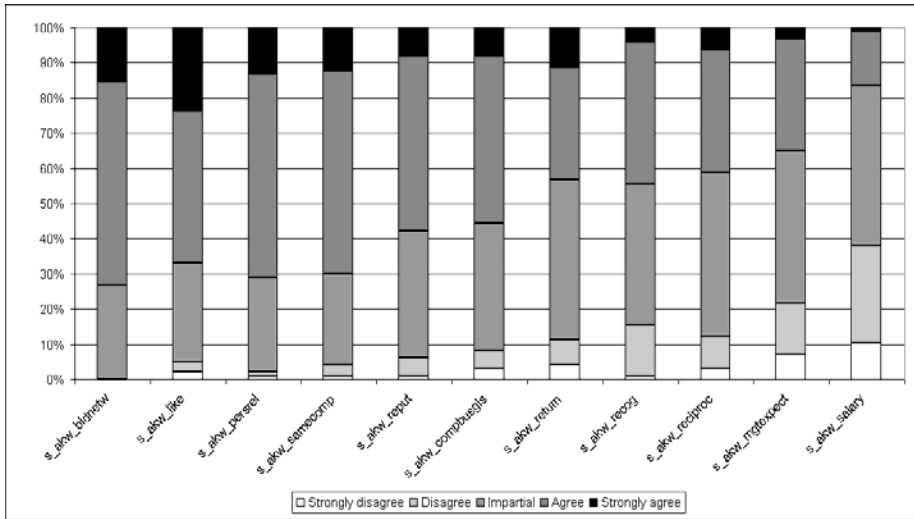


Fig. 2. AK Sharing Motivation

The only motivation that more architects disagree with (38%) than agree with (17%) is salary. A related finding is the unpopularity of management expectation as a motivator: 65% of respondents are at most impartial to this motivator.

AK Sharing Timing *When did you share architectural knowledge in your latest assignment?*

The distribution of the response values is visualized in Fig. 3. By far the most popular times to share AK are when problems occur, at the end of projects and when asked by colleagues (other than managers); these three timings are all used often or very often by over 50% of the architects. Almost 30% of architects indicate they never share AK "when management asks me to do so". We assume this is because in those cases management does not ask - an assumption supported by the observation that there is no lack of willingness to share (see Fig. 4). This fortifies our previous observation about management expectation as a motivator.

AK Sharing Challenges *What challenges in architectural knowledge sharing did you experience in your latest assignment?*

The distribution of the response values is visualized in Fig. 4. The ordering of the challenges by average response value in Table 1 allows an interesting categorization of challenges with descending response values:

s_chl_requnders, s_chl_stkhpart, s_chl_custdiv *Difficulty to achieve common understanding of requirements, participation from relevant stakeholders, and diversity in customer culture and business* are all related to communication issues on group level (as opposed to personal level); this is the category of challenges that most architects consider relevant in their latest projects.

Table 1. AK related responses, average values and correlations

	ID	avg	prj succ rho	prj size rho
Architectural knowledge types				
Standards; principles and guidelines	s_akt_std	2.95	-0.062	0.012
Tools and methods	s_akt_tsmeth	2.80	-0.096	.234*
Known and proven practices	s_akt_prctc	2.71	0.135	-0.017
Product and vendor knowledge	s_akt_prodkn	2.71	0.187	-.244*
Requirements	s_akt_req	2.71	0.178	-0.113
Design Decisions including alternatives; assumptions; rationale	s_akt_dd	2.69	0.1	-0.025
Business knowledge	s_akt_buskn	2.61	0.082	-0.023
Patterns and tactics	s_akt_ptrm	2.46	0.044	0.011
Reference architectures	s_akt_ra	2.28	0.074	-0.014
Legal knowledge	s_akt_legal	1.79	0.097	0.03
AK Sharing Motivation				
To build up my professional network	s_akw_bldnetw	3.89	-0.116	-0.009
I just like to share my knowledge	s_akw_like	3.84	0.115	-0.107
Personal relation with colleague(s)	s_akw_persrel	3.81	-.230*	0.037
We all work for the same company	s_akw_samecomp	3.77	0.109	-0.147
To enhance my professional reputation	s_akw_reput	3.59	0.042	0.022
To contribute to the company's business goals	s_akw_compbusgls	3.53	0.054	-0.014
I hope the favour will be returned some day	s_akw_return	3.39	-.204*	0.147
I will be recognised as a contributor	s_akw_recog	3.32	0.018	-0.107
I have received useful information from him/her	s_akw_reciproc	3.32	-.223*	-0.019
My management expects me to	s_akw_mgtexpect	3.09	.275**	-0.091
This may work in my favour at my next salary review	s_akw_salary	2.69	0.002	0.037
AK Sharing Timing				
Whenever needed to solve problems	s_akh_problems	3.48	0.153	-0.035
At the end of the project	s_akh_prjend	3.41	0.027	0.002
When colleagues ask me to do so	s_akh_collask	3.39	0.048	-0.066
When management ask me to do so	s_akh_mgtask	2.59	0.177	-0.052
Whenever I have time	s_akh_freetime	2.57	-0.025	0.065
In the evening	s_akh_evening	2.53	0.012	-0.008
Continuously during the project	s_akh_prjcnt	2.34	.205*	-0.133
AK Sharing Challenges				
Difficulty to achieve common understanding of requirements	s_chl_requnders	3.82	-0.146	0.055
Difficulty to achieve appropriate participation from relevant stakeholders	s_chl_stkhpert	3.66	-0.165	0.017
Diversity in customer culture and business	s_chl_custdiv	3.61	-0.102	0.051
Poor quality of information	s_chl_infqual	3.42	-0.11	0.071
Lack of information	s_chl_inflack	3.31	-0.086	0.12
Inconsistency in information obtained from different sources	s_chl_infincons	3.26	-0.114	0.088
Lack of time	s_chl_time	3.25	0.06	-0.017
Delays in delivery	s_chl_delays	3.24	-0.167	0.194
Difficulty of obtaining the appropriate skills within the project	s_chl_skills	3.24	-0.115	0.11
Conflicts and differences of opinion	s_chl_conflict	3.19	-.214*	0.156
Difficulty to organise effective meetings	s_chl_effmeet	3.09	-0.153	0.17
Lack of informal communication	s_chl_lackinformal	3.01	-0.204	.226*
Inaccessibility of technical facilities	s_chl_tinacc	2.99	-0.183	.272**
Growing and shrinking of project population	s_chl_growshrink	2.82	-0.117	.317**
Lack of trust between the project locations	s_chl_sitetrust	2.77	-.272**	.244*
Project personnel turnover	s_chl_perstov	2.67	-0.116	.270**
No appreciation from (project or competence) management	s_chl_mgtappr	2.60	-0.125	.241*
No willingness to share knowledge	s_chl_nowill	2.39	-.224*	.245*

* Correlation is significant at the 0.05 level (2-tailed).

** Correlation is significant at the 0.01 level (2-tailed).

s_chl_infqual, s_chl_inflack, s_chl_infincons *Poor quality, inconsistency or lack of information* are about issues with quality or absence of codified AK; this is the second most commonly relevant category of challenges.

s_chl_time, s_chl_delays *Lack of time and delays in delivery* are related to planning; this is the third most commonly relevant category of challenges.

other challenges all less commonly relevant than the three categories mentioned above, are related to obtaining resources, interpersonal issues, teaming, continuity and management.

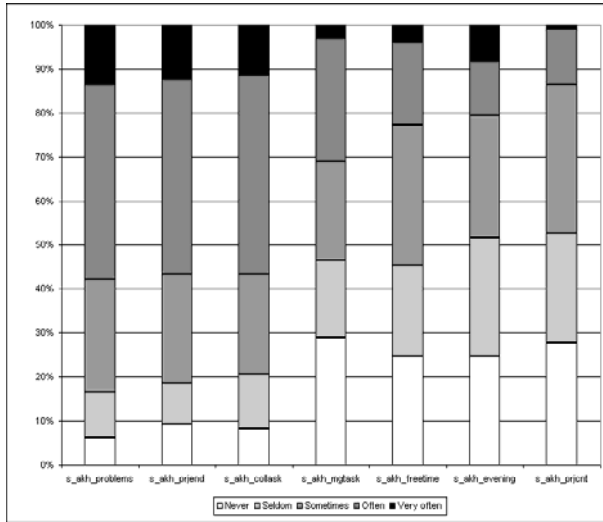


Fig. 3. AK Sharing Timing

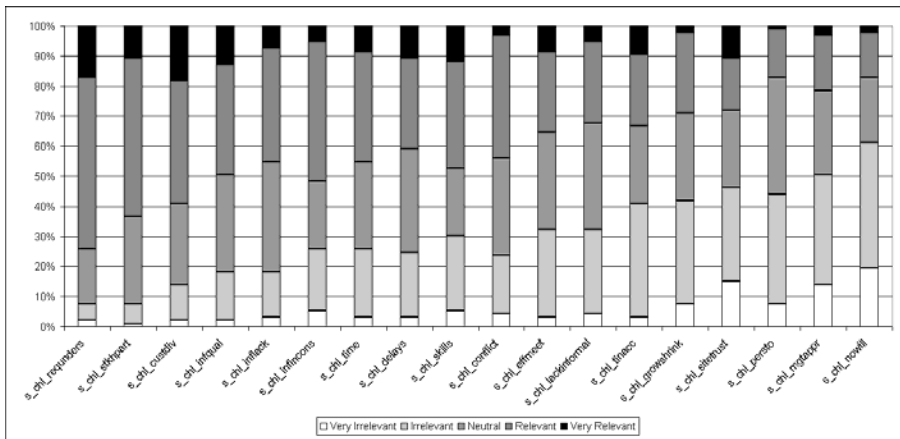


Fig. 4. AK Sharing Challenges

In discussions about challenges in knowledge sharing, “knowledge is power” [2] is often cited as a reason for professionals not to want to share knowledge. In our survey however, *lack of willingness to share knowledge* emerges as the least relevant challenge, which the majority of architects find irrelevant, and which only 18% find relevant. The next least relevant challenge is *lack of management appreciation*, which only 21% find relevant. The unpopularity of this response suggests that, even though we have seen in section 3.1 that both salary and management expectations are at the bottom of the list of reasons to share AK, architects are not actively discouraged by their management’s apparent disinterest. Seeing that 65% of respondents are at most impartial to

management as a motivator (Fig. 2) and almost 80% are at most impartial to management as a challenge (Fig. 4), one might conclude that *architects do not see management as an important factor in Architectural Knowledge Sharing*. As we will see later on, they might be wrong about this.

3.2 AK Practices in Context

In this section, we analyze the relationship between the AK practices and challenges and their project context, by examining significant correlations between the AK-related responses and some of the context-related questions. The two context factors analyzed here are project success and project size.

The first context factor analyzed is project success, as perceived by the architects. Perceived project success³ is determined by asking the architects how they rated seven aspects of project success on a 5-point Likert scale from Poor to Excellent. The aspects they rated are: Sticking to budget, Delivery in time, Client satisfaction, Management support, Personnel turnover, Solution quality and Team satisfaction. The combined answers of these seven aspects were subsequently averaged to obtain a quantification of overall project success per case. Cronbach's alpha test for internal consistency [6] was used to verify that these seven responses measure the same construct of success ($\alpha = 0.82$).

The second context factor analyzed is project size. Projects were assigned an exponential size category between 1 and 5, based on the number of project members: 10 or less became category 1, 11 through 30 category 2, 31 through 100 category 3, 101 through 300 category 4, and over 300 category 5.

Table 1 shows the Spearman's rho correlations between project success and the AK practice related responses in column *prj succ rho*. Correlations between project size category and the AK practice related responses are in column *prj size rho*.

Correlations with a positive or negative slope of over 0.2 and a significance level of under .05 (indicated by one or two asterisks) are considered significant and discussed here. In the discussion of the correlations, some speculation is presented as to the underlying mechanisms, based on the experience of the practicing architects among the authors.

Cause and Effect One of the objectives of this survey was to gain insight into mechanisms around architectural knowledge sharing in projects. In other words, we were looking for ways in which Architectural Knowledge Sharing impacts projects and vice versa - questions of cause and effect.

When analyzing correlations like the ones found in this survey, the question of causality between the correlated measurements deserves careful consideration. The mere presence of a correlation by itself does not imply a causal relationship. In order to determine potential causality, we resorted to three additional means: reasoning, literature and the experience of two of the authors as practicing architects in ABC.

The four categories of measurements we are correlating here are:

³ In this paper, we use the terms "project success" and "perceived project success" interchangeably, always meaning the success as perceived by the architects and reported in the survey.

AKS Practices the responses related to the type, motivation and timing of architectural knowledge sharing

AKS Challenges the responses to the question: "What challenges in architectural knowledge sharing did you experience in your latest assignment?"

Project Success the perceived success of the respondents' latest project

Project Size the size of the respondents' latest project (category based on number of project members)

There are six possible correlations between these four categories. We are not analyzing correlations between AKS Practices and Challenges. Fig. 5 visualizes potential causality arrows for the five remaining possible correlations. In this figure and Fig. 8, a causality arrow from A to B symbolizes that A has impact on B, implying that making changes to A would cause related changes in B. The arrows are based on the following reasoning:

Project Size ↔ **Project Success** Project size is well known to influence project success in many ways, both in literature [8, 9] and experience, so the primary arrow of causality is from Size to Success

Project Size ↔ **AKS Practices** Experience indicates that mechanisms determining project size are only marginally impacted by architectural knowledge sharing; on the other hand, project size determines factors like organizational and physical distance between project members, which are obvious factors in AKS. We conclude that any correlation found means that project size impacts AKS, and not the other way around.

Project Size ↔ **AKS Challenges** Like with AKS Practices, project size causes AKS challenges. There are some challenges that may in time conversely influence project size: for example, difficulty to obtain the appropriate skills may either lead to a smaller project because there is no staff available, or to a larger project because the lower skill level is compensated by adding more staff. We conclude that there is a primary causal arrow from project size to AKS challenges, and a potential secondary reverse arrow.

Project Success ↔ **AKS Practices** Examples of causality in both directions are experienced: e.g., a more successful project may lead to a better atmosphere causing more knowledge to be exchanged, or conversely more knowledge sharing may contribute to a more successful project. We conclude that we cannot a priori attach causality direction to correlations found between project success and AKS practices.

Project Success ↔ **AKS Challenges** The word *challenge* is used here as a synonym for *obstacle*, which can be defined as *something that makes achieving one's objectives more difficult*. Since the objective here is a successful project, the primary arrow of causality is by definition from Challenge to Success. There is also a possibility of reverse causality here: challenges may be exacerbated or caused by (lack of) project success, e.g. the atmosphere in an unsuccessful project may lead to lack of trust.

The causality arrows between the four categories of measurements as visualized in Fig. 5 will be elaborated at the end of this section, based on correlations measured.

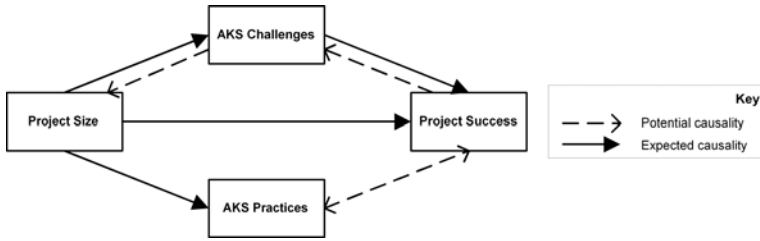


Fig. 5. Causality as deduced from reasoning, literature and experience

Correlation with project success We now discuss the correlations between architectural practices and challenges and project success. In column 4 of Table 1, we find 8 significant correlations. Summarizing:

In *more successful* projects, architects tend to:

- be *less* motivated to share AK for interpersonal relationship reasons, but are more motivated by their management’s expectations
- face *less* challenges related to interpersonal relationships

We find no correlation between project success and the type of the Architectural Knowledge shared.

Motivation: s_akw_persrel, s_akw_return, s_akw_reciproc *Personal relation with colleagues, or because I have received or hope to receive information from the other:* remarkably, all motivation responses that are related to one-to-one relationships between colleagues show a significant negative correlation with project success. Fig. 6(a) visualizes this relationship, showing a clearly downward slanting cluster: the x-axis represents the individual architects’ average mark given to these three responses.⁴ There are many possible explanations, but in view of our findings about AK sharing challenges a few items further down, the most plausible one appears to be related to trust. Problems in projects tend to reduce trust, which might cause architects to place more value on interpersonal motives.

Motivation: s_akw_mgtexpect *My management expects me to:* even though management expectations are considered one of the least important motivations for sharing AK by the architects, it is the only motivation that has a positive correlation with project success. The explanation may also be related to trust levels: architects working on successful projects have more confidence in their management, and hence are more inspired or motivated by them.

Timing: s_akh_prjcnt *Continuously during the project:* the only AK sharing timing response that has a correlation with project success. However, visual inspection of Fig. 6(b) suggests that this is a spurious effect.

Challenges: s_chl_conflict, s_chl_sitetrust and s_chl_nowill *Conflicts and differences of opinion, Lack of trust between the project locations, and No willingness to share knowledge.* Since there is by definition a causality between AKS challenges and

⁴ The lines in the scatter plots in this section represent linear regression fit lines and their 95% confidence interval.

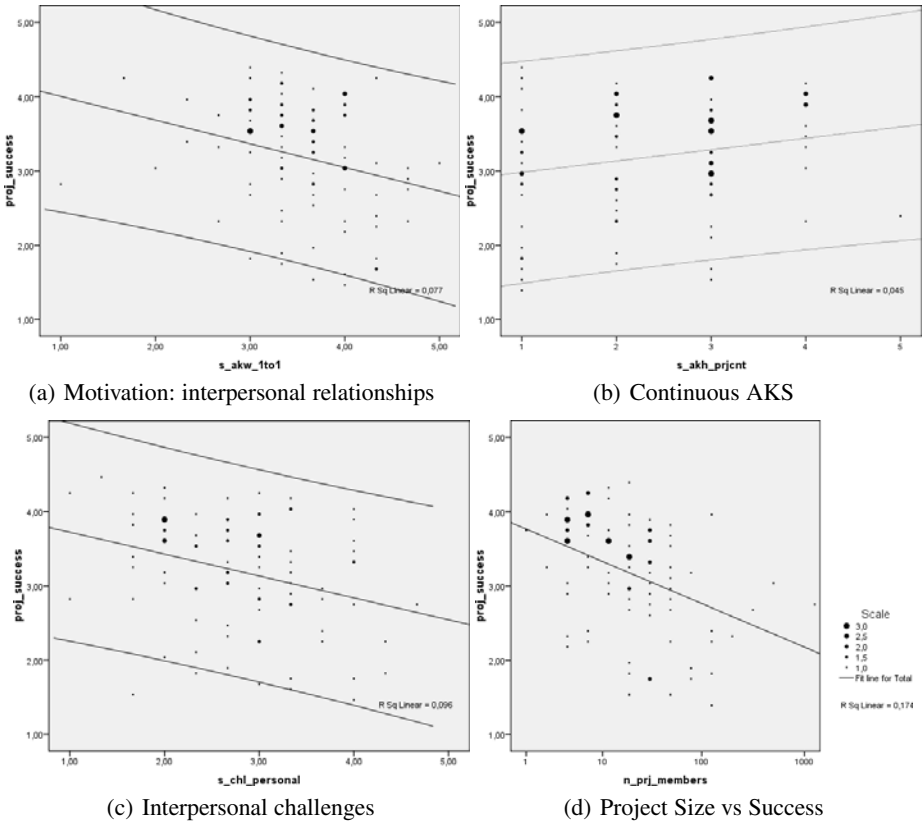


Fig. 6. Various AKS parameters plotted against project success

project success, we *expect* to find correlations. Remarkably, only three challenges are significantly correlated with project success. These three challenges, all with a very clear negative correlation, have in common that they are related to interpersonal relationships and emotion: conflicts, trust and willingness. We have plotted the correlation between project success and the individual architects' average mark given to these three responses related to interpersonal challenges in Fig. 6(c). As for the other challenges, finding *no* correlation indicates one of two things: either the challenge is so insignificant that the correlation is too small to be measured in a sample this size, or the challenge is somehow neutralized.

From these correlations, we can draw the following conclusion: the only significant AKS challenges that are not neutralized in projects are those related to emotion and interpersonal relationships. In less successful projects, there is less trust and willingness to share AK, and more conflict. This appears not to affect the type of AK shared. It does, however, have a significant effect on architects' motivation to share architectural knowledge: in more successful projects, they are more motivated by management and less by interpersonal relationships between colleagues.

Correlation with project size We proceed to discuss the correlations between architectural practices and challenges and project size, as documented in column 5 of Table 1. We find 9 significant correlations. Summarizing:

In *larger projects*, architects tend to:

- face significantly *more* challenges of multiple kinds
- share *more* knowledge about tools and methods, but *less* about products and vendors

Project size has no effect on AK sharing motivation or timing.

s_akt_tlsmeth Architects in larger projects share slightly more *information related to tools and methods* than architects in smaller projects. This is likely due to the fact that there are simply more developers to educate on tools and methods.

s_akt_prodkn Architects in some smaller projects tend to share more *knowledge related to products and vendors*. We suspect that this is due to the fact that in larger projects, decisions about products and vendors are often made on a higher (management) level, whereas smaller project architects are more likely to be involved in these decisions, and hence have to share more knowledge related to products and vendors.

AKS challenges Table 1 shows that out of the 18 types of challenges surveyed, 7 are significantly correlated to project size. We have also calculated the aggregated AKS challenge level as the average of each architect's challenge-related responses. It turns out this aggregated AKS challenge level is correlated to project size with a correlation coefficient of 0.356 at a 0.001 significance level. The seven challenges at the bottom of Table 1 are the only ones that are also individually correlated to project size. Apparently, some challenges are universal, and others are considered less relevant in smaller projects, bringing down their average response value. We have illustrated this by plotting the average response values of both the seven least commonly relevant and the eleven most commonly relevant challenges against project size in Fig. 7. The figure confirms that there is indeed a clear upward trend, and that it is steeper for the less commonly relevant challenges.

Based on the fact that larger projects are likely to include more distinct departments or locations, and the well-known issue of tension between departments, we would expect larger projects to suffer more from emotion-related challenges. We do indeed find correlations between project size and lack of both willingness (.245) and trust (.244), but no significant correlation with the challenge of conflicts and differences of opinion.

3.3 Refined Model of Causality

We now use the correlations observed in the previous section to obtain a more detailed picture of causality. Fig. 8 shows the causality arrows between the four categories of measurements as visualized in Fig. 5, but the AKS category boxes have been replaced with more specific subcategories corresponding to the responses that showed correlations. Additional symbols show whether correlations are positive or negative. Specifically, we have:

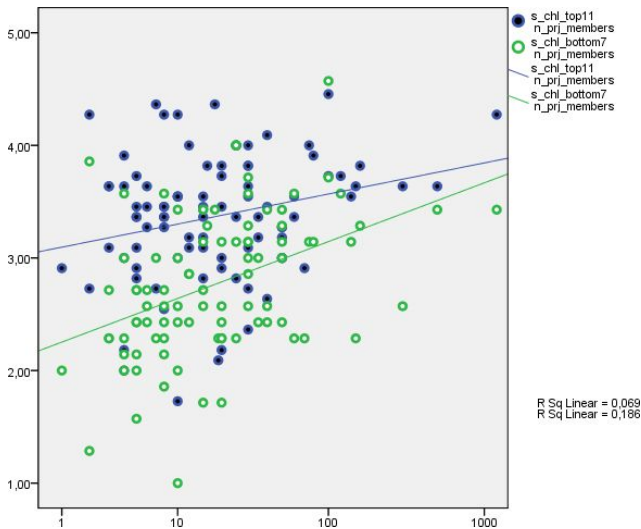


Fig. 7. AKS Challenges versus project size

- replaced the generic box *AKS Challenges* with a box *Less common AKS Challenges*, representing the seven least common AKS challenges that have significant positive correlations with project size
- created a box *Interpersonal challenges* inside the *Less common AKS Challenges* box, representing the three challenges related to willingness, trust and conflict that are negatively correlated with project success
- replaced the generic *AKS Practices* box with four specific boxes representing the practices that we have found to be correlated with either project size or project success
- added + and - symbols to the causality arrows representing the sign of the observed correlations

There is one correlation that we had not discussed yet: that between project size and perceived project success. Fig. 6(d) displays a very clear relationship between project size and perceived project success. Perceived project success and the logarithmic project size category described above show a negative Spearman's rho correlation coefficient of -0.449, with a significance of 0.000. This is in line with results found by [9], and conversely provides some additional validation that our input data behave according to known properties of IT projects. Brooks [8] gives a clear explanation of one of the mechanisms that cause this correlation. Surprisingly, a more recent survey [7] does not find this correlation.

Fig. 8 summarizes in one picture the combined mechanisms in the interplay between AKS and project size and success. We see how project size impacts some challenges, and which challenges impact project success. We also see that project size impacts the type of knowledge shared, and we observe a relationship between AKS motivation and project success, a relationship with an as yet undetermined arrow of causality.

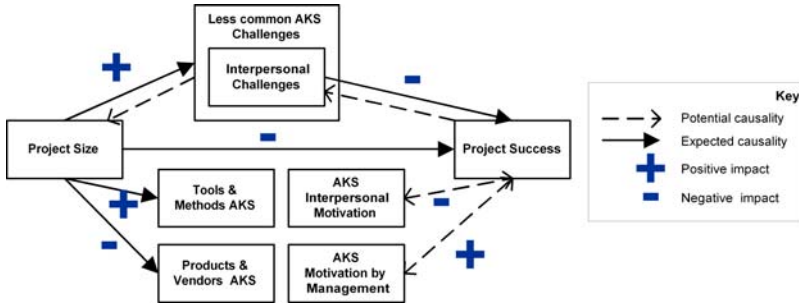


Fig. 8. Causality as observed

4 Discussion and Related Work

In this section, we further discuss the results found above and threats to validity, and we relate them to additional related material found in literature.

4.1 Threats to Validity

These results are based on a survey of architects in one IT services company in one country. This limitation is somewhat softened by the fact that 64% of respondents work mostly at customers’ sites, but the results are certainly influenced by cultural aspects of both the ABC company and the Netherlands location. It would be very interesting to repeat the survey in other companies and locations.

The ordering of the responses in Table 1 and the response value distribution bar charts is based on average response values. The meaning of the average number itself is not clear, since the Likert-scale is not equidistant. An alternative ordering quantity would be the percentile responses of e.g. the two most positive Likert values. This would have the advantage of being able to say exactly what the ordering quantity means, but the disadvantage of ignoring the information inherent in the detailed distribution of responses. Visual inspection of the bar charts shows that, with the exception of Fig. 1, the order of the responses would not be that much different, specifically in those cases where we have based reasoning on the response ordering. As an example: the ”seven least commonly relevant challenges” in Fig. 4 that we have discussed above would also be the seven bottom-most challenges if ordered by percentile of respondents answering ”Relevant” or ”Very Relevant”.

There is a weakness in the four questions analyzed in section 3.1, in that they all appear to have slightly different scopes for AK sharing: two of the questions are about sharing towards or from ABC, one is explicitly about sharing with colleagues, and two are explicitly from the perspective of the originator. These scope differences are ignored in the analysis, since they cannot be remedied without redoing the survey.

A final threat is caused by our approach of doing multiple statistical tests, and deriving our model from significant statistical results found in those tests. This approach implies a risk of introducing spurious statistical results in the model. We have mitigated

this risk by using reasoning, experience and literature, but it would be interesting to further validate the model by using it to predict results in other surveys.

4.2 Project Success in Literature

Project success has long been an active research topic. Traditionally, project success is defined in terms of meeting time, cost and quality objectives [14]. These correspond to the first three of the seven project success criteria used in our survey. More recently, it has been observed that projects can be successful in ways that cannot be measured by these traditional criteria. Based on these insights, Baccarini et al. [1] have constructed a conceptual framework for project success. Baccarini's framework distinguishes between *Project Management Success*, which includes the three traditional criteria of time, cost and process quality, and *Product Success*, which adds criteria related to the product in a more strategic way, involving the product's goal and purpose and product satisfaction. In Baccarini's framework, our criteria would all fall in the Project Management Success category, with the exception of Solution Quality. Team Satisfaction in Baccarini's framework can relate to both project and product; in our experience, this is especially true for architects, who derive a large part of their job satisfaction from product quality. This observation is confirmed by research by Linberg et al. [11] and more recently by Procaccino et al. [15], who observe that developers' perception of project success often deviates significantly from the traditional criteria. Developers (including architects) tend to judge success by criteria that extend beyond the project, sometimes even to the extent that even canceled projects can be successful in their eyes.

4.3 Motivation and Emotion in Architectural Knowledge Sharing

An interesting finding about motivation in this survey is the observed shift in motivation source from colleagues to management in more successful projects. Could there be an either/or effect, in the sense that the 1-on-1 motivation by colleagues and motivation by management are somehow mutually exclusive? In that case, one would expect a negative correlation between these two motivation sources, which we did not measure (Spearman's $\rho = 0.107$ with a two-tailed significance of 0.295). We conclude that the mechanisms causing these shifts are independent. The finding does, however, cause one to wonder about architects' apparent indifference to management expectations as either a motivator or a challenge. The well-known Chaos Reports [16] already showed empirical evidence for management attention being a key project success factor.

Markus already identified the importance of being aware of one's motivation long before the term *architect* was used in the context of system design: "Self-examination of interests, motives, payoffs, and power bases will lend much to the implementor's ability to understand other people's reactions to the systems the implementor is designing..." [12]. In literature, motivation is reported to have the single largest impact on developer productivity [4, 13]. Moreover, in system development, the architecture represents the system's earliest design decisions with the highest impact on success [3]. Combining these facts, it is only to be expected that the motivation to share Architectural Knowledge is correlated with project success. Our results not only point to the importance of motivation and its source, but also shed some light on the mechanisms

through which motivation and emotion impact project success through Architectural Knowledge management.

Finally, some words on the topic of *emotion*, a term that we introduced in section 3.2 as the common element between the three only challenges that have a significant negative correlation with project success: *Conflicts and differences of opinion*, *Lack of trust between the project locations* and *No willingness to share knowledge*. During the analysis, we often wondered how it was possible that we did not find any significant correlation between the *other* challenges in AKS and Project Success. Consider, for example, the most commonly encountered challenge: *Difficulty to achieve common understanding of requirements*. How can a project be successful without common understanding of requirements? As stated above, the only plausible explanation is that all of these other challenges are apparently neutralized. With neutralize we mean that if these challenges occur, there are other factors that prevent them from having a significant impact on project success. In the case of our example, these could be compensating activities to promote the common understanding of requirements, such as client meetings. In the end, the only challenges that are not neutralized are those related to lack of trust, willingness, conflicts and differences of opinion: all issues in interpersonal relationships that have a strong negative emotional connotation. Apparently, it is harder for architects to neutralize challenges when such negative emotions are involved. This is a phenomenon that the practicing architects among the authors have often observed in real life, and it should be no surprise, given that architects are human beings. The significant finding here is that these emotional challenges are not neutralized where all other challenges are, and hence they merit extra attention, leading to the warning in our title: *Beware of Emotions*.

We conclude:

FOR ARCHITECTS, TO UNDERSTAND THEIR MOTIVATION AND DEAL WITH EMOTIONS ARE CRUCIAL KNOWLEDGE SHARING SKILLS.

5 Conclusions

We set out on this survey with two goals, which were both achieved: to establish the current state of architectural knowledge sharing in the ABC company and its customers, and to gain insight into the mechanisms around architectural knowledge sharing in projects. In order to gain this insight, we looked at architects' responses to four questions about AK sharing, and the correlations between these responses and their latest projects' success and size, and we reasoned about impact mechanisms and causality.

The analysis revealed the following mechanisms:

- Architects face many challenges sharing architectural knowledge in projects;
- these challenges are more numerous and diverse in larger projects than in smaller ones.
- The most common of these challenges are related to group level communication issues, the quality of codified knowledge and planning issues;
- however, these common challenges are not correlated with project success, so apparently they are generally neutralized somehow.

- The only challenges that *are* correlated with project success are the ones related to interpersonal relationships: conflicts, trust and willingness to share knowledge.
- Architects' motivation to share knowledge is more personal in less successful projects.
- Architects do not see management as an important factor in Architectural Knowledge Sharing, but those architects that are motivated by management tend to work in more successful projects.

Our final conclusion is that *dealing with emotions* is a crucial factor in how architectural knowledge sharing leads to successful projects. It is important for architects to understand their motivation, and they should be careful when dealing with emotions when sharing knowledge.

References

1. Baccarini, D.: The logical framework method for defining project success. *Project Management Journal* 30, 25–32 (1999)
2. Bacon, S.F.: *Religious Meditations*, 1597
3. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. Addison-Wesley, Reading (2003)
4. Boehm, B.: *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs (1981)
5. Clements, P., Shaw, M., Shaw, M.: The golden age of software architecture: A comprehensive survey, tech., Technical report (2006)
6. Cronbach, L.J.: Coefficient alpha and the internal structure of tests. *Psychometrika* 16(3), 297–334 (1951)
7. El Emam, K., Koru, A.G.: A replicated survey of IT software project failures. *IEEE Software*, 84–89 (September/October 2008)
8. Frederick, J., Brooks, P.: *The Mythical Man-Month: Essays on Software Engineering*, 20th edn. Addison-Wesley, Reading (1995)
9. Jones, C.: *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, Reading (2000)
10. Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) *QoSA 2006*. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
11. Linberg, K.: Software developer perceptions about software project failure: a case study. *The Journal of Systems and Software* 49, 177–192 (1999)
12. Markus, M.L.: Power, politics, and M.I.S. implementation. *Commun. ACM* 26(6), 430–444 (1983)
13. McConnell, S.: *Rapid Development*. Microsoft Press (1996)
14. Pinto, J., Slevin, D.: Project success: definitions and measurement techniques. *Project Management Journal* 19, 67–72 (1988)
15. Procaccino, J.D.: What do software practitioners really think about project success: an exploratory study. *Journal of Systems and Software* 78, 194–203 (2005)
16. Standish Group, *Chaos Report* (1994)
17. Tyree, J., Akerman, A.: Architecture decisions: Demystifying architecture. *IEEE Software* 22(2), 19–27 (2005)

Toward a Catalogue of Architectural Bad Smells

Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic

University of Southern California, Los Angeles, CA, USA
{joshuaga, dpopescu, gedwards, neno}@usc.edu

Abstract. An *architectural bad smell* is a commonly (although not always intentionally) used set of architectural design decisions that negatively impacts system lifecycle properties, such as understandability, testability, extensibility, and reusability. In our previous short paper, we introduced the notion of architectural bad smells and outlined a few common smells. In this paper, we significantly expand upon that work. In particular, we describe in detail four representative architectural smells that emerged from reverse-engineering and re-engineering two large industrial systems and from our search through case studies in research literature. For each of the four architectural smells, we provide illustrative examples and demonstrate the smell’s impact on system lifecycle properties. Our experiences indicate the need to identify and catalog architectural smells so that software architects can discover and eliminate them from system designs.

1 Introduction

As the cost of developing software increases, so does the incentive to evolve and adapt existing systems to meet new requirements, rather than building entirely new systems. Today, it is not uncommon for a software application family to be maintained and upgraded over a span of five years, ten years, or longer. However, in order to successfully modify a legacy application to support new functionality, run on new platforms, or integrate with new systems, evolution must be carefully managed and executed. Frequently, it is necessary to *refactor* [1], or restructure the design of a system, so that new requirements can be supported in an efficient and reliable manner.

The most commonly used way to determine how to refactor is to identify code *bad smells* [2] [1]. Code smells are implementation structures that negatively affect system *lifecycle properties*, such as understandability, testability, extensibility, and reusability; that is, code smells ultimately result in maintainability problems. Common examples of code smells include very long parameter lists and duplicated code (i.e., clones). Code smells are defined in terms of *implementation-level* constructs, such as methods, classes, parameters, and statements. Consequently, refactoring methods to correct code smells also operate at the implementation level (e.g., moving a method from one class to another, adding a new class, or altering the class inheritance hierarchy).

While detection and correction of code smells is one way to improve system maintainability, some maintainability issues originate from poor use of *software*

architecture-level abstractions — components, connectors, styles, and so on — rather than implementation constructs. In our previous work [3], we introduced the notion of *architectural bad smells* and identified four representative smells. *Architectural bad smells* are combinations of architectural constructs that induce reductions in system maintainability. Architectural smells are analogous to code smells because they both represent common “solutions” that are not necessarily faulty or errant, but still negatively impact software quality. In this paper, we expand upon the four smells identified in our previous work by describing them in detail and illustrating their occurrence in case studies from research literature and our own architectural recovery [4] [5] and industrial maintenance efforts.

The remainder of this paper is organized as follows. Section 2 explains the characteristics and significance of architectural smells. Section 3 summarizes research efforts in related topics. Section 4 introduces two long-term software maintenance efforts on industrial systems and case studies from research literature that we use to illustrate our four representative architectural smells. Section 5 describes our four architectural smells in detail, and illustrates the impact of each smell through concrete examples drawn from the systems mentioned in Section 4. Finally, Section 6 provides closing discussion and insights.

2 Definition

In this section, we define what constitutes an architectural smell and discuss the important properties of architectural smells.

We define a software system’s *architecture* as “the set of principal design decisions governing a system” [6]. The system stakeholders determine which aspects are deemed to be “principal.” In practice, this usually includes (but is not limited to) how the system is organized into subsystems and components, how functionality is allocated to components, and how components interact with each other and their execution environment.

The term *architectural smell* was originally used in [7]. The authors of [7] define an architectural smell as a bad smell, an indication of an underlying problem, that occurs at a higher level of a system’s granularity than a code smell. However, we found that this definition of architectural smell does not recognize that both code and architectural smells specifically affect lifecycle qualities, not just any system quality. Therefore, we define architectural smells as a commonly used architectural decision that negatively impacts system lifecycle qualities. Architectural smells may be caused by applying a design solution in an inappropriate context, mixing combinations of design abstractions that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity. Architectural smells must affect lifecycle properties, such as understandability, testability, extensibility, and reusability, but they may also have harmful side effects on other quality properties like performance and reliability. Architectural smells are remedied by altering the internal structure of the system and the behaviors of internal system elements without changing the external behavior of the system. Besides defining architectural smells explicitly in terms of lifecycle

properties, we extend, in three ways, the definition of architectural smell found in [7].

Our first extension to the definition is our explicit capture of architectural smells as design *instances* that are independent from the engineering *processes* that created the design. That is, human organizations and processes are orthogonal to the definition and impact of a specific architectural smell. In practical terms, this means that the detection and correction of architectural smells is not dependent on an understanding of the history of a software system. For example, an independent analyst should be able to audit a documented architecture and indicate possible smells without knowing about the development organization, management, or processes.

For our second extension to the definition, we do not differentiate between architectural smells that are part of an *intended* design (e.g., a set of UML specifications for a system that has not yet been built) as opposed to an *implemented* design (e.g., the implicit architecture of an executing system). Furthermore, we do not consider the non-conformance of an implemented architecture to an intended architecture, by itself, to be an architectural smell because an implemented architecture may improve maintainability by violating its intended design. For example, it is possible for an intended architecture of a system to include poor design elements, while the (non-conforming) implemented architecture replaces those elements with better solutions.

For our last extension, we attempt to facilitate the detection of architectural smells through specific, concrete definitions captured in terms of standard architectural building blocks — components, connectors, interfaces, and configurations. Increasingly, software engineers reason about their systems in terms of these concepts [8,6], so in order to be readily applicable and maximally effective, our architectural smell definitions similarly utilize these abstractions (see Section 5). The definition in [7] does not utilize explicit architectural interfaces or first-class connectors in their smells.

In many contexts, a design that exhibits a smell will be justified by other concerns. Architectural smells always involve a trade-off between different properties, and the system architects must determine whether action to correct the smell will result in a net benefit. Furthermore, refactoring to reduce or eliminate an architectural smell may involve risk and almost always requires investment of developer effort.

3 Related Work

In this section, we provide an overview of four topics that are directly related to architectural smells: code smells, architectural antipatterns, architectural mismatches, and defects.

The term *code smells* was introduced by Beck and Fowler [2] for code structures that intuitively appear as bad solutions and indicate possibilities for code improvements. For most code smells, refactoring solutions that result in higher quality software are known. Although bad smells were originally based on subjective intuitions of bad code practice, recent work has developed ways to detect

code smells based on metrics [9] and has investigated the impact of bad smells using historical information [10]. As noted in Section 1, code smells only apply to implementation issues (e.g., a class with too many or too few methods), and do not guide software architects towards higher-level design improvements.

Closely related to code smells are *antipatterns* [11]. An antipattern describes a recurring situation that has a negative impact on a software project. Antipatterns include wide-ranging concerns related to project management, architecture, and development, and generally indicate organizational and process difficulties (e.g., design-by-committee) rather than design problems. Architectural smells, on the other hand, focus on design problems that are independent of process and organizational concerns, and concretely address the internal structure and behavior of systems. The general definition of antipatterns allows both code and architectural smells to be classified as antipatterns. However, antipatterns that specifically pertain to architectural issues typically capture the causes and characteristics of poor design from a system-wide viewpoint (e.g., stove-piped systems). Therefore, not all architectural antipatterns are defined in terms of standard architectural building blocks (e.g., vendor lock-in). Defining architectural smells in terms of standard architectural building blocks makes it possible to audit documented or recovered architecture for possible smells without needing to understand the history of a software system. Furthermore, architectural antipatterns can negatively affect any system quality, while architectural smells must affect lifecycle properties.

Another concept similar to architectural smells is *architectural mismatch* [12]. Architectural mismatch is the set of conflicting assumptions architectural elements may make about the system in which they are used. In turn, these conflicting assumptions may prevent the integration of an architectural element into a system. Work conducted in [13] and [14] has resulted in a set of conceptual features used to define architectural designs in order to detect architectural mismatch. While instructive to our work, architectural mismatch research has focused heavily on the functional properties of a system without considering the effects on lifecycle properties.

Finally, *defects* are similar to architectural smells. A defect is a manifestation of an error in a system [15]. An error is a mental mistake made by a designer or developer [15]. In other words, a defect is an error that is manifested in either a requirements, design, or implemented system that is undesired or unintended [16]. Defects are never desirable in a software system, while smells may be desirable if a designer or developer prefers the reduction in certain lifecycle properties for a gain in other properties, such as performance.

4 Systems under Discussion

Our experience with two long-term software projects brought us to the realization that some commonly-used design structures adversely affect system maintainability. In this section, we introduce these projects by summarizing their context and objectives. Later in the paper, we utilize specific examples from these projects to illustrate the impact of architectural bad smells.

Maintenance of large-scale software systems includes both architectural *recovery* and *refactoring* activities. Architectural recovery is necessary when a system’s conceptual architecture is unknown or undocumented. Architectural refactoring is required when a system’s architecture is determined to be unsatisfactory and must be altered. We discovered architectural bad smells during both an architectural recovery effort (summarized in Section 4.1) and an architectural refactoring effort (summarized in Section 4.2). To substantiate our observations, we found further examples of architectural bad smells that appear in recovery and refactoring efforts published in the research literature.

4.1 Grid Architecture Recovery

An extensive study of grid system [17] implementations contributed to our collection and insights of architectural smells. Grid technologies allow heterogeneous organizations to solve complex problems using shared computing resources. Four years ago, we conducted a pilot study [18] in which we extracted and studied the architecture of five widely-used grid technologies and compared their architectures to the published grid reference architecture [17]. We subsequently completed a more comprehensive grid architecture recovery project and recently published a report [5] on the architectures of eighteen grid technologies, including a new reference architecture for the grid. The examined grid systems were developed in C, C++, or Java and contained up to 2.2 million SLOC (Source Lines of Code). Many of these systems included similar design elements that have a negative effect on quality properties.

Figure 1 shows the identified reference architecture for the grid. A grid system is composed of four subsystem types: *Application*, *Collective*, *Resource*, and

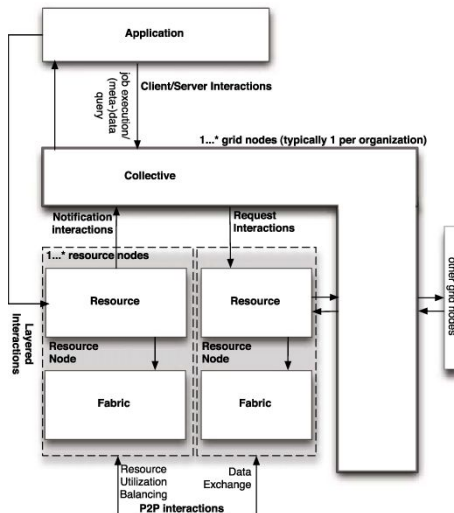


Fig. 1. Structural View of the Grid Reference Architecture

Fabric. Each subsystem type is usually instantiated multiple times. An Application can be any client that needs grid services and is able to use an API that interfaces with Collective or Resource components. The components in the Collective subsystem are used to orchestrate and distribute data and grid jobs to the various available resources in a manner consistent with the security and trust policies specified by the institutions within a grid system (i.e., the virtual organization). The Resource subsystem contains components that perform individual operations required by a grid system by leveraging available lower-level Fabric components. Fabric components offer access capabilities to computational and data resources on an individual node (e.g., access to file-system operations). Each subsystem type uses different interaction mechanisms to communicate with other subsystems types, as noted in Figure 1. The interaction mechanisms are described in [5].

4.2 MIDAS Architecture Refactoring

In collaboration with an industrial partner, for the last three years we have been developing a lightweight middleware platform, called MIDAS, for distributed sensor applications [19] [20]. Over ten software engineers in three geographically distributed locations contributed to MIDAS in multiple development cycles to address changing and growing requirements. In its current version, MIDAS implements many high-level services (e.g., transparent fault-tolerance through component replication) that were not anticipated at the commencement of the project. Additionally, MIDAS was ported to a new operating system (Linux) and programming language (C++), and capabilities tailored for a new domain (mobile robotics) were added. As a consequence, the MIDAS architecture was forced to evolve in unanticipated ways, and the system’s complexity grew substantially. In its current version, the MIDAS middleware platform consists of approximately

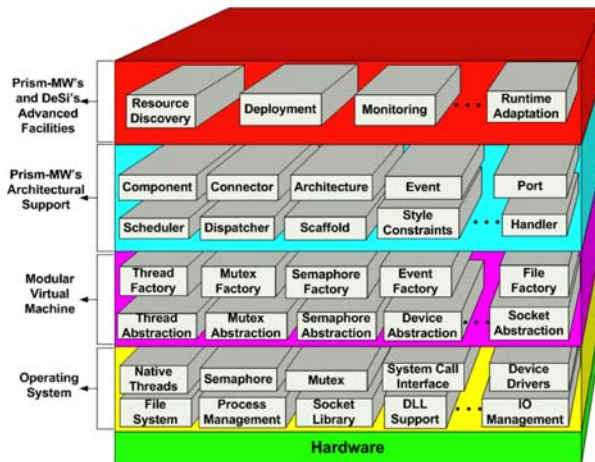


Fig. 2. System Stack Layers in MIDAS

100 KSLOC in C++ and Java. The iterative development of MIDAS eventually caused several architectural elements to lose conceptual coherence (e.g., by providing multiple services). As a consequence, we recently spent three person-months refactoring the system to achieve better modularity, understandability, and adaptability. While performing the refactoring, we again encountered architectural structures that negatively affected system lifecycle properties.

Figure 2 shows a layered view of the MIDAS middleware platform. The bottom of the MIDAS architecture is a virtual machine layer that allows the middleware to be deployed on heterogeneous OS and hardware platforms efficiently. The host abstraction facilities provided by the virtual machine are leveraged by the middleware’s architectural constructs at the layer above. These architectural constructs enable a software organization to directly map its system’s architecture to the system’s implementation. Finally, these constructs are used to implement advanced distributed services such as fault-tolerance and resource discovery.

4.3 Studies from Research Literature

Given the above experiences, we examined the work in architectural recovery and refactoring published in research literature [4] [21] [22] [23], which helped us to understand architectural design challenges and common bad smells. In this paper, we refer to examples from a case study that extracted and analyzed the architecture of Linux [4]. In this study, Bowman et al. created a conceptual architecture of the Linux kernel based on available documentation and then extracted the architectural dependencies within the kernel source code (800 KSLOC). They concluded that the kernel contained a number of design problems, such as unnecessary and unintended dependencies.

5 Architectural Smells

This section describes four architectural smells in detail. We define each architectural smell in terms of participating architectural elements — components, connectors, interfaces, and configurations. Components are computational elements that implement application functionality in a software system [24]. Connectors provide application-independent interaction facilities, such as transfer of data and control [25]. Interfaces are the interaction points between components and connectors. Finally, configurations represent the set of associations and relationships between components and/or connectors. We provide a generic schematic view of each smell captured in one or more UML diagrams. Architects can use diagrams such as these to inspect their own designs for architectural smells.

5.1 Connector Envy

Description. Components with *Connector Envy* encompass extensive interaction-related functionality that should be delegated to a connector. Connectors

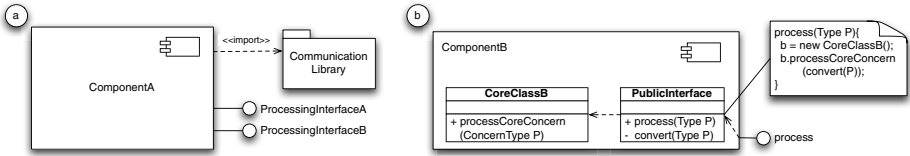


Fig. 3. The top diagram depicts Connector Envy involving communication and facilitation services. The bottom diagram shows Connector Envy involving a conversion service.

provide the following types of interaction services: communication, coordination, conversion, and facilitation [25]. Communication concerns the transfer of data (e.g., messages, computational results, etc.) between architectural elements. Coordination concerns the transfer of control (e.g., the passing of thread execution) between architectural elements. Conversion is concerned with the translation of differing interaction services between architectural elements (e.g., conversion of data formats, types, protocols, etc). Facilitation describes the mediation, optimization, and streamlining of interaction (e.g., load balancing, monitoring, and fault tolerance). Components that extensively utilize functionality from one or more of these four categories suffer from the Connector Envy smell.

Figure 3a shows a schematic view of one Connector Envy smell, where *ComponentA* implements communication and facilitation services. *ComponentA* imports a communication library, which implies that it manages the low-level networking facilities used to implement remote communication. The naming, delivery and routing services handled by remote communication are a type of facilitation service.

Figure 3b depicts another Connector Envy smell, where *ComponentB* performs a conversion as part of its processing. The interface of *ComponentB* called *process* is implemented by the *PublicInterface* class of *ComponentB*. *PublicInterface* implements its *process* method by calling a conversion method that transforms a parameter of type *Type* into a *ConcernType*.

Quality Impact and Trade-offs. Coupling connector capabilities with component functionality reduces reusability, understandability, and testability. Reusability is reduced by the creation of dependencies between interaction services and application-specific services, which make it difficult to reuse either type of service without including the other. The overall understandability of the component decreases because disparate concerns are commingled. Lastly, testability is affected by Connector Envy because application functionality and interaction functionality cannot be separately tested. If a test fails, either the application logic or the interaction mechanism could be the source of the error.

As an example, consider a *MapDisplay* component that draws a map of the route followed by a robot through its environment. The component expects position data to arrive as Cartesian coordinates and converts that data to a screen coordinate system that uses only positive *x* and *y* values. The *MapDisplay* suffers

from Connector Envy because it performs conversion of data formats between the robot controller and the user interface. If the *MapDisplay* is used in a new, simulated robot whose controller represents the world in screen coordinates, the conversion mechanism becomes superfluous, yet the *MapDisplay* cannot be reused intact without it. Errors in the displayed location of the robot could arise from incorrect data conversion or some other part of the *MapDisplay*, yet the encapsulation of the adapter within the *MapDisplay* makes it difficult to test and verify in isolation.

The Connector Envy smell may be acceptable when performance is of higher priority than maintainability. More specifically, explicitly separating the interaction mechanism from the application-specific code creates an extra level of indirection. In some cases, it may also require the creation of additional threads or processes. Highly resource-constrained applications that use simple interaction mechanisms without rich semantics may benefit from retaining this smell. However, making such a trade-off simply for efficiency reasons, without considering the maintainability implications of the smell, can have a disastrous cumulative effect as multiple incompatible connector types are placed within multiple components that are used in the same system.

Example from Industrial Systems. The Gfarm Filesystem Daemon (*gfsd*) from a grid technology called Grid Datafarm [26] is a concrete example of a component with Connector Envy that follows the form described in Figure 3. The *gfsd* is a Resource component and runs on a Resource node as depicted in Figure 1. The *gfsd* imports a library that is used to build the lightweight remote procedure call (RPC) mechanism within the *gfsd*. This built-in RPC mechanism provides no interfaces to other components and, thus, is used solely by the *gfsd*. While the general schematic in Figure 3 shows only an instance of communication and facilitation, this instance of the smell also introduces coordination services by implementing a procedure call mechanism. The interfaces of the *gfsd* provide remote file operations, file replication, user authentication and node resource status monitoring. These interfaces and the *gfsd*'s RPC mechanism enable the notification, request, and P2P interactions shown in Figure 1 that occur across Resource nodes in Grid Datafarm.

Reusability, modifiability, and understandability are adversely affected by the Connector Envy smell in the *gfsd*. The reusability effects of Connector Envy can be seen in a situation where a new Resource component, called Gfarm workflow system daemon (*gwsd*), that provides workflow-based services is added to Grid Datafarm. The RPC mechanism within the *gfsd* is built without interfaces that can be made available to other components, hence the RPC mechanism cannot be used with the *gwsd*. Understandability is reduced by the unnecessary dependencies between the *gfsd*'s application-specific functionality (e.g., file replication, local file operations, etc.) and RPC mechanism. The combination of application-specific functionality and interaction mechanisms throughout the functions of the *gfsd* enlarge the component in terms of function size, number of functions, and shared variables. Both modifiability and understandability are

adversely affected by having the overwhelming majority of the *gfsd*'s functions involve the use or construction of Grid Datafarm's RPC mechanism.

It is possible that since grid technologies need to be efficient, the creators of Grid Datafarm may have intentionally built a *gfsd* with Connector Envy in order to avoid the performance effects of the indirection required for a fully separated connector. Another fact to consider is that Grid Datafarm has been in use for at least seven years and has undergone a significant number of updates that have expanded the *gfsd*'s functionality. This has likely resulted in further commingling of connector-functionality with application-specific functionality.

5.2 Scattered Parasitic Functionality

Description. *Scattered Parasitic Functionality* describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. This smell violates the principle of separation of concerns in two ways. First, this smell scatters a single concern across multiple components. Secondly, at least one component addresses multiple orthogonal concerns. In other words, the scattered concern infects a component with another orthogonal concern, akin to a parasite. Combining all components involved creates a large component that encompasses orthogonal concerns. Scattered Parasitic Functionality may be caused by cross-cutting concerns that are not addressed properly. Note that, while similar on the surface, this architectural smell differs from the *shotgun surgery* code smell [2] because the code smell is agnostic to orthogonal concerns.

Figure 4 depicts three components that are each responsible for the same high-level concern called *SharedConcern*, while *ComponentB* and *ComponentC* are responsible for orthogonal concerns. The three components in Figure 4 cannot be combined without creating a component that deals with more than one clearly-defined concern. *ComponentB* and *ComponentC* violate the principle of separation of concerns since they are both responsible for multiple orthogonal concerns.

Quality Impact and Trade-offs. The Scattered Parasitic Functionality smell adversely affects modifiability, understandability, testability, and reusability. Using the concrete illustration from Figure 4, modifiability, testability, and understandability of the system are reduced because when *SharedConcern* needs to be

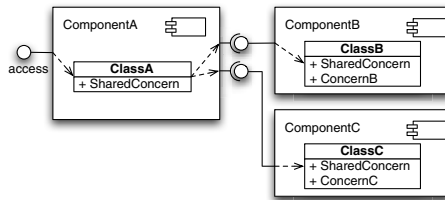


Fig. 4. The Scattered Parasitic Functionality occurring across three components

changed, there are three possible places where *SharedConcern* can be updated and tested. Another facet reducing understandability is that both *ComponentB* and *ComponentC* also deal with orthogonal concerns. Designers cannot reuse the implementation of *SharedConcern* depicted in Figure 4 without using all three components in the figure.

One situation where scattered functionality is acceptable is when the *SharedConcern* needs to be provided by multiple off-the-shelf (OTS) components whose internals are not available for modification.

Example from Industrial Systems. Bowman et al.'s study [4] illustrates an occurrence of Scattered Parasitic Functionality in the widely used Linux operating system. The case study reveals that Linux's status reporting of execution processes is actually implemented throughout the kernel, even though Linux's conceptual architecture indicates that status reporting should be implemented in the PROC file system component. Consequently, the status reporting functionality is scattered across components in the system. This instance of the smell resulted in two unintended dependencies on the PROC file system, namely, the Network Interface and Process Scheduler components became dependent on the PROC file system.

The PROC file system example suffers from the same diminished lifecycle properties as the notional system described in the schematic in Figure 4. Modifiability and testability are reduced because updates to status reporting functionality result in multiple places throughout the kernel that can be tested or changed. Furthermore, understandability is decreased by the additional associations created by Scattered Parasitic Functionality among components.

The developers of Linux may have implemented the operating system in this manner since status reporting of different components may be assigned to each one of those components. Although it may at first glance make sense to distribute such functionality across components, more maintainable solutions exist, such as implementing a monitoring connector to exchange status reporting data or creating an aspect [27] for status reporting.

5.3 Ambiguous Interfaces

Description. *Ambiguous Interfaces* are interfaces that offer only a single, general entry-point into a component. This smell appears especially in event-based publish-subscribe systems, where interactions are not explicitly modeled and multiple components exchange event messages via a shared event bus. In this class of systems, Ambiguous Interfaces undermine static dependency analysis for determining execution flows among the components. They also appear in systems where components use general types such as strings or integers to perform dynamic dispatch. Unlike other constructs that reduce static analyzability, such as function pointers and polymorphism, Ambiguous Interfaces are not programming language constructs; rather, Ambiguous Interfaces reduce static analyzability at the architectural level and can occur independently of the implementation-level constructs that realize them.

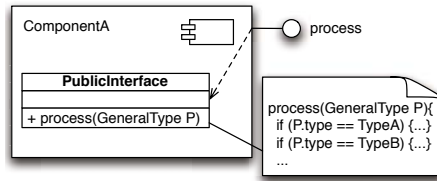


Fig. 5. An Ambiguous Interface is implemented using a single public method with a generic type as a parameter

Two criteria define the Ambiguous Interface smell depicted in Figure 5. First, an Ambiguous Interface offers only one public service or method, although its component offers and processes multiple services. The component accepts all invocation requests through this single entry-point and internally dispatches to other services or methods. Second, since the interface only offers one entry-point, the accepted type is consequently overly general. Therefore, a component implementing this interface claims to handle more types of parameters than it will actually process by accepting the parameter P of generic type *GeneralType*. The decision whether the component filters or accepts an incoming event is part of the component implementation and usually hidden to other elements in the system.

Quality Impact and Trade-offs. Ambiguous Interfaces reduce a system’s analyzability and understandability because an Ambiguous Interface does not reveal which services a component is offering. A user of this component has to inspect the component’s implementation before using its services. Additionally, in an event-based system, Ambiguous Interfaces cause a static analysis to over-generalize potential dependencies. They indicate that all subscribers attached to an event bus are dependent on all publishers attached to that same bus. Therefore, the system seems to be more widely coupled than what is actually manifested at run-time. Even though systems utilizing the event-based style typically have Ambiguous Interfaces, components utilizing direct invocation may also suffer from Ambiguous Interfaces. Although dependencies between these components are statically recoverable, the particular service being invoked by the calling component may not be if the called component contains a single interface that is an entry point to multiple services.

The following example helps to illustrate the negative effect of the wide coupling. Consider an event-based system containing n components, where all components are connected to a shared event bus. Each component can publish events and subscribes to all events. A change to one publisher service of a component could impact $(n - 1)$ components, since all components appear to be subscribed to the event, even if they immediately discard this event. A more precise interface would increase understandability by narrowing the number of possible subscribers to the publishing service. Continuing with the above example, if each component would list its detailed subscriptions, a maintenance engineer could see which m components ($m \leq n$) would be affected by changing the specific

publisher service. Therefore, the engineer would only have to inspect the change effect on m components instead of $n - 1$. Often times, components exchange events in long interactions sequences; in these cases, the Ambiguous Interface smell forces an architect to repeatedly determine component dependencies for each step in the interaction sequence.

Example from Industrial Systems. A significant number of event-based middleware systems suffer from the form of Ambiguous Interface smell depicted in Figure 5. An example of a widely used system that follows this design is the Java Messaging Service (JMS) [28]. Consumers in JMS receive generic Message objects through a single *receive* method. The message objects are typically cast to specific message types before any one of them is to be processed. Another event-based system that acts in this manner is the *Information Bus* [29]. In this system, publishers mark the events they send with subjects and consumers can subscribe to a particular subject. Consumers may subscribe to events using a partially specified subject or through wild-cards, which encourage programmers to subscribe to more events than they actually process.

The event-based mechanism used by MIDAS conforms to the diagram in Figure 5. In the manner described above, MIDAS is able to easily achieve dynamic adaptation. Through the use of DLLs, MIDAS can add, remove, and replace components during run-time, even in a highly resource-constrained sensor network system. As mentioned in Section 4.2, we have recently spent three person-months refactoring the system to achieve better modularity, understandability, and adaptability. During the refactoring, determining dependencies and causality of events in the system was difficult due to the issues of over-generalized potential dependencies described above. An extensive amount of recovery needed to be done to determine which dependencies occur in what context.

5.4 Extraneous Adjacent Connector

Description. The *Extraneous Adjacent Connector* smell occurs when two connectors of different types are used to link a pair of components. Eight types of connectors have been identified and classified in the literature [25]. In this paper, we focus primarily on the impact of combining two particular types of connectors, procedure call and event connectors, but this smell applies to other connector types as well. Figure 6 shows a schematic view of two components that communicate using both a procedure call connector and an event-based connector.

In an event-based communication model, components transmit messages, called events, to other components asynchronously and possibly anonymously. In Figure 6, *ComponentA* and *ComponentB* communicate by sending events to the *SoftwareEventBus*, which dispatches the event to the recipient. Procedure calls transfer data and control through the direct invocation of a service interface provided by a component. As shown in Figure 6, an object of type *ClassB* in *ComponentB* communicates with *ComponentA* using a direct method call.

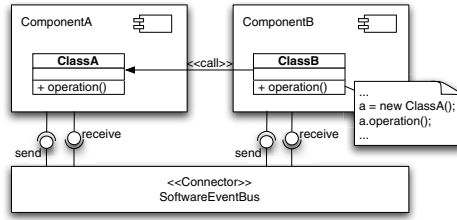


Fig. 6. The connector `SoftwareEventBus` is accompanied by a direct method invocation between two components

Quality Impact and Trade-offs. An architect’s choice of connector types may affect particular lifecycle properties. For example, procedure calls have a positive affect on understandability, since direct method invocations make the transfer of control explicit and, as a result, control dependencies become easily traceable. On the other hand, event connectors increase reusability and adaptability because senders and receivers of events are usually unaware of each other and, therefore, can more easily be replaced or updated. However, having two architectural elements that communicate over different connector types in parallel carries the danger that the beneficial effects of each individual connector may cancel each other out.

While method calls increase understandability, using an additional event-based connector reduces this benefit because it is unclear whether and under what circumstances additional communication occurs between *ComponentA* and *ComponentB*. For example, it is not evident whether *ComponentA* functionality needs to invoke services in *ComponentB*. Furthermore, while an event connector can enforce an ordered delivery of events (e.g., using a FIFO policy), the procedure call might bypass this ordering. Consequently, understandability is affected, because a software maintenance engineer has to consider the (often unforeseen and even unforeseeable) side effects the connector types may have on one another.

On the other hand, the direct method invocation potentially cancels the positive impact of the event connector on adaptability and reusability. In cases where only an event connector is used, components can be replaced during system runtime or redeployed onto different hosts. In the scenario in Figure 6, *ComponentA*’s implementation cannot be replaced, moved or updated during runtime without invalidating the direct reference *ComponentB* has on *ClassA*.

This smell may be acceptable in certain cases. For example, standalone desktop applications often use both connector types to handle user input via a GUI. In these cases, event connectors are not used for adaptability benefits, but to enable asynchronous handling of GUI events from the user.

Example from Industrial Systems. In the MIDAS system, shown in Figure 2, the primary method of communication is through event-based connectors provided by the underlying architectural framework. All high-level services of MIDAS, such as resource discovery and fault-tolerance were also

implemented using event-based communication. While refactoring as described in Section 4.2, we observed an instance of the Extraneous Adjacent Connector smell. We identified that the Service Discovery Engine, which contains resource discovery logic, was directly accessing the Service Registry component using procedure calls. During the refactoring an additional event-based connector for routing had to be placed between these two components, because the Fault Tolerance Engine, which contains the fault tolerance logic, also needed access to the Service Registry. However, the existing procedure call connector increased the coupling between those two components and prevented dynamic adaptation of both components.

This smell was accidentally introduced in MIDAS to solve another challenge encountered during the implementation. In the original design, the Service Discovery Engine was broadcasting its events to all attached connectors. One of these connectors enabled the Service Discovery Engine to access peers over a UDP/IP network. This instance of the Extraneous Adjacent Connector smell was introduced so that the Service Discovery Engine could directly access the Service Registry, avoiding unnecessary network traffic. However, as discussed, the introduced smell instance caused the adaptability of the system to decrease.

6 Conclusion

Code smells have helped developers identify when and where source code needs to be refactored [2]. Analogously, architectural smells tell architects when and where to refactor their architectures. Architectural smells manifest themselves as violations of traditional software engineering principles, such as isolation of change and separation of concerns, but they go beyond these general principles by providing specific repeatable forms that have the potential to be automatically detected. The notion of architectural smells can be applied to large, complex systems by revealing opportunities for smaller, local changes within the architecture that cumulatively add up to improved system quality. Therefore, architects can use the concept (and emerging catalogue) of smells to analyze the most relevant parts of an architecture without needing to deal with the intractability of analyzing the system as a whole.

Future work on architectural smells includes a categorization of architectural smells, architectural smell detection and correction processes, and tool support to aid in those processes. A categorization of architectural smells would include an extensive list of smells and an analysis of the impact, origins, and ways to correct the smells. Architectural smells may be captured in an architectural description language, which would allow conceptual architectures to be analyzed for smells before they are implemented. Correction of smells would include the inception of a set of architectural refactoring operations and the provision of tools to help recommend particular operations for detected smells. In attempting to repair architectures of widely-used systems, the authors of [23] identified a set of operations that can be used as a starting point for determining a complete set of architectural refactoring operations. By trying to correct some of the

architectural smells we found in both our own and others' experiences, such as [4] [21] [22] [23], we hope to identify other architectural refactoring operations and determine which operations are relevant to particular smells.

Acknowledgments

This material is based upon work sponsored by Bosch RTC. The work was also sponsored by the National Science Foundation under Grant numbers ITR-0312780 and SRS-0820170.

References

1. Mens, T., Tourwe, T.: A survey of software refactoring. *IEEE Transactions on Software Engineering* (January 2004)
2. Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Reading (1999)
3. Garcia, J., Daniel Popescu, G.E., Medvidovic, N.: Identifying Architectural Bad Smells. In: *13th European Conference on Software Maintenance and Reengineering* (2009)
4. Bowman, I., Holt, R., Brewster, N.: Linux as a case study: its extracted software architecture. In: *Proc. of the 21st International Conference on Software Engineering* (1999)
5. Mattmann, C.A., Garcia, J., Krka, I., Popescu, D., Medvidovic, N.: The anatomy and physiology of the grid revisited. Technical Report USC-CSSE-2008-820, Univ. of Southern California (2008)
6. Taylor, R., Medvidovic, N., Dashofy, E.: *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Chichester (2008)
7. Lippert, M., Roock, S.: *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, Chichester (2006)
8. Shaw, M., Garlan, D.: *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River (1996)
9. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: *Proc. of the 20th IEEE International Conference on Software Maintenance* (2004)
10. Lozano, A., Wermelinger, M., Nuseibeh, B.: Assessing the impact of bad smells using historical information. In: *9th International Workshop on Principles of Software Evolution* (2007)
11. Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J., Wiley, J., Sons, I.: *AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York (1998)
12. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: *Proc. of the 17th International Conference on Software Engineering* (1995)
13. Gacek, C.: *Detecting Architectural Mismatches During Systems Composition*. PhD thesis, Univ. of Southern California (1998)
14. Abd-Allah, A.: *Composing heterogeneous software architectures*. PhD thesis, University of Southern California (1996)

15. Roshandel, R.: Calculating architectural reliability via modeling and analysis. In: Proc. of the 26th International Conference on Software Engineering (2004)
16. Leveson, N.G.: *Safeware: System Safety and Computers*. Addison-Wesley, Reading (1995)
17. Foster, I., et al.: The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 15(3) (2001)
18. Mattmann, C., Medvidovic, N., Ramirez, P., Jakobac, V.: Unlocking the Grid. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) *CBSE 2005. LNCS, vol. 3489*, pp. 322–336. Springer, Heidelberg (2005)
19. Malek, S., Seo, C., Ravula, S., Petrus, B., Medvidovic, N.: Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In: Proc. of the 29th International Conference on Software Engineering (2007)
20. Seo, C., Malek, S., Edwards, G., Popescu, D., Medvidovic, N., Petrus, B., Ravula, S.: Exploring the role of software architecture in dynamic and fault tolerant pervasive systems. In: *International Workshop on Software Engineering for Pervasive Computing Applications, Systems and Environments* (2007)
21. Godfrey, M.W., Lee, E.H.S.: Secrets from the monster: Extracting mozilla’s software architecture. In: Proc. of the Second International Symposium on Constructing Software Engineering Tools (2000)
22. Gröne, B., Knöpfel, A., Kugel, R.: Architecture recovery of apache 1.3 – a case study. In: Proc. of the International Conference on Software Engineering Research and Practice 2002 (2002)
23. Tran, J., Godfrey, M., Lee, E., Holt, R.: Architectural repair of open source software. In: *8th International Workshop on Program Comprehension* (2000)
24. Shaw, M., et al.: Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering* (1995)
25. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proc. of the 22nd International Conference on Software Engineering (2000)
26. Tatebe, O., Morita, Y., Matsuoka, S., Soda, N., Sekiguchi, S.: Grid datafarm architecture for petascale data intensive computing. In: Proc. of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (2002)
27. Kiczales, G., Hilsdale, E.: *Aspect-Oriented Programming*. Springer, Heidelberg (2003)
28. Haase, K.: *Java message service tutorial* (2002)
29. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus: an architecture for extensible distributed systems. In: Proc. of the 14th ACM Symposium on Operating Systems Principles (1994)

On the Consolidation of Data-Centers with Performance Constraints

Jonatha Anselmi, Paolo Cremonesi, and Edoardo Amaldi

Politecnico di Milano, DEI
Via Ponzio 34/5, I-20133 Milan, Italy
jonatha.anselmi@polimi.it

Abstract. We address the *data-center consolidation* problem: given a working data-center, the goal of the problem is to choose which software applications must be deployed on which servers in order to minimize the number of servers to use while avoiding the overloading of system resources and satisfying availability constraints. This in order to tradeoff between quality of service issues and data-center costs. The problem is approached through a robust model of the data-center which exploits queueing networks theory. Then, we propose two mixed integer linear programming formulations of the problem able to capture novel aspects such as workload partitioning (load-balancing) and availability issues. A simple heuristic is proposed to compute solutions in a short time. Experimental results illustrate the impact of our approach with respect to a real-world consolidation project.

1 Introduction

As the complexity of information technology (IT) infrastructures increases due to mergers or acquisitions, new challenging problems arise in the design and management of the resulting computer systems. In fact, they are often costly, non-flexible, yielding under-utilized servers and energy wastings. To reduce conflicts among the offered services, many enterprise data-centers host most of their services on dedicated servers without taking into account the possibility of deploying multiple services on a single server. Therefore, many servers are not used at their maximum capabilities and, in turn, expensive hardware investments are often required. Nowadays companies search for IT solutions able to significantly drop data-centers costs, e.g., energy consumption, space costs, and obtain a flexible system satisfying customer demand.

In this framework, the *consolidation* of data-center resources is a current solution adopted by many industries. The objective of consolidation problems is to reduce the complexity of computer systems while guaranteeing some performance and availability constraints. This is usually achieved by searching for the best mapping between software applications and servers which minimizes data-center costs. The main issues taken into account by a data-center consolidation solution are i) to reduce the complexity of the IT infrastructure, ii) to increase system performance, iii) to obtain a flexible system, iv) to reduce data-center costs, and

v) to improve operational efficiencies of business processes. The current challenge characterizing a consolidation is finding an optimal allocation of services to target servers able to meet the above issues while satisfying performance and availability requirements. This results in new capacity planning problems which we address in this work. An important example of performance index is given by the response time, i.e., the time interval between the submission of a job into a system and its receipt. Such variable is strictly related to the servers utilizations, i.e., the proportion of time in which a server is used. In fact, the higher the servers utilizations, the higher the resulting response time. Therefore, constraints on the maximum utilization of each server are important to

1. avoid the saturation of physical resources letting the system handle unexpected workload peaks,
2. guarantee a low sensitivity of data-center response time in front of small workload variations (it is well-known, e.g., [6], that the response time curve grows to infinity according to a hyperbole when the utilization of a server approaches unity),
3. increase the data-center reliability because if a failure occurs on a server, then the associated applications can be moved on different servers preventing a drastic growth of data-center response time.

Constraints on the maximum response time are very often used in many applications for ensuring some quality of service (see, e.g., [4],[1]). The solution of such problem is aimed to yield a data-center *configuration* able to satisfy the above issues while minimizing costs.

1.1 Related Work

During the last decades, the resource management problem has been analyzed in depth by many researchers in many frameworks and several works are available in the literature. However, little appeared in the literature for the recent problem of data-center consolidation with performance constraints even though it attracted the attention of many IT companies.

Rolia et al. [12] analyze the consolidation problem with a dynamic approach taking into account the workloads seasonal behavior to estimate the server demands. Their consolidation problem limits the overall utilization of each server and it assumed that each application is deployed on a single server. An integer linear program formulation and a genetic algorithm are proposed to solve the problem and a case study with 41 servers is presented. Bichler et al. [3] present a similar dynamic approach tailored for virtualized systems. The main difference of their approach is that the optimization problem is solved exploiting multi-dimensional bin-packing approximate algorithms [8]. In the context of virtualized servers, an example of how to consolidate servers is also shown in [9],[10]. Our previous work [1] tackles the data-center consolidation problem exploiting queueing networks theory. Linear and non-linear optimization problems are provided as well as accurate and efficient heuristics. However, such work is essentially based on the assumption that one software application must

be deployed on exactly one server. Furthermore, it is assumed that the *service demands* of the queueing networks model are known.

1.2 Our Contribution

In the present work, we again tackle the data-center consolidation through an optimization problem extending our previous work [1]. The proposed formulation now takes into account the capability to handle the workload partitioning (or load-balancing) of applications, i.e., the fact that one application can be deployed on many servers. This is clearly related to availability issues. The estimates of performance indices are again obtained by exploiting queueing networks (QN) theory (see, e.g., [6]) because it provides versatile and robust models for predicting performance. However, this is achieved through a new, innovative approach. In fact, the standard theory underlying QN models assumes that a number of input parameters, e.g., arrival rates and service demands, must be known in advance to obtain performance estimates. Unfortunately, in practice these parameters can be very difficult to obtain for a variety of reasons (see, e.g., [7]). Therefore, we adopt a new, robust methodology to estimate performance which is only based on the observable variables which are usually easy to measure. As a matter of fact, in real-world scenarios a working infrastructure exists before starting a performance evaluation and a measurement phase can be carried out. In the context of IT systems, common experience reveals that server utilizations and *speed-ups* possess such requirements. Our analysis assumes the knowledge of only these two latter parameters, i.e., standard input parameters such as arrival rates and service times are not part of our approach. We then propose a number of linear optimization models related to the data-center consolidation. Given that the computational effort needed by standard exact solution algorithm is expensive, an heuristic is shown to efficiently solve the optimization problems in an approximate manner. The computational effort and the accuracy of such heuristic is evaluated with respect to a real-world consolidation project with 38 servers and 311 web applications. We then present several minor extensions of practical interest to the above issues, e.g., the case in which applications require storage.

This work is organized as follows. In Section 2, we discuss the parameters characterizing the data center and define the associated QN model. In Section 3, we present our main formulation of the consolidation problem proposing a heuristic for its efficient solution. Section 4 is devoted to experimental results on a real-world consolidation project. Finally, Section 5 draws the conclusions of our work and outlines further research.

2 Data-Center Queueing Network Model

2.1 Data-Center Description

The data center is composed of M heterogeneous servers. The cost of using server j , which comprises energy consumption, maintainability costs, etc., is denoted by c_j , $j = 1, \dots, M$.

The *speed-up* of server j is denoted by ρ_j and it is understood as its relative processing capacity obtained by the execution of suitable benchmarks with respect to a reference server (say server 1), i.e., the ratio between the processing speeds of server j and 1.

The data center hosts R different applications (or services) and each application is deployed on multiple tiers (e.g., web-server tier, application-server tier, etc.). Application r sequentially spans L_r tiers, $r = 1, \dots, R$, and when an application r job (or client) joins the data center, it initially executes tier 1 on some server, then it proceeds to tier 2 and so on till the L_r -th. For application r jobs, when the L_r -th tier is reached, the request is forwarded back to the $(L_r - 1)$ -th for some further processing and so on till the first one. It is well-known that this behavior agrees with standard multi-tiered architectures. We denote by

$$L = \sum_{r=1}^R L_r \quad (1)$$

the total number of application tiers. More than one application tier can be deployed on a given server and each tier of each application can be deployed on multiple servers.

The deployment of a given application on multiple tiers is usually referred to as *vertical scalability* and it is important to provide a better performance handling larger workloads and to solve possible conflicts among different layers (different application tiers may use different technologies). On the other hand, the deployment of a given application tier on multiple servers is usually referred to as *horizontal scalability* and lets us deal with load-balancing issues. The horizontal scalability is also important to guarantee availability constraints: in fact, if a given application tier is deployed on multiple servers, then a failure on a single server does not prevent the availability of the application because the workload can be rearranged among the available servers.

To reduce management costs and to increase the data-center availability, we assume that each application tier must be deployed on a number of servers ranging between two fixed values. Therefore, we denote by $m_{r,l}$ and $n_{r,l}$, respectively, the maximum and the minimum number of servers in which tier l of application r must be deployed.

Another source of lack of data-center availability is the deployment of several tiers on a same server. Therefore, we assume that a maximum number of v_j application tiers can be deployed on server j . This assumption is also meant to avoid the modeling of non-negligible overheads in service times estimates (usually referred to as *virtualization overhead*) which would be introduced by the middleware management if the number of virtual machines running on a single server is large.

In agreement with the notation of basic queueing networks theory [6], we denote by $D_{j,r,l}$ the mean *service demand* (time units) required by a job executing tier l of application r on server j when the network contains no other job.

If not specified, indices j , r and l will implicitly range, respectively, in sets $\{1, \dots, M\}$, $\{1, \dots, R\}$ and $\{1, \dots, L_r\}$ indexing servers, applications and tiers.

2.2 QN Model

QN models are a popular tool for evaluating the performance of computer systems and, in the mathematical formulation of the data-center consolidation problem, they let us deal with simple analytical expressions of performance indices. The class of queueing network models we consider goes beyond the popular class of product-form (or separable) queueing networks [2],[6],[11]. In fact, we consider those queueing networks satisfying the utilization law, e.g., [6] (it is well-known that this is a much larger class). This is simply due to the fact that the performance indices we consider are server utilizations only. Therefore, this lets our approach rely on wide assumptions and be widely applicable and robust.

Since the data center hosts different applications (characterized by different service demands) and an arriving job can execute only one of them, the model we build is *multiclass*. For convenience, a job requesting the execution of application r is referred to as a class- r job. Since the number of jobs populating the data center is not constant, the model we build is *open* and we denote by λ_r the mean workload (arrival rate) of class- r jobs, $r = 1, \dots, R$. Jobs circulate in the network visiting a number of stations and eventually leave the network. The stations of the QN model the data center servers and, in the following, we use the term *station* when we refer to the QN and the term *server* when we refer to the data-center.

Let $D_{j,r}$ be the mean *service demand* [6] of class- r jobs at station j , i.e., the total average time required by a class- r job to station j during the execution of all its tiers and when the network contains no other job. Within this standard definition, we underline that the service demands include the processing times of jobs at servers when they visit stations passing from the first tier to the last one and returning back from the last tier to the first one. This notion of service demand also takes into account that it is possible to deploy more tiers of a given application on the same server. For instance, assuming that only tiers from 1 to $l_r \leq L_r$ of application r are deployed on server j , we have

$$D_{j,r} = \sum_{l=1}^{l_r} D_{j,r,l}. \quad (2)$$

The time interval needed by a server to transfer a job to an other server is assumed to be negligible.

Within this queueing network model of the data center, we recall that the average utilization of station j due to class- r jobs, i.e., the *busy* time proportion of server j due to class- r jobs, is given by

$$U_{j,r} = U_{j,r}(\lambda_r) = \lambda_r D_{j,r}. \quad (3)$$

Formula (3) is known as the *utilization law* [6]. Clearly, the total average utilization of server j is given by

$$U_j = U_j(\lambda_1, \dots, \lambda_R) = \sum_{r=1}^R U_{j,r} < 1. \quad (4)$$

Since we deal with the *averages* of utilizations, when referring to an index we will drop the word average.

We now show a simple example to illustrate the queuing network model underlying the data center. Let us consider the case of two applications, i.e., $R = 2$, having both three tiers, i.e., $L_1 = L_2 = 3$ and $M = 5$ available servers, and let us also suppose that the application tiers are deployed on the servers as indicated in Table I. For instance, we have that tier 2 of application 2 is deployed on server 3. We notice that server 5 is not used. Since each tier of each application is deployed on exactly one server, all service demands are given by the sum of service times as in (2).

Table 1. Deployment scheme of the example

Tier	Class 1	Class 2
1	1	2
2	1	3
3	2	4

The QN model underlying the deployment scheme of Table I is such that the stations service demands are given by Table II.

Table 2. Service demands of the deployment scheme in Table I

Station	Class 1	Class 2
1	$D_{1,1,1} + D_{1,1,2}$	0
2	$D_{2,1,3}$	$D_{2,2,1}$
3	0	$D_{3,2,2}$
4	0	$D_{4,2,3}$
5	0	0

Within this example, each application tier is deployed on a single server and server 5 is not used. We also note that the QN model of the data-center does not explicitly take into account the notion of tier which is embedded in the notion of service demands.

Within the definition of speed-up given in Section 2.1, the following relation must hold

$$\frac{D_{i,r,l}}{\rho_i} = \frac{D_{j,r,l}}{\rho_j} \quad (5)$$

for all i, j, r, l , which implies

$$\frac{D_{i,r}}{\rho_i} = \frac{D_{j,r}}{\rho_j}. \quad (6)$$

3 Formulation and Algorithm

The objective of the data-center consolidation problem is to exploit the available servers in order to obtain a *configuration* able to satisfy, *in the average*, performance constraints on utilizations and data-center response times while minimizing the sum of servers costs.

The decision variables we include in our optimization models are

$$x_{j,r,l} = \begin{cases} 1 & \text{if tier } l \text{ of application } r \text{ is deployed} \\ & \text{on server } j, \\ 0 & \text{otherwise,} \end{cases} \quad (7)$$

$$y_j = \begin{cases} 1 & \text{if server } j \text{ is used} \\ 0 & \text{otherwise,} \end{cases} \quad (8)$$

and

$$z_{j,r,l} \geq 0 \quad (9)$$

denoting the *proportion* of application r and tier l workload assigned to server j .

Let a *configuration* be a possible assignment of variables $x_{j,r,l}$ satisfying the issues discussed in Section 2.1, i.e., a *feasible* deployment scheme. A configuration can be interpreted as a function f mapping tier l of application r to a subset of \mathbf{M} , i.e., a subset of the set of stations. The goal of the optimization problem is to find the configuration of minimum cost which satisfies constraints on server utilizations and constraints on data-center *structural* properties such as the fact that each application tier must be deployed on at least $n_{r,l}$ and at most $m_{r,l}$ servers. We refer to this latter property as *workload partitioning* and it is an innovative aspect of our formulation. The deployment of an application tier on multiple servers is known to increase its availability.

We assume that the data-center has an initial configuration f and that the per-class utilizations of such configuration are known. This reflects a common real-world scenario because in practice a data-center consolidation is performed on a working infrastructure and, within this framework, server utilizations are usually easy to measure and robust. Therefore, we assume the knowledge of per-class utilizations $U_{f(r,l),r,l}$ for all r and l . Considering (3) and (5), we have

$$U_{j,r,l} = \frac{\rho_j}{\rho_{f(r,l)}} U_{f(r,l),r,l}, \quad \forall j, r, l \quad (10)$$

which expresses, in a robust manner, the per-class utilization of tier l of application r if it would be deployed on server j as a function of measured data and known parameters.

3.1 Formulation of the Consolidation Problem

Let \hat{U}_j denote the value of the maximum utilization that server j is allowed to have. We formulate the consolidation problem through the following ILP problem

$$\mathcal{P} : \min \sum_{j=1}^M c_j y_j \tag{11}$$

subject to:

$$\sum_{j=1}^M z_{j,r,l} = 1, \quad \forall r, l \tag{12}$$

$$\sum_{r=1}^R \sum_{l=1}^{L_r} U_{f(r,l),r,l} \frac{\rho_j}{\rho_{f(r,l)}} z_{j,r,l} \leq \hat{U}_j y_j, \quad \forall j \tag{13}$$

$$z_{j,r,l} \geq \frac{x_{j,r,l}}{m_{r,l}}, \quad \forall j, r, l \tag{14}$$

$$z_{j,r,l} \leq \frac{x_{j,r,l}}{n_{r,l}}, \quad \forall j, r, l \tag{15}$$

$$\sum_{r=1}^R \sum_{l=1}^{L_r} x_{j,r,l} \leq v_j, \quad \forall j \tag{16}$$

$$z_{j,r,l} \geq 0, \quad \forall j, r, l \tag{17}$$

$$x_{j,r,l} \in \{0, 1\}, \quad \forall j, r, l \tag{18}$$

$$y_j \in \{0, 1\}, \quad \forall j \tag{19}$$

Clearly, the objective function (11) minimizes the weighted sum of server costs.

Constraints (12) ensure that variable $z_{j,r,l}$ represents proportions of the workload of tier l of application r to forward to server j .

Constraints (13) limit the overall utilization of j by means of relation (10).

Constraints (14) and (15) model, respectively, the fact that the workload of tier l of application r must be allocated on at most $m_{r,l}$ and at least $n_{r,l}$ servers. These constraints ensure the avoidance of very *unbalanced* workloads which may yield situations where most of the workload of an application tier is assigned to a particular server (this is ensured by (14)), and the avoidance of splitting the workload among a very large number of servers which may result in maintainability cost and inefficiencies (this is ensured by (15)). Both constraints (14) and (15) imply that $x_{j,r,l} = 1$ if and only if $z_{j,r,l} > 0$. This can be easily seen if we rewrite (14) and (15) as follows

$$\frac{x_{j,r,l}}{m_{r,l}} \leq z_{j,r,l} \leq \frac{x_{j,r,l}}{n_{r,l}}, \quad \forall j, r, l \tag{20}$$

where we see that if $x_{j,r,l} = 0$ (respectively $x_{j,r,l} = 1$) then $z_{j,r,l}$ is forced to be zero (strictly positive).

Finally, constraints (16) limit the number of application tiers to deploy on j .

3.2 Heuristic Solution

The number of binary variables adopted by \mathcal{P} is $M + ML$. Since large-scale data-centers are composed of hundreds of servers and applications, i.e., several thousands of variables $x_{j,r,l}$, the exact solution of \mathcal{P} through standard techniques (e.g., branch and cut) requires a strong computational effort. Therefore, we now provide a simple heuristic aiming to find a good solution in a shorter time.

The heuristic we propose initially guesses the set of servers which yields the configuration of minimum cost and, with respect to this set only, checks whether or not a feasible configuration exists. If such configuration does not exist, then the guess is iteratively refined by adding the *best* server until a feasible solution is found.

Algorithm 1 is the heuristic we propose for the efficient solution of \mathcal{P} .

Algorithm 1. Heuristic solution for \mathcal{P}

- 1: Solve the relaxation of \mathcal{P} when $x_{j,r,l}$ are continuous between 0 and 1, and y_j are binary;
 - 2: $Y := \{j \in \{1, \dots, M\} : y_j = 1\}$;
 - 3: $\tilde{Y} := Y$;
 - 4: **for** $k = 1, \dots, M - |Y|$ **do**
 - 5: Let \mathcal{P}' be problem \mathcal{P} where
 - the objective function (11) is removed,
 - variables y_j are fixed to 1 for all $j \in \tilde{Y}$, and
 - variables y_j and $x_{j,r,l}$, for all $r, l, j \notin \tilde{Y}$ are removed;
 - 6: Solve \mathcal{P}' ;
 - 7: **if** a feasible solution of \mathcal{P}' exists **then**
 - 8: **break**;
 - 9: **end if**
 - 10: Let \mathcal{P}'' be problem \mathcal{P} where variables
 - y_j are fixed to 1 for all $j \in Y$, and
 - $x_{j,r,l}$ are binary if $j \in Y$, otherwise continuous between 0 and 1,
 and the following constraint is included

$$\sum_{j=1}^M y_j \leq |Y| + k; \tag{21}$$
 - 11: Solve \mathcal{P}'' ;
 - 12: $\tilde{Y} := \{j \in \{1, \dots, M\} : y_j = 1\}$;
 - 13: **end for**
 - 14: **return** variables $x_{j,r,l}$;
-

We initially solve \mathcal{P} assuming that variables $x_{j,r,l}$ are continuous. Therefore, the number of binary variables drops from $M + ML$ to M . The optimum of this

problem requires a significantly smaller computational effort and the optimal configuration found must be a lower bound on the configuration of minimum cost. We note that a feasible solution of this problem always exists because we assumed that the data-center initially has a working configuration. Then, we define set Y as the set of servers chosen by the optimal configuration of the relaxed problem and \mathcal{P}' which takes into account the servers belonging to Y only. \mathcal{P}' is thus composed of much fewer variables and constraints than \mathcal{P} . We then search for a feasible solution of problem \mathcal{P}' (Line 4). If this problem is feasible then a solution is found and the algorithm ends. Otherwise, through problem \mathcal{P}'' we augment the space of feasible solutions by adding to Y a server not included in the configuration computed by the relaxed problem in Line 1. Then, we iteratively check for the feasibility of \mathcal{P}' until a feasible configuration exists. We remark that such configuration eventually exists because we initially assume a working configuration.

Given that the optimum of the relaxed problem defined in Line 1 is a lower bound on the solution of \mathcal{P} , if the condition in the loop holds at its first evaluation, then Algorithm 1 provides the optimum. In general, if n is the number of iterations performed by the algorithm, then n is an upper bound on the difference between the number of servers identified by the optimal solution of \mathcal{P} and by the proposed heuristic. This holds because we add a server to Y at each iteration and because the objective function value corresponding to the optimal configuration of \mathcal{P} cannot be less than the one obtained in the relaxation of Line 1.

3.3 Minor Extensions

We now propose minor extensions of practical interest related to the formulation above.

1. Consider the case in which tiers l_1, \dots, l_K of application r_1 must be deployed on single but different servers, which implies $n_{r_1, l_1} = m_{r_1, l_1} = n_{r_1, l_2} = \dots = m_{r_1, l_K} = 1$. This need can be due to operating systems incompatibilities, e.g., Windows software on Linux servers. In this case, the constraint is given by

$$\sum_{k=1}^K z_{j, r_1, l_k} \leq 1, \quad \forall j. \quad (22)$$

We note that (22) is expressed in terms of continuous variables z_{j, r_1, l_k} (instead of x_{j, r_1, l_k}). It is known that this yields a more efficient formulation. Analogously, we can avoid the deployment of particular application tiers on some servers by simply imposing $z_{j, r, l} = 0$ for some j , r and l .

2. Consider the opposite case where tiers l_1, \dots, l_K of application r_1 must be deployed on the *same* (single) server, which implies $n_{r_1, l_1} = m_{r_1, l_1} = n_{r_1, l_2} = \dots = m_{r_1, l_K} = 1$. In this case, we add the constraints

$$\begin{aligned}
z_{j,r_1,l_1} &= z_{j,r_1,l_2}, & \forall j \\
z_{j,r_1,l_2} &= z_{j,r_1,l_3}, & \forall j \\
&\dots \\
z_{j,r_1,l_{K-1}} &= z_{j,r_1,l_K}, & \forall j.
\end{aligned} \tag{23}$$

3. In many practical cases, some applications must be deployed only on a given subset of servers. This situation can arise for security issues where some critical applications must be deployed in virtual private networks. Let S denote the subset of set $\{1, \dots, M\}$ containing the indices of the data-center servers which are able to execute the tiers of application r_1 . In this case, the constraints are given by

$$x_{j,r_1,l} = 0, \forall j \notin S, t, \tag{24}$$

which reduce the size of the problem because many binary variables become constants.

4 Experimental Results

In this section, we present experimental results in order to evaluate the accuracy and the computational requirements of our approach. Experimental analyses have been performed by running the Ilog Cplex v10.0.0 optimization solver on a 2.80GHz Intel Xeon CPU with hyperthreading technology. Algorithm 1 has been implemented in the AMPL language [5].

We apply Algorithm 1 to a real consolidation project within the data-center of one of the largest European telecommunication companies. The portion of the data-center involved in the consolidation project consists of 311 single-tier applications running on 311 dedicated servers. The applications were originally consolidated with a manual mapping between the applications and 38 brand-new systems. For each system, the mapping required to keep overall utilization below a 70% threshold. Figure 1 shows the CPU utilizations for the manually consolidated servers. The applications were consolidated using VMWare ESX Server 3.5. The systems used for the consolidation were HP ProLiant BL680c G5 blade servers and ProLiant DL58x servers. Most of the servers have 8 CPUs, but the blade systems have up to 80 processors (see Figure 2). The total computational power of the selected systems exceeds 1.6 THz.

We applied Algorithm 1 to the above data-center configuration in order to find a better consolidation strategy. Before running the algorithm, systems and applications have been monitored for a one-month period in order to measure, for each application, the average CPU utilization. Moreover, for each server, configuration information have been collected, describing processing power (MHz) and number of CPUs. Such metrics have been used to derive the relative speed-ups between systems. Algorithm 1 has been applied by varying the utilization thresholds in the range between 0.3 and 0.7, with step 0.1. In Figure 3.a, we show the number of servers identified by our approach. When the target maximum server utilization of 0.7 is considered, we show that it is possible to obtain

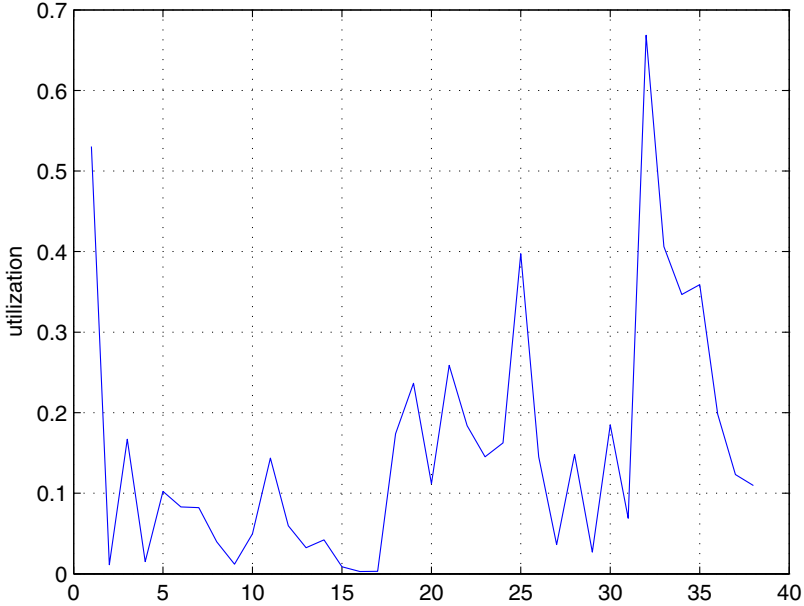


Fig. 1. Utilizations of the initial configuration

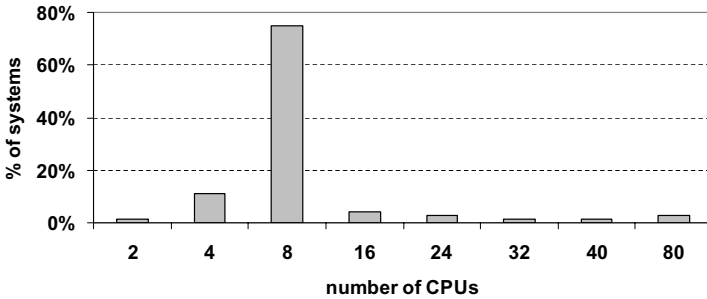


Fig. 2. Distribution of the number of CPUs of the servers

a configuration which adopts only 6 servers. With respect to the 38 servers chosen by the initial configuration, this has a drastic impact on data-center costs. We notice that the number of servers identified by our approach decreases as the maximum server utilization increases. This is obviously due to the fact that more applications can be deployed on a single server as its maximum utilization increases. In all cases, the number of iterations performed in the loop of the algorithm was zero. This implies that an optimal configuration has been always found. With our heuristic, all experiments terminated within 3 seconds. This because the relaxation in Line 1 of Algorithm 1 identifies a very small set of

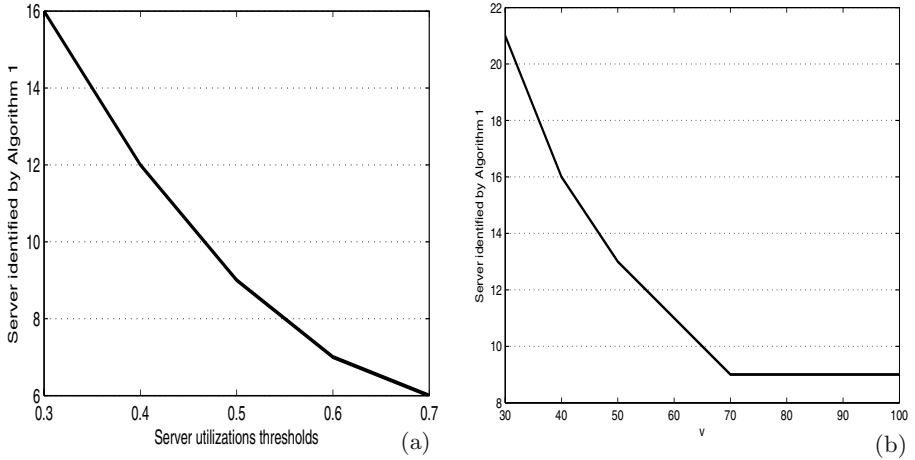


Fig. 3. Number of servers identified by the proposed heuristic by varying the utilization thresholds (on the left). Number of servers identified by the proposed heuristic assuming $n_{r,t} = 2$ and $m_{r,t} = 4$ and varying v (on the right).

servers which significantly yields to reduce the total number of binary variables $x_{j,r,l}$.

We now consider the case where each application must be deployed on at least 2 and at most 4 servers. Assuming 0.7 as maximum utilization thresholds, we vary the maximum number of applications to deploy on a given server from 30 to 100 with step 10 and show the number of servers identified by Algorithm 1 (see Figure 3.b). Even in this case, the number of iterations performed in the loop of the algorithm was zero. In the figure, we see the price we have to pay for load-balancing applications among multiple servers. In fact, in this case the optimal configuration is composed of 9 servers.

5 Conclusions

In this paper, we addressed the problem of finding an *optimal* data-center configuration able to satisfy performance and availability constraints. Recently, this problem received a lot of attention by industries. We built a queueing network model of the data-center and imposed constraints on server utilizations, a critical parameter strictly related to data-center stability. Then, we tackled the problem as an optimization problem and proposed new mixed integer linear programming formulations able to take into account innovative aspects. These include the possibility of deploying a given software applications on a number of servers between two given thresholds in a controlled, load-balanced manner. Given that the computational effort needed by standard exact solution algorithm is expensive, an heuristic is proposed to efficiently solve the optimization problem in an

approximate manner. The approach is robust because servers utilizations are derived without taking into account the standard input parameters characterizing queueing models, e.g., arrival rates and service demands. In fact, the expressions of server utilizations have been obtained within the observable variables which, in data-centers, are usually easy to measure and robust. Experimental results on a real consolidation project revealed that the heuristic is able to compute optimal configuration in very short time. We leave as future work the extension of our formulation which takes into account resources *profiles*, i.e., the possibility of having different workload demands at different time intervals.

References

1. Anselmi, J., Amaldi, E., Cremonesi, P.: Service consolidation with end-to-end response time constraints. In: Software Engineering and Advanced Applications, 2008. SEAA 2008. 34th Euromicro Conference, September 3-5, pp. 345–352 (2008)
2. Baskett, F., Chandy, K., Muntz, R., Palacios, F.: Open, closed, and mixed networks of queues with different classes of customers. *J. ACM* 22(2), 248–260 (1975)
3. Bichler, M., Setzer, T., Speitkamp, B.: Capacity planning for virtualized servers. In: Proceedings of the Workshop on Information Technologies and Systems, Milwaukee, Wisconsin, USA (2006)
4. Cardellini, V., Casalicchio, E., Grassi, V., Mirandola, R.: A framework for optimal service selection in broker-based architectures with multiple qos classes. In: SCW 2006: Proceedings of the IEEE Services Computing Workshops, pp. 105–112. IEEE Computer Society, Washington (2006)
5. Fourer, R., Gay, D.M., Kernighan, B.W.: *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press (November 2002)
6. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River (1984)
7. Liu, Z., Wynter, L., Xia, C.H., Zhang, F.: Parameter inference of queueing models for it systems using end-to-end measurements. *Perform. Eval.* 63(1), 36–60 (2006)
8. Martello, S., Pisinger, D., Toth, P.: *New trends in exact algorithms for the 0-1 knapsack problem* (1997)
9. Menasce, D.A.: *Virtualization: Concepts, applications, and performance modeling. The volgenau school of information technology and engineering* (2005)
10. Menasce, D.A., Almeida, V.A.F., Dowdy, L.W.: *Performance by Design: Computer Capacity Planning by Example: Computer Capacity Planning*. Prentice Hall International, Englewood Cliffs
11. Menasce, D.A., Dowdy, L.W., Almeida, V.A.F.: *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River (2004)
12. Rolia, J., Andrzejak, A., Arlitt, M.F.: Automating enterprise application placement in resource utilities. In: Brunner, M., Keller, A. (eds.) *DSOM 2003*. LNCS, vol. 2867, pp. 118–129. Springer, Heidelberg (2003)

Evolving Industrial Software Architectures into a Software Product Line: A Case Study

Heiko Kozirolek, Roland Weiss, and Jens Doppelhamer

ABB Corporate Research, Industrial Software Systems,
Wallstadter Str. 59, 68526 Ladenburg, Germany
{heiko.kozirolek,roland.weiss,jens.doppelhamer}@de.abb.com

Abstract. Industrial software applications have high requirements on performance, availability, and maintainability. Additionally, diverse application landscapes of large corporate companies require systematic planning for reuse, which can be fostered by a software product-line approach. Analyses at the software architecture level can help improving the structure of the systems to account for extra-functional requirements and reuse. This paper reports a case study of product-line development for ABB's robotics PC software. We analysed the software architectures of three existing robotics applications and identified their core assets. As a result, we designed a new product-line architecture, which targets at fulfilling various extra-functional requirements. This paper describes experiences and lessons learned during the project.

1 Introduction

The high requirements for extra-functional properties, such as performance, availability, and maintainability, in industrial software applications demand carefully designed software architectures. Industrial software applications control complex machines and have to respond to user requests or other external stimuli within strict time constraints to avoid failures and harm to human beings. They have to exhibit a high availability with very limited down-time to provide maximal benefit for customers. Internally, they should be structured to allow for efficient maintenance and long-term evolution. All these features should be enforced by the underlying software architecture.

Large corporate companies, which serve multiple application domains, have to deal with diverse software application landscapes that complicate fulfilling all extra-functional requirements. In our context, we analyzed the situation for the robotics software at ABB. There are more than 100 software applications in the robotics domain from ABB. These applications have been developed by distributed development teams with limited centralized planning and coordination. This situation has accounted for a high functional overlap in the applications, which has led to high and unnecessary development and maintenance costs.

A common solution for this problem is the introduction of a software product-line [1], which systematically targets at bundling common assets and building customized applications from reusable software components. Many companies,

such as Nokia, Philips, and Bosch have successfully introduced software product lines. While several studies have been reported (e.g. [2,3,4]), which aim at deriving product-line architecture from existing software application, no cookbook solution can be applied for industrial software applications so far.

In this paper, we report our experiences from 3 year running project at ABB Research reconstructing and evolving the software architectures of three robotics PC applications from ABB. We analyzed the different applications for their shared functionalities and special advantages. We identified core assets and bundled common functionality into reusable software components. We designed new interfaces and ultimately developed a software product-line architecture to systematize reuse among the applications. During the course of the project, we learned several lessons, which could be interesting both for other software architects and researchers.

The contribution of this paper is a case study on architecture evolution and software product-line design in the industrial application domain. The case study includes experiences and findings, which could stimulate further research. We used and assessed different methods from research for the benefits in our domain.

This paper is organized as follows: Section 2 reports on a survey of ABB robotics software, which revealed a significant functional overlap and little reuse. The application domain and the three applications we analyzed are described in more detail in Section 3. Section 4 elaborates on the three phases of our architecture evolution and product-line development project. Section 5 summarizes our lessons learned, and Section 6 surveys related work. Finally, Section 7 concludes the paper and sketches future work.

2 The Challenge: Functional Overlap

A comprehensive survey on ABB's robotics software was conducted in 2006 and motivated our project. The software application landscape within ABB Robotics is diverse and scattered with over 120 applications developed in 8 different countries (mainly Sweden and Norway) by 10 different organizations. The software supports a large number of robot application domains, such as arc welding, spot welding, press automation, painting, sealing, material handling, etc.

The used programming languages include C, C++, C#, VB, and JavaScript. Furthermore, a Pascal-like imperative language called RAPID is used by many applications for implementing robot programming logic for ABB's main robot controller called IRC5. Several applications target the Windows operating system, while other applications run directly on robot controllers using the VxWorks real-time operating system. The code complexity of the applications ranges from small tools with 1 KLOC to large applications with more than 600 KLOC.

The survey analysed 58 ABB robotics applications in detail. Fig. 1 shows a high-level overview of the application landscape. The 58 applications depend on 13 base elements, which provide functionality for example for remote communication and graphical user interfaces. The applications themselves provide different extension interfaces to allow user-specific customization. However, apart from

the base elements there is very few reuse among the applications as depicted by the low number of dependencies between the applications in Fig. 1.

Therefore, the survey broke down the functionalities of the applications in detail and categorized them into 30 different functions. Each application developer group was asked what functionality their tool implemented. Fig. 2 shows a condensed view of the results. The left-hand side depicts the number of applications implementing the same function. For example, function 1 was implemented repeatedly in 11 different applications.

It could be argued that the low amount of reuse results from the distinct robot application domains, where software applications are implemented without regard of other application domains. Therefore, the right-hand side of Fig. 2 shows the number of functions, which were implemented multiple times within a single application domain. For example, in domain 1 developers have implemented 161 functions multiple times in different applications.

The low level of reuse among the applications contributes to the high maintenance costs for the applications, which the survey found to be in the range of several million US-dollars per year. As expected, the most complex applications have the highest maintenance costs. However, the survey also identified some outlier applications with unproportionally high maintenance costs despite a small amount of code.

There are several *reasons* for the undesirable functional overlap within ABB Robotics applications. The organisational structure of the software development units has no central unit coordinating and organizing systematic reuse. Several company acquisitions into the corporate body have contributed to the situation. The software is created by a large number of development teams, sometimes

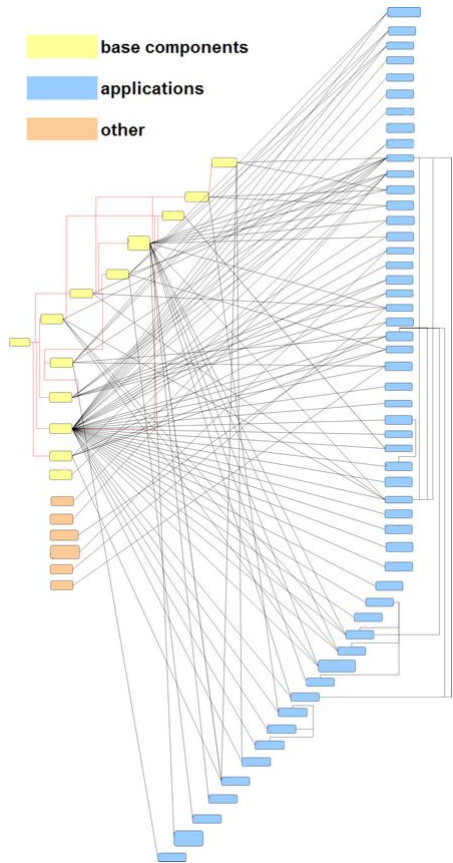


Fig. 1. Application Landscape of ABB's Robotics Software (schematic view, anonymised)

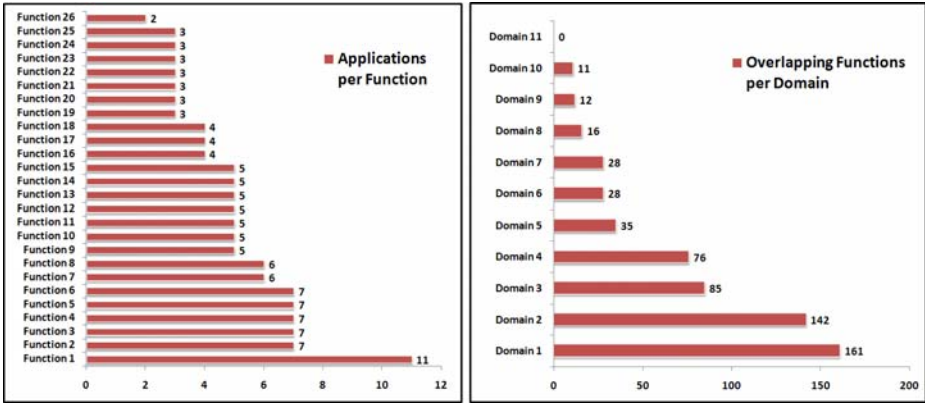


Fig. 2. Functional Overlap in ABB Robotics Software

consisting only of 2-3 developers. The communication among the teams is limited within and across application domains and organizational units.

Because of the large size of the application landscape, no developer can follow which applications are developed in other units and where reuse potential might be high. Some applications are small and tailored for very specific functions and thus exhibit limited reuse potential. Other applications are large, but their software architecture is documented only in a limited way, so that it is not easy to isolate certain functions in the code.

The amount of functional overlap bears high potential for sharing code (i.e., reusable components) among the applications. To decrease maintenance costs and time-to-market, the survey suggests to bundle more common functionality into reusable software components (e.g., COM components or .NET components). More communication among the development units is needed and a central organization for planning systematic reuse would be desirable.

Notice that the survey did not involve all ABB robotics software, therefore the potential for reuse might be even higher. We believe that this situation of functional overlap is not specific to ABB, but common for large corporate companies, which serve different applications domains and rely on distributed development teams. More research should be devoted to documenting, analysing, and redesigning complex application landscapes (cf. [5]).

3 Systems under Study

With the challenge of functional overlap in mind, we started an architecture redesign project in 2006 focussing on ABB robotics PC applications. Applications running on embedded devices were out of scope for our project. This section first briefly describes the robotics PC application domain (Section 3.1) to let the reader understand the extra-functional requirements for these systems. Then, it sketches the high-level software architectures of three PC applications, which

were the basis for our project (Section 3.2), and lists the extra functional requirements (Section 3.3).

3.1 Application Domain

Fig. 3 depicts a typical industrial robot system. It may involve a single robot for a specific task or a whole robot line consisting of several robots arranged in subsequent order.

One or several robot controllers handle movements of the robot arms. This involves path planning and robot axis control. Mustapic et al. [6] have detailed on the open platform architecture of ABB's robot controller. It is an embedded system consisting of ca. 2500 KLOC in C divided into 400-500 classes. The controller kernel provides special support for implementing application specific extensions. The robot controller has an attached teach pendant, a small handheld device for manual control and supervision of a robot. The robot controller and its extensions typically run on an embedded operating system such as VxWorks or Windows CE.

The robot system can include a number of external sensors, such as cameras for scanning items to be processed or automatic scales for weighting items. Multiple conveyor belts may feed the robots with items and forward the processed items. In larger robot systems, operators supervise the whole process supported by an arbitrary number of PC workstations, which for example visualize scanned items or allow manipulating robot configurations.

The following coarse functionalities are carried out by PC applications in such a robot system:

- **Engineering:** deals with offline robot programming on an office PC, which is decoupled from robot production. It allows configuring and preparing robot programs in advance without interfering with robot production. Modern robot engineering tools offer 3D visualizations of robots systems to assist path planning.

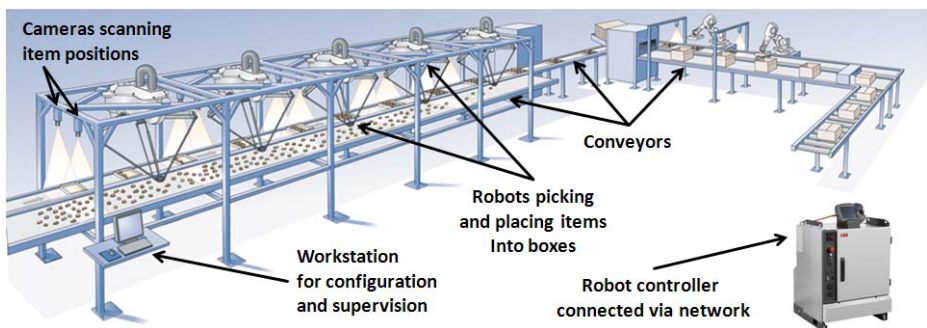


Fig. 3. Exemplary Industrial Robotics System for Packaging: Overview

- **Simulation:** allows testing the robot installations and programs (i.e., RAPID code) created during engineering. This functionality is similar to program debugging, as it allows to set break points and execute the programs step-by-step. It targets identifying robot collisions and analysing robot reachability.
- **Supervision:** lets operators monitor and control the robot system. This includes consolidating logs from different robot controllers and visualizing the data collected from different robot controllers.
- **Job Control:** manages a job queue and controls the execution of jobs from the queue. A job captures a task the robot system shall execute, for example painting a car or picking and placing a certain amount of goods. Controlling the job queue may involve simply executing RAPID code on the robot controllers, or, in more complex cases, collecting and analysing data from external sensors and distributing item coordinates to different robot controllers.
- **Job Coordination:** coordinates jobs running on multiple job controllers during production. Job coordination for example allows synchronizing different jobs in a robot system, so that subsequent jobs execute with minimal delay, or switching jobs on multiple controllers in a coordinated way (e.g., a new color for painting the next car has been chosen and all involved robots have to adjust accordingly).

Additionally, attached programmable logic controllers (PLC) are used for coordinating the robot system within the context of superordinated systems. For example, information from ERP systems or other production systems can be used to direct robot execution.

3.2 Initial Architectures

We analyzed three different PC applications, each one targeting a specific application domain. These applications were chosen because of their considerable value to ABB business and their perceived similarities. The picking/packing application supports high speed packing of small goods. The painting application for the automotive industry supports colouring cars. The palletizing application supports piling and unpling goods onto pallets. The three applications have been implemented by different development teams and only exhibit limited reuse among each other.

Fig. 4 shows the high-level software architectures of the three applications in a component-and-connector view. The functionalities described in Section 3.1 have been implemented differently in the the different applications.

The picking/packing application combines engineering, supervision, and job control in a single software tool. In this case, the job control functions involve scanning captured camera images for item positions and distributing the item positions to different robot controllers. The tool is a typical Win32 application with a graphical user interface. It communicates with plant interaction controllers via a remote interface and with the robot controllers via the controller API. For this application, the robot controller features a special picking extension. Simulation is not supported for this type of application.

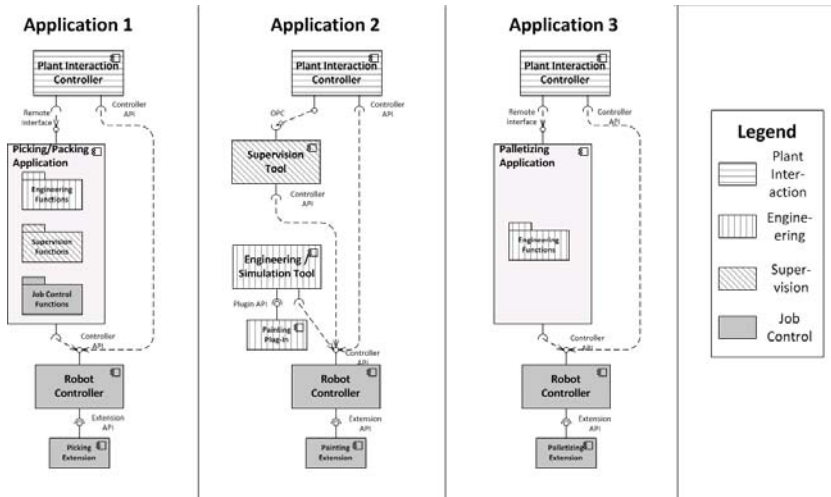


Fig. 4. High-Level Software Architecture for three ABB Robotics PC Applications

The painting application includes two distinct tools for supervision and engineering/simulation. It does not have additional job control functionality. The engineering and simulation tool has an attached painting plug-in, which tailors the tools for the application domain. The supervision tool communicates with the plant interaction controller via OPC DA (OLE for Process Control Data Access). The supervision tool features a rich and customizable graphical user interface. Additionally, there is a controller extension with special support for painting applications.

The palletizing application mainly provides engineering functionality to set up palletizing jobs. Supervision has to be carried out using the teach pendants or programming the plant interaction controller. There is also no additional job control or simulation functionality. However, there is a robot controller extension specifically for the palletizing domain.

3.3 Extra-Functional Requirements

Designing a quality software architecture for robotics PC applications is challenging because of the high extra-functional requirements:

- **Availability:** Usually, a robot line is part of a larger production process, where a failure of the robots can result in substantial costs. As a particular example, for the picking/packaging application, the job controller functionality must run without interruption. Otherwise, no more targets for the robot controller might be available, which stops the whole production process.
- **Scalability:** Robot systems are sold to a large variety of customers. Some customers operate small robot cells with single controllers and robots, while other customers run large distributed robot lines with dozens of robots. The

architecture must support adapting the application for different installation sizes.

- **Maintainability:** High maintenance costs should be avoided to keep the applications profitable. Redundant maintenance effort for functionality implemented in multiple applications should be avoided at any costs.
- **Time-to-Market:** The applications should be adaptable so that new application domains can be supported in a short amount of time. Therefore, reusability of existing assets for new application domains is highly desirable.
- **Sustainability:** As the robot systems have an expected operation time of more than 10 years, the applications should be ready to cope with technology changes in the future.
- **Security:** Remotely accessible robot systems need user authentications to avoid being compromised.
- **Performance:** Once in production, the picking/packing application has to deliver the coordinate batches to the robot controller in time. If the image analysis takes too long, the conveyor tracking mechanism skips item coordinates, which means that items get not processed. Distributing the coordinate batches onto multiple robots and controllers also happens in real time. Static and dynamic load balancing mechanisms must not slow down the robot controllers so that it cannot handle the timing constraints.
- **Usability:** A common look-and-feel for all ABB Robotics PC applications is desirable so that users can quickly orient themselves when using tools from different application domains.

4 The Solution: Step-Wise Evolution

Our project consisted of three phases: reconstructing and documenting the detailed architecture of the picking/placing application (Section 4.1), designing a new remote interface for communication within the architecture (Section 4.2), and finally, designing a new product-line architecture based on identified reusable components from the architecture reconstruction and also including the new remote interface (Section 4.3).

4.1 Architecture Reconstruction and Documentation

As already indicated in Fig. 4, the picking/placing application was perceived as bundling much functionality with limited separation of concerns, which hampered introducing reuse. Therefore, we analysed the architecture of this application in detail in the first phase. Initially, there was no architectural documentation and only limited design documents. First, we reconstructed the architecture, then we documented it, and finally we made suggestions for improvements.

For *architecture reconstruction*, we looked at the application both externally (running different test cases) and internally (analysing the source code). We browsed the code manually to find out how the coarse functionalities listed in Section 3.1 were spread among the code. Furthermore, we used source code

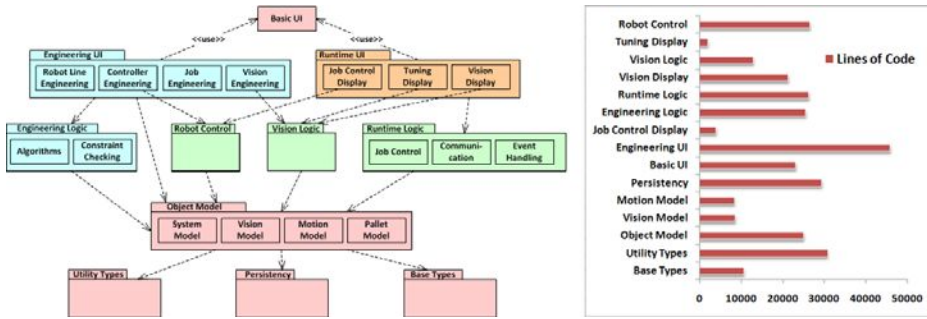


Fig. 5. Picking/Packing Application - Layered Architecture and Code Metrics

analysis tools, such as SourceMonitor [7], Doxygen [8], and SISSy [9] to visualize the static structure and to derive code metrics.

As a result, we recovered a layered architecture as depicted on a high abstraction level in Fig. 5. There are engineering user interfaces and logic as well as runtime user interfaces and logic, the latter including both supervision and and job control functionality. All modules rely on a common object model, which is based on several elementary data types and supports persistency via XML serialization. In total, the application consists of more than 300 KLOC in more than 600 files. Technologically, it is a Win32 application written in C++ with dependencies to Microsoft’s Foundation Classes (MFC) and several third-party components.

While the tools we applied for source code analysis are useful to derive layered structures and bad smells in the code, they provide only limited support for identifying reusable components. Up to now, this is still mainly a manual tasks. Similar tools in this area are Dali, Lattix, or Sotograph. While they can analyse package structures and class dependencies, it is still difficult to locate common functionality spread among multiple packages with these tools. Reverse engineering tools require a more strict software component definition with provided and required interfaces, which are distinct from classes or modules. This way higher level structures could be identified to make components replaceable. A preliminary example for analysing Java code has been presented in [10].

For *architecture documentation*, we used annotated UML diagrams as well as textual descriptions. We used component and package diagrams for a static view on the architecture, as well as sequence diagrams for a dynamic view. Besides the high-level architecture depicted in Fig. 5, we also documented the structure and behaviour on lower levels (e.g., we decomposed included composite components into smaller structures). Our UML models do not directly map to the code structure, but instead visualize higher level structures. In our application domain, UML models are still only used for documentation, not for code generation or analysis of extra-functional properties.

Our suggestions for *architectural improvements* mainly targeted the extra functional requirements modifiability and sustainability. Modifiability requires

isolating separated concerns, so that parts of the application become replaceable. It is an important prerequisite for introducing a software product-line approach. Sustainability is especially critical for industrial software application with life-times often longer than typical IT technology life-times.

To improve *modifiability* and maintainability, we suggested to enforce the layered structure of the architecture more on the code-level. Modifiability tactics [11], such as localizing modifications by maintaining semantic coherence in the different components and layers were presented. We suggested to factor out more base types from the object model, to restructure the central system package from the source code to adhere to the layered structure, and to isolate UI functionality from the object model, which was not fully decoupled from the higher layers. Additionally, the Sissy tool revealed several problem patterns on the design and implementation level, such as dead imports, cyclic dependencies, god classes, permissive visibility of attributes or methods, or violation of data encapsulation.

To improve *sustainability*, we suggested a step-wise migration of the code-base to the .NET framework. Such a technology change shall result in higher developer productivity due to the higher level APIs of the framework and a modern programming language with C#. The application could reuse .NET platform features, such as the frameworks for user interfaces and persistency. Reliability and security shall be improved via type safe code and a new user authentication mechanism. Besides using newer technologies, this change also prepares the application to incorporate third party components off-the-shelf (COTS), as third party vendors are migrating or have already migrated to the new platform. Therefore, we suggested to replace the number of dependencies to the MFC framework with dependencies to the .NET framework to make the application more portable.

4.2 Extending a Remote Interface

The goal of the second phase of the project was to extend the remote interface of the picking/placing application to allow for more application level services, such as tuning sensor parameters during runtime and remote robot control. The existing remoting interface (called RIS) of the application was mainly used by low-level devices, such as PLCs. The new version of the interface should support higher-level systems such as distributed control systems or customer HMIs. Furthermore, it was required that the interface was compliant to interface standards such as OPC, and regulations by the Organization for Machine Automation and Control (OMAC), which for example requires user authentication.

To formulate the functionality provided by the extended interface, we used UML use cases with textual descriptions. Additionally, we used quality attribute scenarios [11] to specify the extra-functional requirements, such as performance, reliability, and security for the extended interface. They describe the source of a scenario, a stimulus initiating the scenario, the artifact touched by the scenario, environmental conditions, as well as expected responses and response measures. Fig. 6 shows an example for a security scenario of the interface.

Scenario name	SS1: Grant only authorized RPS clients access to the system
Overview	In general, a full installation requires RPS clients to authorize themselves. This enforces access to the system according to the user's role.
Source	External RPS client
Stimulus	RPS client tries to perform an action through RPS on the Job Controller
Artifact	RPS authentication service
Environment	Production operation
Response	The authentication service responds to the request by either granting or denying access to the system.
Response Measure	Only authorized clients are serviced, data and operation integrity is preserved.
Priority	Low (for PLCs) – Med (operator panels)

Fig. 6. Security Scenario for the new Remote Interface

We soon realized that we could not incorporate access by low-level devices and high-level systems into a single interface. Therefore, we subsumed the high-level application services in a new interface called Remote Production Services (RPS), and left the old remote interface intact. The new interface was implemented as a web service based on the Windows Communication Framework (WCF). Additionally, it can be provided as an OPC interface. It allows various functionalities, such as controller management, job management, robot management, user management, logging, and parameter hot tuning. The remoting capabilities of RPS allow external clients to access and control the robot system with an interface at the application level, beyond a generic robot controller interface.

The quality attribute scenarios were helpful in discussions with the stakeholder. We got a better understanding of the application domain because of the quality attribute scenarios. Furthermore, the scenarios helped to define the priorities of the stakeholders, as it was discovered that certain extra-functional properties were only secondary to them. As a result, quality attribute scenarios are currently also used for the specification of a new software architecture.

Later, the new remote interface was an important part of the newly designed product-line architecture.

4.3 Designing a Product-Line Architecture

The goal of the third phase of our project was to design a sustainable software product line architecture based on the applications described in Section 3.2. The design incorporated both the architectural documentation of the picking/placing application from phase 1 and the newly designed remote interface RPS from phase 2. The main requirements for the product-line were separating different concerns in the applications, increasing reuse, providing a common look-and-feel, and improving maintainability. Additionally the extra-functional requirements stated in Section 3.3 had to be addressed.

Fig. 7 depicts a high-level static view of the new product-line software architecture incorporating all identified core assets. The three existing applications can be deduced as instances from this product-line. New applications for different

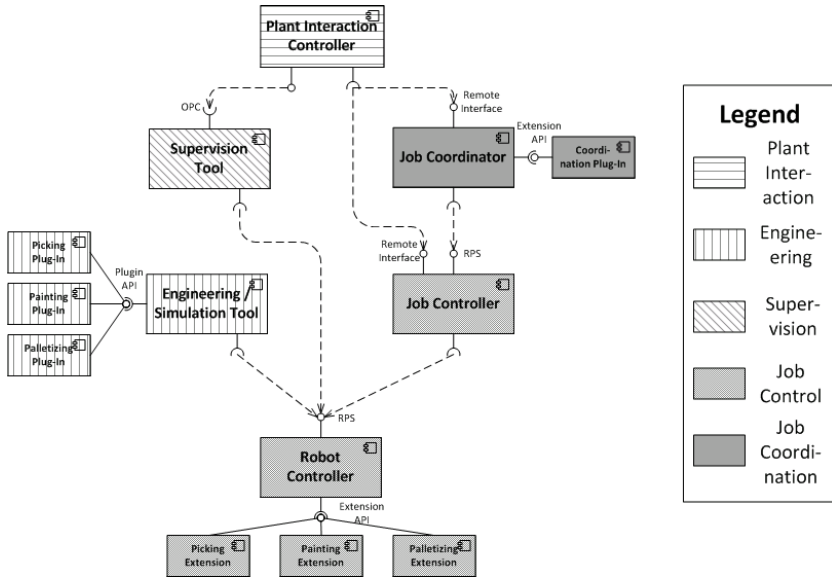


Fig. 7. Product-Line Architecture

domains shall be deducible with limited effort. Common functionality has been bundled. For example, there is a single engineering tool with attached application-specific plugins providing a common look-and-feel for engineering. Furthermore, all applications use the same tool for supervision. The extensions for the robot controller have been left untouched. The architecture enables simulation functionality to become available for different domain-specific applications.

For sensor data analysis and item position generation of picking/placing tasks an additional GUI-less job controller component has been extracted from the former picking/placing application. It is an optional component not present in the instances for painting and palletizing applications. We had to align and compare concepts present in the existing applications. For example one application called a configured and running robot application a "project", another one called it a "job" with slightly different properties. Harmonizing the concepts allowed for more reuse and a common look-and-feel of the applications.

The architecture features a new job coordinator component based on .NET technology. It allows coordinating multiple, possibly concurrently running jobs during production. For some custom robot system installations, this functionality had been implemented using PLCs. Opposed to that, the new job coordinator component is based on .NET technology and shall run on Windows nodes. With a special extension API and the .NET framework, it allows users to easily implement customized coordination logic.

Bundling common functionality into reusable components was enabled by the architecture reconstruction and documentation from phase 1. The engineering layer was replaced by the Engineering/Simulation tools based on .NET

technology, while the supervision layer was replaced by the Supervision tool. We incorporated the new high-level remote interface RPS from phase 2 into the architecture. It provides a new way of accessing the robot controller with high-level services instead of former low-level commands.

The architecture addresses the extra-functional requirements listed in Section 3.3 as follows:

- Availability: For picking/placing applications, the architecture allows for *multiple* job controllers to analyse sensor data and produce item positions for the robot controller. Formerly, this was a single point-of-failure, as a whole robot line had to stop production once the job controller functionality failed.
- Scalability: The architecture can be flexibly adapted for small and large customers. Small systems might not incorporate the optional job coordinator for synchronizing different jobs. Large systems may additionally attach customer HMIs, which can make use of the RPS interface. They may also use multiple job controllers or even run combined picking/placing and palletizing robot lines.
- Maintainability: As common functionality has been bundled into reusable components, the future maintenance effort for the application should decrease. It is no more necessary to test the same functionality multiple times. Critical functionality, such as the picking/placing job controller has been isolated, so that it can be tested more thoroughly. Using common programming languages and frameworks in the architecture is beneficial to distribute the maintenance tasks to different teams.
- Time-to-market: The product-line architecture features an engineering tool and robot controller, which can be quickly adapted to new application domains via plug-ins. New applications do not have to reprogram basic functionality provided by both platforms (e.g., basic user interfaces and robot control logic).
- Sustainability: The product-line architecture features new components based on the .NET framework, which is expected to ease the impact on technology changes over the course of the robot system life-cycle.
- Security: The RPS interface provides services for user authentication to prevent unwanted remote accesses to a robot system.
- Performance: In case of the picking/placing application, concepts for distributing the sensor data analysis and item position generation to multiple job controller instances have been discussed. This should allow to balance the workload on the available hardware better and enable very large robot lines with vast amounts of sensor data, which were formerly difficult to handle.
- Usability: Through the common engineering tool and the common supervision tool, the look-and-feel of the robotics PC applications for engineers and operators is similar across application domains. With the aligned concepts of the different applications, users can quickly learn the new applications. Furthermore, developers are provided with a common remote interface and several extension APIs, which enable user-customizations with limited development effort.

5 Lessons Learned

While the design of the architecture is specific for the robotics domain and ABB, we have learned some general lessons during the course of the projects. These lessons could stimulate further research from the software architecture community and are thus reported in the following.

On the *technical* side, we learned that in our case no common low-level interfaces both for PLC and PC applications could be provided with reasonable effort. Therefore we split the remote interface for the robot applications to a low-level interface (RIS) to be accessed by controllers and a high-level interface (RPS) with different services on the application level to be accessed by DCSs or ERPs.

Reverse engineering techniques to analyse legacy source code could be improved to better identify common functionality spread within the code of an object-oriented application. While existing source code analysis tools are helpful in capturing the structure of a system, they are limited for identifying reusable functionality to be isolated and bundled into components, if this had not been intended by the architecture beforehand.

Unifying some concepts within the different products (e.g., job and robot line concepts) by introducing small changes gave all stakeholders a better understanding of the different application domains. We found that such a step is an important prerequisite when designing a product-line from legacy applications.

From a *methodological* view point, we found quality attribute scenario and attribute-driven design helpful in determining priorities for different extra-functional properties together with the customers. Both methods helped finding focus when designing the RPS interface and the product-line architecture. Furthermore, we found through a business analysis that the benefits from the architecture investigation in terms of saved future development and maintenance costs largely outweigh its costs.

Some *social* aspects could also be learned from the project. During the design of the product-line architecture, we worked closely with the three development teams of the applications. The existing products and known customer installations had a major impact on our PLA design. The stakeholders of the architecture desired to incorporate all relevant product set-ups into the PLA. A survey of existing products and user customizations was essential to ensure stakeholder support in the PLA.

The development teams were initially hesitant towards the redesign of their applications into a PLA. The emotional bindings towards their established products was an obstacle to get their commitment. We resolved their reluctance by getting the different teams into dialogue and emphasizing their individual benefits from the PLA approach (e.g., more focus on core functionality, less maintenance effort).

Development of a PLA should be aligned with the future plans and development cycles of the individual development teams to ensure their support and make the architecture sustainable. With multiple stakeholders having equal rights to the project, the proposal for the architectural design needed more

argumentation and an iterative approach. A champion advocating the benefits of a PLA can speed-up the design process.

6 Related Work

Basic literature on software product-line development has been provided by Clements and Northop [1]. Bass et al. [11] described foundations on software architectures in practise with a special focus on extra-functional requirements. Many industrial case studies as in this paper have been included in the book. Another book on software product-lines has been published by Pohl et al. [12].

In the context of ABB, Mustapic et al. [6] described the software product line architecture of ABB's robotics controller. As described in this paper, it is an open platform, which allows to extend the controller with application specific functionality. The architectural design accounts for fault tolerance and there are methods and tools to assess the real-time properties of different instances of the architecture. Furthermore, Kettu et al. [13] from ABB proposed a holistic approach for architectural analysis of complex industrial software systems, which relies on static and dynamic analysis as well as incorporating documentation, developer interviews, and configuration management data.

In the area of software product line engineering Stoermer et al. [2] presented the MAP approach for mining legacy architecture to derive product lines. O'Brien et al. [3] reports on a case study in this direction using the Dali workbench and reverse engineering techniques to identify components for a product line. Smith et al. [4] describe a method with systematic, architecture-centric means for mining existing components for a product-line.

Numerous software product-lines from the industry have been reported and incorporated into SEI's product-line hall of fame [14]. It includes for example product lines from Bosch for a gasoline system [15] or from Philips for a telecommunication switching system [16]. Hetrick et al. [17] reported on the the incremental return on investment for the transition to a software product-line for Engenio's embedded RAID controller firmware. Deelstra et al. [18] pointed out that deriving individual products from shared software assets in more time-consuming and expensive than expected.

7 Conclusions

Motivated by a survey on ABB robotics software, which found high functional overlap and maintenance costs, we have documented in this paper how we evolved three existing robotics PC applications into a software product line architecture. The PLA addresses various extra-functional properties, such as availability, scalability, performance, and maintainability. The paper has reported several lessons learned from the project, which could stimulate further research.

As next steps, we plan to model the product-line in a formal way and conduct model-driven predictions for extra-functional properties, such as performance, reliability, and maintainability. Creating such models suitable for extrapolating

the extra-functional properties to answer sizing and capacity question requires static as well as dynamic analyses techniques. We will assess whether we can predict the impact of system updates or changes based on the models without implementing these changes. These activities shall be conducted in context of the EU FP7 project Q-IMPRESS [19].

Acknowledgements. We thank Peter Weber from ABB Corporate Research for kindly providing us details about the ABB robotics applications survey. Furthermore, we thank all members of the I1 group at ABB Corporate Research Ladenburg for their valuable review comments, which help improving the quality of this paper.

References

1. Clements, P.C., Northrop, L.: *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Reading (2001)
2. Stoermer, C., O'Brien, L.: MAP - mining architectures for product line evaluations. In: *Proceedings of the First Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, Netherlands, pp. 35–44 (August 2001)
3. O'Brien, L.: *Architecture reconstruction to support a product line effort: Case study*. Technical Report CMU/SEI-2001-TN-015, Software Engineering Institute (SEI), Carnegie Mellon University (CMU) (July 2001)
4. Smith, D.B., Brien, L.O., Bergey, J.: Using the options analysis for reengineering (oar) method for mining components for a product line. In: *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pp. 316–327. Springer, London (2002)
5. Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.P.: *Quasar Enterprise: Anwendungslandschaften service-orientiert gestalten*. Dpunkt-Verlag (2008)
6. Mustapic, G., Andersson, J., Norstroem, C., Wall, A.: A dependable open platform for industrial robotics - a case study. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems II*. LNCS, vol. 3069, pp. 307–329. Springer, Heidelberg (2004)
7. Campwood Software: *SourceMonitor* (January 2009), <http://www.campwoodsw.com>
8. van Heesch, D.: *Doxygen: Source code documentation generator tool*, <http://www.stack.nl/~dimitri/doxygen/>
9. Forschungszentrum Informatik (FZI), Karlsruhe: *SISSy: Structural Investigation of Software Systems* (January 2009), <http://sissy.fzi.de>
10. Chouambe, L., Klatt, B., Krogmann, K.: *Reverse Engineering Software-Models of Component-Based Systems*. In: Kontogiannis, K., Tjortjis, C., Winter, A. (eds.) *12th European Conference on Software Maintenance and Reengineering*, Athens, Greece, April 1–4, pp. 93–102. IEEE Computer Society Press, Los Alamitos (2008)
11. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. SEI Series in Software Engineering. Addison-Wesley, Reading (2003)
12. Pohl, K., Böckle, G., van der Linden, F.J.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, New York (2005)

13. Kettu, T., Kruse, E., Larsson, M., Mustapic, G.: Using architecture analysis to evolve complex industrial systems. In: de Lemos, R., Di Giandomenico, F., Gacek, C., Muccini, H., Vieira, M. (eds.) WADS 2007. LNCS, vol. 5135, pp. 326–341. Springer, Heidelberg (2008)
14. Software Engineering Institute: Product Line Hall of Fame (January 2009), http://www.sei.cmu.edu/productlines/plp_hof.html
15. Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing pla at bosch gasoline systems: Experiences and practices. In: Proc. 3rd Int. Software Product Line Conference (SPLC 2004) (2004)
16. Wijnstra, J.G.: Critical factors for a successful platform-based product family approach. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines, London, UK, pp. 68–89. Springer, Heidelberg (2002)
17. Hetrick, W.A., Krueger, C.W., Moore, J.G.: Incremental return on incremental investment: Engenio’s transition to software product line practice. In: OOPSLA 2006: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 798–804. ACM Press, New York (2006)
18. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. *J. Syst. Softw.* 74(2), 173–194 (2005)
19. Q-Impress Consortium: Q-Impress Project Website, <http://www.q-impress.eu>

Adaptive Application Composition in Quantum Chemistry

Li Li¹, Joseph P. Kenny², Meng-Shiou Wu³, Kevin Huck⁴, Alexander Gaenko³,
Mark S. Gordon³, Curtis L. Janssen², Lois Curfman McInnes¹,
Hirotoishi Mori⁵, Heather M. Netzloff³,
Boyana Norris¹, and Theresa L. Windus³

¹ Argonne National Laboratory, Argonne, IL

² Sandia National Laboratories, Livermore, CA

³ Ames Laboratory, Ames, IA

⁴ University of Oregon, Eugene, OR

⁵ Ochanomizu University, Japan

Abstract. Component interfaces, as advanced by the Common Component Architecture (CCA), enable easy access to complex software packages for high-performance scientific computing. A recent focus has been incorporating support for computational quality of service (CQoS), or the automatic composition, substitution, and dynamic reconfiguration of component applications. Several leading quantum chemistry packages have achieved interoperability by adopting CCA components. Running these computations on diverse computing platforms requires selection among many algorithmic and hardware configuration parameters; typical educated guesses or trial and error can result in unexpectedly low performance. Motivated by the need for faster runtimes and increased productivity for chemists, we present a flexible CQoS approach for quantum chemistry that uses a generic CQoS database component to create a training database with timing results and metadata for a range of calculations. The database then interacts with a chemistry CQoS component and other infrastructure to facilitate adaptive application composition for new calculations.

1 Introduction

As computational science progresses toward ever more realistic multiphysics applications, no single research group can effectively select or tune all components of a given application, and no solution strategy can seamlessly span the entire spectrum of configurations efficiently. Common component interfaces, along with programming language interoperability and dynamic composability, are key features of component technology that enable easy access to suites of independently developed algorithms and implementations. By means of the Common Component Architecture (CCA) [1, 2], such capabilities are now making inroads in scientific computing. The challenge then becomes how to make the best choices for reliability, accuracy, and performance, both when initially composing and configuring a component application, and when dynamically adapting to respond to continuous changes in component requirements and execution environments.

Computational quantum chemistry is a mature domain of scientific computing populated by numerous software packages offering a range of theoretical methods with a variety of implementation approaches. These packages provide a vast array of tools to be employed by practitioners who are often not developers or knowledgeable about the implementation details of these packages. The existence of certain methods as well as their performance on various types of hardware varies greatly within these packages, and the optimal configuration of these calculations is often a matter of trial and error, at least until a great deal of experience is accumulated with each package. For example, as the number of cores increases on commodity processors, memory bandwidth limitations will likely limit the number of cores that can be used effectively per socket to significantly fewer than the number available. Furthermore, predicting runtime can be useful for planning and queue management when running unfamiliar job types.

The challenges of efficiently employing and configuring quantum chemistry packages, faced also in combustion, fusion, and accelerator modeling [3], motivate the design and implementation of generic support for *computational quality of service* (CQoS) [4], or the automatic composition, substitution, and dynamic reconfiguration of components to suit a particular computational purpose and environment. CQoS embodies the familiar concept of quality of service in networking as well as the ability to specify and manage characteristics of the application in a way that adapts to the changing (computational) environment. CQoS expands on traditional QoS ideas by considering application-specific metrics, or metadata, which enable the annotation and characterization of component performance. Before automating the selection of component instances, however, one must be able to collect and analyze performance information and related metadata. The two main facets of CQoS tools, therefore, are measurement and analysis infrastructure and control infrastructure for dynamic component replacement and domain-specific decision making. This paper focuses on the performance and metadata management and analysis support in CQoS infrastructure.

We present in this paper recent work by members of the CCA Forum and the Quantum Chemistry Science Application Partnership (QCSAP) [5], which includes developers of several leading high-performance quantum chemistry codes (GAMESS [6], MPQC [7], and NWChem [8]), to utilize the new CQoS infrastructure to guide adaptive runtime composition and performance optimization of component-based parallel quantum chemistry applications. Parallel application configuration has driven the initial development and integration of CQoS infrastructure in the QCSAP, laying the groundwork for more sophisticated analysis to configure algorithmic parameters for particular molecular targets, calculation approaches, and hardware environments.

The remainder of this paper is organized as follows. Section 2 provides an overview of related work; Section 3 highlights key features of the Common Component Architecture and its use in high-performance quantum chemistry. Section 4 introduces our CQoS approach for quantum chemistry, and Section 5 reports on some preliminary experiments. Section 6 provides concluding remarks and discussion of future work.

2 Related Work

Adaptive software architecture is an area of emerging research, as evidenced by numerous recent projects and related work [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Many approaches to addressing different aspects of adaptive execution are represented in these projects, from compiler-based techniques to performance model-based engineering approaches and development of new adaptive numerical algorithms.

Unlike these efforts, our approach specifically targets large-scale parallel computations and support of interoperability among scientific packages. In designing our CQoS database interfaces and middleware components, we rely on the existing high-performance infrastructure provided by the CCA, in which multiple component implementations conforming to the same external interface standard are interoperable, and the runtime system ensures that the overhead of component substitution is negligible.

A large number of tools for performance analysis exist, including TAU [22], Prophecy [23], and SvPablo [24]. Each tool defines its own performance data representation and storage, from custom ASCII representations or XML to SQL, DB2, or Oracle databases. Efforts are under way to define common data representations for performance data; however, we expect that the standardization will take some time and it will be longer before tools implement mappings from their native formats to the standard one. Thus, we have focused on defining *interfaces* for querying and manipulating the data, which can then be implemented as components mapping to different representations. To our knowledge, the research discussed in this paper is the first attempt to provide language-independent component interfaces and corresponding implementations for performance database manipulation, specifically targeting parallel scientific applications. This approach supports multiple underlying representations and does not preclude the use of non-component performance analysis tools.

A rich set of performance tools [25, 26, 27], including PerfExplorer [28], aim to improve the execution behavior of a program based on information on its current or previous runtime behavior. The tools, however, use low-level compiler-based techniques or are restricted to specific parallel computer systems or application domains. In contrast, we integrate PerfExplorer into the CCA infrastructure to support adaptation in generic parallel scientific applications.

3 Common Component Architecture and Quantum Chemistry

CCA Overview. This work leverages the component standard for scientific computing under development by the CCA Forum. Component technology (see, e.g., [29]), which is now widely used in mainstream computing but has only recently begun to make inroads in high-performance computing (HPC), extends the benefits of object-oriented design by providing coding methodologies and supporting infrastructure to improve software's extensibility, maintainability, and reliability.

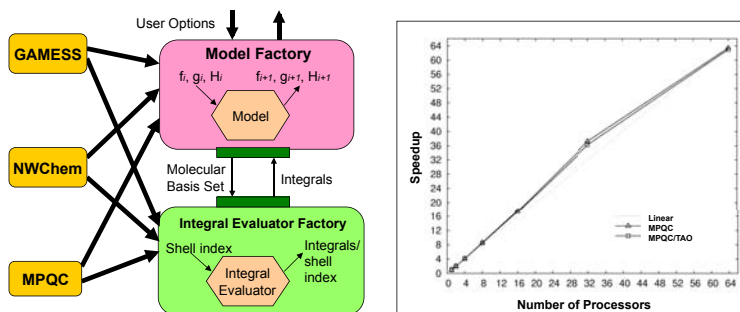


Fig. 1. *Left:* CCA component approach for a quantum chemistry application. *Right:* Isoprene HF/6-311G(2df,2pd) parallel speedup in MPQC-based CCA simulations of molecular geometry; the plot shows nearly linear speedup on 1 through 64 processors both when using MPQC alone and when using external numerical optimization components in TAO.

The CCA Forum is addressing productivity challenges of diverse scientific research teams by developing tools for plug-and-play composition of applications in parallel and distributed computing. The core of this work is a component model and reference implementation [2] tailored to the needs of high-end scientific computing. Key features are language-neutral specification of common component interfaces, interoperability for software written in programming languages important to scientific computing, and dynamic composability, all with minimal runtime overhead.

The specification of the Common Component Architecture defines the rights, responsibilities, and relationships among the various elements of the model. Briefly, the elements of the CCA model are as follows:

- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces.
- *Ports* are the interfaces through which components interact. Components may provide ports, meaning that they implement the functionality expressed in a port (called *provides ports*), or they may use ports, meaning that they make calls on a port provided by another component (called *uses ports*).
- *Frameworks* manage CCA components as they are assembled into applications and executed. The framework is responsible for connecting *uses* and *provides* ports.

Quantum Chemistry Components. The QCSAP has adopted a component architecture based on the CCA. Both coarse-grain componentization (where a component encompasses a task such as energy, gradient, or Hessian evaluation) and fine-grain componentization (where a component computes only integrals) are incorporated [30,31]. The left-hand side of Figure 1 illustrates how common

component interfaces for molecules, models, basis sets, and integral evaluation facilitate the sharing of code among these three chemistry teams.

The component approach also enables the chemistry teams to leverage external capabilities in the wider scientific community. For example, as shown in the right-hand side of Figure 1, chemists have achieved scalable performance in MPQC-based CCA simulations for molecular shape determination using parallel components from the Toolkit for Advanced Optimization (TAO) [30].

4 CQoS Infrastructure and Its Application in High-Performance Quantum Chemistry

Scientific computations often require the specification of many options and parameters. Quantum chemical options, for instance, include the basic selection of methods, expansion basis, and convergence criteria, as well as low level details such as hardware configuration and algorithmic parameters. The initial focus of CQoS tools is parallel application configuration to effectively exploit high-performance architectures. The difficulty of configuring parallel computations is compounded by the proliferation of multicore processors, resulting in three levels of processing elements (nodes, processors/sockets, and processor cores), and the variety of hardware environments, ranging from networks of workstations for development to massively parallel machines for production runs.

A key aspect of CQoS research is generating and collecting meaningful data for storage in a database and subsequent analysis by performance tools. Full automation of the processes of collecting and managing performance data and metadata, building a performance model, and conducting detailed evaluation is beyond the scope of this paper. We focus on coarse-grain computations, where we select several major parameters that can affect the performance of scientific codes and then use the performance data to enable adaptive application configuration.

While motivated by adaptivity in different problems (quantum chemistry, combustion, fusion, and accelerator simulations), we believe that the *infrastructure* for analyzing and characterizing the problems and determining and invoking solution strategies will indeed be similar for such large-scale scientific simulations. We introduce nonfunctional properties and application-specific information, or metadata, into performance analysis and decision-making. Metadata include algorithm or application parameters, such as problem size and physical constants, compiler optimization options, and execution information, such as hardware and operating system information. Ideally, for each application execution, the metadata should provide enough information to be able to repeat the run. The metadata can help classify performance measurements and provide clues for tuning performance. CQoS infrastructure, therefore, includes database components that manage performance data and associated metadata, comparator components that support query and extraction of data, and analysis components that conduct performance analysis to suggest adaptation strategies. Figure 2 illustrates the utilization of the CQoS components (described in more detail in Sections 4.2 and 4.3) in an adaptive quantum chemistry application. Thanks to our uniform quantum chemistry interfaces, the capabilities for

Generating and collecting meaningful data for subsequent analysis by performance tools is not as straightforward as one might expect, however, and the task is especially challenging for high-performance scientific applications like quantum chemistry. Many parameters in quantum chemistry computations can affect the efficiency and accuracy of a computation; moreover, it is not always obvious how to quantify some of these parameters. The complexity of modern HPC architectures only exacerbates the difficulty of finding appropriate parameters to achieve the best results (or tradeoff between accuracy and efficiency).

In order to acquire more detailed performance data for CQoS research, some supporting tools have been developed that facilitate data collection and management [32]. In addition, CCA-compliant TAU-based performance monitoring components [33] can be employed to collect performance data for computational components.

4.2 Database Components and Their Usage in Data Training

The database component interface design is intended to support the management and analysis of performance and application metadata, so that the mapping of a problem to a solution that can potentially yield the best performance can be accomplished statically or at runtime. The UML diagram in Figure 3 shows the main interfaces and some of their methods.

We introduce two types of components for storing and querying CQoS performance data and metadata. The database component provides general-purpose interfaces for storing and accessing data in a physical database. The comparator interfaces compare and/or match properties of two problems under user-specified conditions.

Comparator Components. Three sets of interfaces are associated with a comparator component: *Parameter*, *ParameterSet*, and *Comparator*. A *Parameter* captures a single property of a problem, for example, the count of a specific atom type in a molecule. A *Parameter*, which is described by its name, data type, and value, is associated with a table in the physical database. The *Parameter* interfaces also support comparisons against another peer parameter under user-specified conditions. A *ParameterSet* represents a group of related parameters, for example, a set of parameters that characterizes a molecule or a set of scalar or Boolean linear system properties. Using the *ParameterSet* interfaces, users can create and manage parameter set members. When selecting

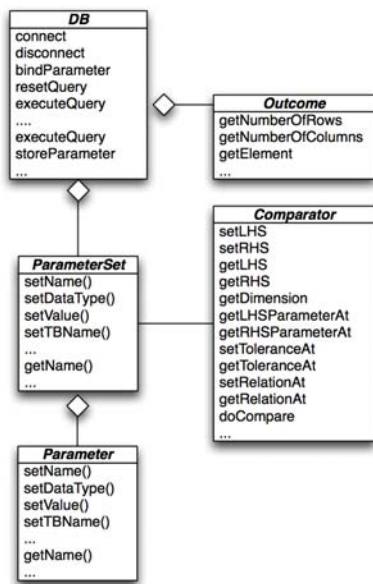


Fig. 3. UML diagram of CQoS database component interfaces and methods

a solution method, time-dependent problem or system properties are described as one or more *ParameterSets*. The user or an automated adaptive heuristic can then match the formatted parameter sets to a database to determine the best solution method or configuration. A *Comparator* defines the rules to compare two sets of parameters. For instance, a *Comparator* can determine the closeness of two sets of parameters (i.e., whether they are within ϵ of each other).

Database Components. There are two classes of interfaces associated with a database component, *DB* and *Outcome*. The application connects to a database component by using the *DB* port, which handles (potentially remote) database connections, queries, and storage and retrieval of parameters and parameter sets. The *DB* interface also supports the query of experimental runs having parameter sets that satisfy user-specified conditions (e.g., limiting the parameter set to a range of values). The *Outcome* interface supports transformation of database results returned from a DB query to user-readable format, as well as access to the individual data elements.

During the training phase of the CQoS process for quantum chemistry, performance statistics and application metadata for selected problem instances are added into the database. This training data can then be used for future performance analysis and solution method matches, as further discussed in Section 4.3. Before the execution of an application, application-specific *Comparator* implementations help match the initial problem properties and system states against historical information to find a good initial solution method. During runtime, time-dependent application and system characteristics are captured in metadata parameter sets. At runtime the *Comparator* implementation can dynamically match the metadata against a lightweight runtime database to determine the best-known method corresponding to the current application state.

4.3 Performance Analysis

We incorporated PerfExplorer [28] into CQoS infrastructure to support performance analysis and decision-making for runtime adaptivity. PerfExplorer, a framework for parallel performance data mining and knowledge discovery in the TAU performance system, was developed to facilitate analysis on large collections of experimental performance data. The framework architecture enables the development and integration of data-mining operations that can be applied to parallel performance profiles. The data repository for PerfExplorer is PerfDMF [34], a performance data management framework that integrates and interfaces to many common database management systems, including the CQoS training database. PerfExplorer is designed to make the process of analyzing large numbers of parallel performance profiles manageable. Dimension reduction methods such as clustering and correlation allow meaningful analysis of large data sets. PerfExplorer does not directly implement these techniques; rather, it is integrated with existing analysis toolkits (e.g., Weka [35]) and provides for extensions using those toolkits. One such extension is to use classification capabilities in the Weka data-mining package to construct a runtime parameter

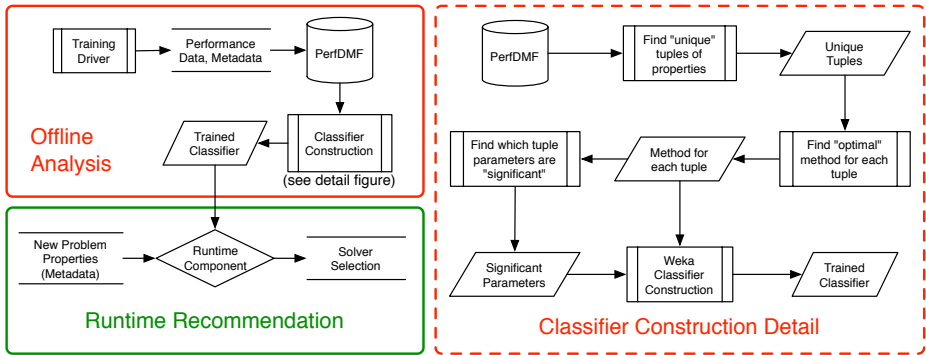


Fig. 4. PerfExplorer classifier construction

recommendation system. Classification systems are a type of machine learning in which training data is input into a decision tree or space partitioning algorithm to construct a classifier. Classifiers belong a particular class of machine learning known as supervised learning, in which vetted training data with pre-selected attributes and known class types are used to train the classifier. In contrast, for exploratory, unsupervised learning methods such as clustering, class identifications are not known ahead of time. With a trained classifier, new test data can be identified as belonging to one of the identified classes. Classifiers can also be used to perform numerical prediction.

Figure 4 shows how a classifier for runtime recommendation is constructed and used. Training data for analysis is generated by executing the application multiple times, varying key parameters that have an effect on the total runtime. After the performance data and associated metadata for a set of training runs have been stored in the performance database, PerfExplorer loads the data and classifies it. A classifier is constructed using a simple Python script interface in which the application developer specifies independent application parameters and the dependent parameter, or class. Within the Python script, the classification method is also selected. Supported methods include alternating decision trees, support vector machines, and multilayer perceptrons (neural networks). Unique tuples of each combination of parameter values are found, and the best performing execution for each unique tuple is selected to represent that class. Optionally, Principle Components Analysis can be used to reduce the parameters to those that have the most influence over the variance in the data set. All classification is performed offline, as it can be a time intensive process. The results of the classification are stored in the form of a serialized Java object. This process can be performed either through PerfExplorer's GUI, the command line interface, or by using a CCA component wrapping PerfExplorer.

For production application runs, the classifier is loaded into a CCA component. The best parameter setting (class) is obtained by querying the classifier with the current values of the application-specific metadata. These values are matched to the classification properties to find the best class selection for the

parameters. The runtime overhead of this step is minimal because it does not require access to the performance data database.

4.4 Adaptive Application Composition and Configuration

In application domains in which multiple software packages implement standard interfaces, we are able to capitalize on the CCA component approach and provide a single domain-specific CQoS component that manages interaction between generic CQoS infrastructure and various domain-specific implementations (in this case, GAMESS, MPQC, and NWChem), thereby reducing and simplifying the CQoS management code required in the domain-specific packages themselves. In Figure 5 a snapshot from the GUI of the Ccaffeine framework [36] illustrates the composition of a QCSAP quantum chemistry package (MPQC) and generic CQoS infrastructure components through a chemistry-specific CQoS component. The left-hand side of the figure shows the *palette* that contains available template components. In the wiring diagram on the right-hand side, a chemistry application instantiates both chemistry and CQoS components and connects so-called *uses* ports, as introduced in Section 3, and *provides* ports between related components.

The training driver component, *Driver*, manages chemistry model objects (e.g., *MPQCFactory* in Figure 5), acquires metadata from the model objects, and serializes interactions with the chemistry CQoS component, *CQoS*. During the training phase (see Section 4.2), the *Driver* populates a CCA type map, or dictionary, with metadata obtained from *MPQCFactory* describing algorithmic parameterization, hardware configuration, and application performance. For

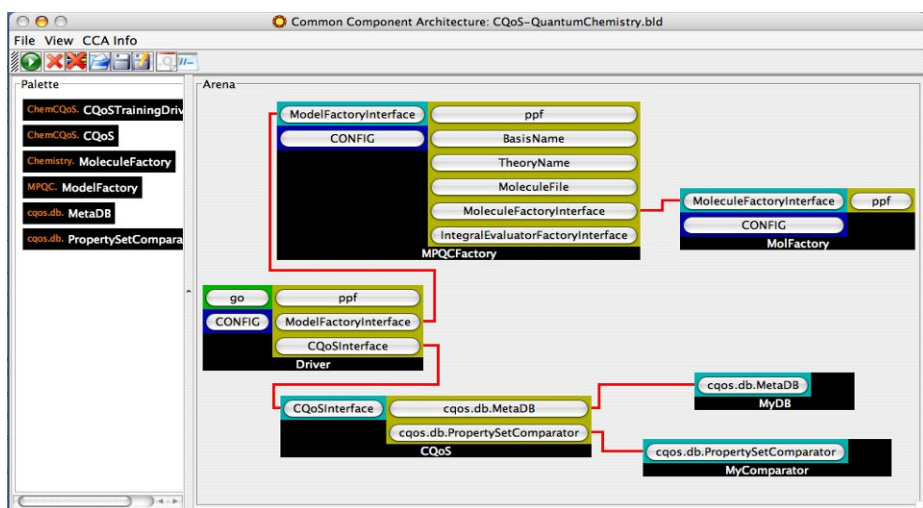


Fig. 5. Component wiring diagram for quantum chemistry showing usage of CQoS database components

each training run, this metadata container is passed, along with a molecule descriptor class, to the chemistry CQoS interfaces, *CQoS*. The *CQoS* derives the metadata and handles the actual calls to the database component, *MyDB*, to store the metadata in the CQoS database.

For production runs, the *CQoS* passes metadata that describes a new calculation to the comparator component, *MyComparator*, to map the data to an application configuration that can potentially yield the best performance. Additionally, the CQoS infrastructure components can leverage the classification capabilities provided within the TAU package to obtain recommended configurations for calculations. The proposed production calculations are related to training calculations based on similarities in molecular properties and the resulting work load. Performance metrics associated with the training runs allow the TAU classifiers to recommend parameters, such as hardware configuration or algorithmic parameters, which aim to achieve properties such as good algorithmic performance (e.g., minimum iterations to solution), minimal time to solution, or maximum parallel efficiency. Once the TAU classifier has been constructed using training performance data and metadata, we do not need to build it again as the execution proceeds. Therefore the cost of training the classifier can be amortized by using it in many application runs.

The CQoS infrastructure can be used either for dynamic reconfiguration of an application during execution to respond to changing computing environments or for selection of initial parameters, with the goal of determining how to maximize performance before the application runs. For runtime adaptation, the driver code checkpoints, collects the metadata describing the current computing state, and passes it to the TAU classifier. After the classifier suggests an optimal configuration, the driver can resume the execution by replacing with the recommended parameters. The period for checkpointing can be variable depending on the computing resource demand and program semantics. In an application where good initial parameter settings are essential for overall performance, we can use the CQoS infrastructure to determine appropriate initial parameter values based on the training or historical data. We expect that the predicted values will perform better than random values or at least as well as the values adopted by other researchers when performing experiments. We have partially evaluated this approach in experiments presented in the next section.

5 Experimental Results

As discussed in Section 1, computational chemistry applications rely on numerous software packages offering a range of theoretical methods and implementations. Our CQoS approach aims to automatically adapt these packages under these challenging situations, simplifying the tasks for end users.

5.1 MPQC

As an initial demonstration of the previously described CQoS architecture, the MPQC model was connected to the chemistry CQoS infrastructure and used to

generate a small training data set. Hartree Fock energies were computed for five molecular structures obtained from the G2 neutral test set, which is commonly used for benchmarking. The five molecules selected were sulfur dioxide, disilane, vinyl chloride, acetic acid, and pyridine. For each molecule, the Hartree Fock energy was calculated using two selected basis sets, cc-pVTZ and aug-cc-pVQZ, with node counts ranging from 1 to 32 (as powers of 2) on the Catalyst cluster at Sandia Livermore (2-way Pentium 4 nodes with an Infiniband interconnection network). For these smaller calculations that cannot support high parallel efficiency at large scales, these node counts span from 100% parallel efficiency at low node counts to severe degradation of efficiency at larger node counts.

To demonstrate the potential for efficiency savings in this software environment, a Hartree Fock energy for the nitromethane molecule using the cc-pVTZ basis was chosen as a sample target calculation. We employed a very simple node count selection algorithm: selecting the training calculation with the nearest basis function count as the target calculation and then choosing the highest node count with parallel efficiency greater than 90% for that training calculation. Using this simplistic algorithm, the nitromethane target calculation achieves a parallel efficiency of 84%, which is 8% greater parallel efficiency than the next larger power of 2 node count. While these small sample calculations will not support efficient execution at large scales of parallelism, increasing the problem size only pushes these effects to larger node counts; the shapes of the efficiency curves and the potential efficiency gains using CQoS approaches will remain. With current environmental and energy efficiency concerns and yearly power budgets for modern HPC installations running into the millions of dollars, it seems clear that CQoS approaches should be a part of HPC efforts.

5.2 GAMESS

GAMESS, another application that computes Hartree Fock energies, has many options for solving the properties of the wavefunctions. There are two implementations for the energy solution, *conventional* and *direct*. The conventional implementation was written first but can be too resource intensive in terms of disk space and file I/O requirements on some systems. The direct version was developed to avoid storing intermediate integrals on disk, thus requiring some redundant computations, and in serial executions, is typically two to three times slower than the conventional. However, in parallel environments at higher processor counts, the direct method outperforms the conventional method due to excessive parallel and I/O overhead. The actual point where it makes sense to change methods depends on the wavefunction solution method, the input molecule or atom, the basis set, and the hardware. In addition, at one stage of the algorithm, a second-order Møller-Plesset correction (MP2 [37]) can be used to take into account the so-called “electron correlation.” The MP2 method consumes more memory and disk space. With regard to the GAMESS application, one goal of the recommender system is to suggest whether to use the direct or conventional method, given the other parameter selections.

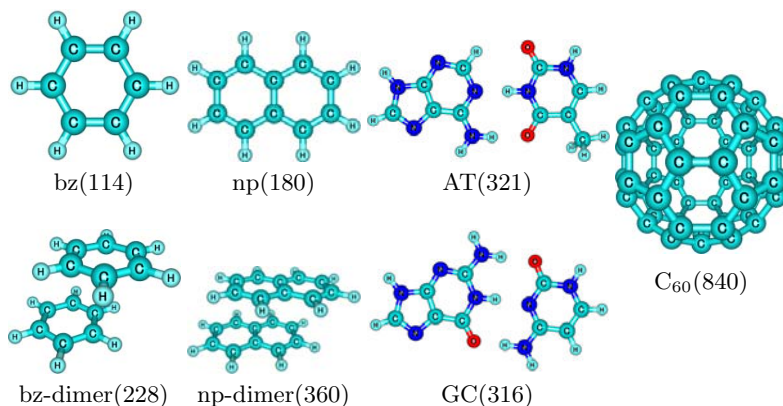


Fig. 6. Test cases: Benzene (bz) and its dimer, Naphthalene (np) and its dimer, Adenine-Thymine DNA base pair (AT), Guanine-Cytosine DNA base pair (GC), Buckminsterfullerene (C_{60}). In parentheses are the numbers of basis functions when using cc-pVDZ basis.

The initial set of molecules used for performance analysis is shown in Figure 6. Indicated numbers of basis functions (in parentheses) roughly correlate with resource demand of the corresponding computations: the greater the number of basis functions, the more demanding the computation is expected to be. The choice of molecules was based on their importance in chemistry and biology as well as on characteristic types of chemical interactions they represent; also, computations of molecules of a similar size (that is, with similar number of atoms and basis functions) are routine in contemporary quantum chemistry. The benzene and naphthalene molecules (labeled “bz” and “np” on the figure) represent fundamental aromatic systems. Their dimers (labeled “bz-dimer” and “np-dimer”) represent models for π - π interactions believed to determine DNA stacking, protein folding, and other phenomena of great importance for biochemistry and chemistry. The pairs of DNA bases (labeled “AT” and “GC”) are examples of hydrogen bonding; the interaction between the bases defines the double-helix structure of DNA. Finally, a molecule of buckminsterfullerene (labeled C_{60}) is taken as a representative of a large, highly symmetrical chemical structure, characteristic of carbon nanomaterials.

Hartree Fock energy was computed for each of these molecules, with and without MP2 correction, with various basis sets, and with varying numbers of nodes (up to 16) and processes per node (up to 8). The runs were computed on Bassi, an IBM p575 POWER5 system at the National Energy Research Scientific Computing Center (NERSC). Bassi has 111 compute nodes with 8 processors and 32 GB of memory per node. This training data was used to construct a classifier in order to recommend whether to use the conventional or direct method to compute the energy. The independent parameters used to construct the classifier are shown in Table 1. There were 561 potential training instances, of which 150 of the best performing unique tuples were selected as the training set. The method

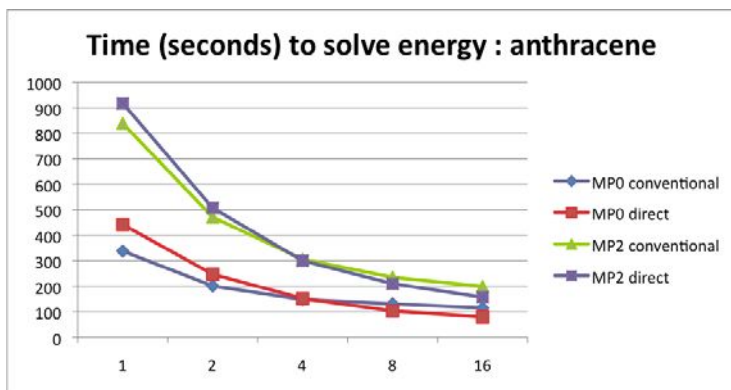


Fig. 7. Anthracene empirical results: Number of nodes vs. wall-clock time. Times are in seconds; less time indicates faster performance.

Table 1. Parameters used for classifier construction

Property	Training Values	Anthracene
# Cartesian Atomic Orbitals	120, 190, 240, 335, 340, 380, 470, 830	640
# Occupied Orbitals	21, 34, 42, 68	47
# Processes per Node	2, 4, 8	8
# Nodes	1, 2, 4, 8, 16	1,2,4,8,16
Second-order Møller-Plesset	disabled (MP0), enabled (MP2)	MP0, MP2

(conventional or direct) used to generate the best performing tuple was used as the class for each training instance. A multilayer perceptron classifier was constructed using these instances.

An eighth molecule, anthracene, was used to test the classifier. The test values for the parameters are also shown in Table 1. When used at runtime, the classifier recommended using the conventional method for the 1, 2, and 4 node runs (8, 16, and 32 processes, respectively), and using the direct method for the 8 and 16 node runs (64 and 128 processes). The empirical results from anthracene are shown in Figure 7. The classifier was correct in classifying 9 out of 10 instances – the 4 node direct MP2 run outperformed the conventional MP2 run, but only barely (300 seconds compared to 306 seconds). In all of the other configurations, the recommender correctly identified the method to use in order to achieve the fastest time to completion.

6 Conclusion and Future Work

This paper introduced a CQoS approach for quantum chemistry that leverages the CCA component environment to address new challenges being faced by applications teams when dynamically composing and configuring codes in high-performance computing environments. We have built prototype database

components for managing performance data and associated metadata for high-performance component applications. These components are part of a larger CQoS infrastructure, which has the goal of enabling automated component selection and configuration of component-based scientific codes to respond to continuous changes in component requirements and their execution environments. We integrated performance analysis capabilities of PerfExplorer into the general CQoS infrastructure to classify performance and meta-information and then suggested appropriate configurations for new problem instances. The usage of the CQoS infrastructure components in quantum chemistry applications demonstrates our initial success in adaptive parallel application configuration.

Our next steps in the quantum chemistry-specific part of this research include predicting, in addition to the runtime, the accuracy of the computations (with respect to energy or other properties). We will employ other classes of widely-used quantum chemical methods, starting with density functional theory and coupled clusters approaches. We will also incorporate new metadata fields whose values will be collected along with performance data, for example, parameters representing molecular symmetry. Furthermore, we plan to venture into the (rather unexplored) area of quantification of similarity between molecules. More generally, future work includes enhancing the CQoS infrastructure to support sophisticated analysis for (re)configuring algorithmic parameters and component instances during runtime, developing application-specific performance models, and incorporating the training CQoS phase into empirical experiment design.

We are also employing the new CQoS infrastructure to facilitate dynamic adaptivity of long-running simulations in other application domains, including parallel mesh partitioning in combustion and efficient solution of large-scale linear systems in fusion and accelerator models [3]. Our long-term goals are to define a comprehensive architecture for enabling CQoS in scientific simulations, consisting of general-purpose performance monitoring, analysis, and database middleware components, which can then be combined with easy-to-write domain-specific components for defining quality metrics and adaptation strategies.

Acknowledgments

We thank the reviewers of this paper for valuable feedback that helped us to clarify the presentation.

The CCA has been under development since 1998 by the CCA Forum and represents the contributions of many people, all of whom we gratefully acknowledge.

This work was supported in part by the Office of Advanced Scientific Computing Research via the Scientific Discovery through Advanced Computing (SciDAC) initiative [38], Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357, DE-AC04-94AL85000, DE-AC02-07CH11358, and DE-FG02-07ER25826. This work is a collaboration among the Quantum Chemistry Scientific Application Partnership (QCSAP) [5], the Center for Component Technology for Terascale Simulation Software [39], and the Performance Engineering Research Institute [40].

References

1. Armstrong, R. et al.: Common Component Architecture (CCA) Forum, <http://www.cca-forum.org/>
2. Allan, B.A., Armstrong, R., Bernholdt, D.E., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl.* 20(2), 163–202 (2006)
3. McInnes, L.C., Ray, J., Armstrong, R., Dahlgren, T.L., Malony, A., Norris, B., Shende, S., Kenny, J.P., Steensland, J.: Computational quality of service for scientific CCA applications: Composition, substitution, and reconfiguration. Technical Report ANL/MCS-P1326-0206, Argonne National Laboratory (February 2006)
4. Norris, B., Ray, J., Armstrong, R., McInnes, L.C., Bernholdt, D.E., Elwasif, W.R., Malony, A.D., Shende, S.: Computational quality of service for scientific components. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) *CBSE 2004*. LNCS, vol. 3054, pp. 264–271. Springer, Heidelberg (2004)
5. Gordon, M. (PI): Chemistry Framework using the CCA, <http://www.scidac.gov/matchem/better.html>
6. Schmidt, M.W., Baldrige, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S.J., Windus, T.L., Dupuis, M., Montgomery, J.A.: General atomic and molecular electronic structure system. *J. Computational Chemistry* 14, 1347–1363 (1993)
7. Janssen, C.L., Nielsen, I.M.B., Colvin, M.E.: Encyclopedia of computational chemistry. In: Schleyer, P.V.R., Allinger, N.L., Clark, T., Gasteiger, J., Kollman, P.A., Schaefer III, H.F., Scheiner, P.R. (eds.) *Encyclopedia of Computational Chemistry*. John Wiley & Sons, Chichester (1998)
8. Kendall, R.A., Apra, E., Bernholdt, D.E., Bylaska, E.J., Dupuis, M., Fann, G.I., Harrison, R.J., Ju, J.L., Nichols, J.A., Nieplocha, J., Straatsma, T.P., Windus, T.L., Wong, A.T.: High performance computational chemistry: An overview of NWChem, a distributed parallel application. *Comput. Phys. Commun.* 128, 260–270 (2000)
9. Dongarra, J., Eijkhout, V.: Self-adapting numerical software for next generation applications. *Int. J. High Performance Computing Applications* 17, 125–131 (2003); also LAPACK Working Note 157, ICL-UT-02-07
10. Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35(2), 101–121 (2005)
11. Liu, H., Parashar, M.: Enabling self-management of component based high-performance scientific applications. In: *Proc. 14th IEEE Int. Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos (2005)
12. Tapus, C., Chung, I.H., Hollingsworth, J.K.: Active Harmony: Towards automated performance tuning. In: *Proc. of SC 2002* (2002)
13. Vetter, J.S., Worley, P.H.: Asserting performance expectations. In: *Proc. SC 2002* (2002)
14. Bramley, R., Gannon, D., Stuckey, T., Villacis, J., Balasubramanian, J., Akman, E., Berg, F., Diwan, S., Govindaraju, M.: The Linear System Analyzer. In: *Enabling Technologies for Computational Science*. Kluwer, Dordrecht (2000)

15. Zhang, K., Damevski, K., Venkatachalapathy, V., Parker, S.: SCIRun2: A CCA framework for high performance computing. In: Proc. 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), Santa Fe, NM. IEEE Press, Los Alamitos (2004)
16. Houstis, E.N., Catlin, A.C., Rice, J.R., Verykios, V.S., Ramakrishnan, N., Houstis, C.E.: A knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software* 26(2), 227–253 (2000)
17. McCracken, M.O., Snaveley, A., Malony, A.: Performance modeling for dynamic algorithm selection. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003. LNCS, vol. 2660, pp. 749–758. Springer, Heidelberg (2003)
18. Vuduc, R., Demmel, J., Bilmes, J.: Statistical models for empirical search-based performance tuning. *Int. J. High Performance Computing Applications* 18(1), 65–94 (2004)
19. Cortellessa, V., Crnkovic, I., Marinelli, F., Potena, P.: Experimenting the automated selection of COTS components based on cost and system requirements. *J. Universal Computer Science* 14(8), 1228–1256 (2008)
20. Becker, S., Kozirolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. *J. Systems and Software* 82(1), 3–22 (2009)
21. Kappler, T., Kozirolek, H., Krogmann, K., Reussner, R.: Towards automatic construction of reusable prediction models for component-based performance engineering. In: Proc. Software Engineering (February 2008)
22. Shende, S., Malony, A.: The TAU parallel performance system. *Int. J. High-Perf. Computing Appl.*, ACTS Collection special issue 20, 287–331 (Summer 2006)
23. Taylor, V., Wu, X., Stevens, R.: Prophecy: An infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev.* 30(4), 13–18 (2003)
24. de Rose, L.A., Reed, D.A.: SvPablo: A multi-language architecture-independent performance analysis system. In: ICPP 1999: Proc. 1999 International Conference on Parallel Processing. IEEE Computer Society, Los Alamitos (1999)
25. Jorba, J., Margalef, T., Luque, E.: Performance analysis of parallel applications with KappaPI2. In: Proc. Parallel Computing: Current and Future Issues of High-End Computing, pp. 155–162 (2006)
26. Voss, M.J., Eigemann, R.: High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices* 36, 93–102 (2001)
27. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An algebra for cross-experiment performance analysis. In: Proc. 2004 International Conference on Parallel Processing (ICPP 2004), Montreal, Quebec, Canada, pp. 63–72 (2004)
28. Huck, K.A., Malony, A.D., Shende, S., Morris, A.: Scalable, automated performance analysis with TAU and PerfExplorer. In: *Parallel Computing* (2007)
29. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York (1999)
30. Kenny, J.P., Benson, S.J., Alexeev, Y., Sarich, J., Janssen, C.L., McInnes, L.C., Krishnan, M., Nieplocha, J., Jurrus, E., Fahlstrom, C., Windus, T.L.: Component-based integration of chemistry and optimization software. *J. Computational Chemistry* 25(14), 1717–1725 (2004)
31. Peng, F., Wu, M.S., Sosonkina, M., Bentz, J., Windus, T.L., Gordon, M.S., Kenny, J.P., Janssen, C.L.: Tackling component interoperability in quantum chemistry software. In: *Workshop on Component and Framework Technology in High-Performance and Scientific Computing, in conjunction with OOPSLA* (2007)

32. Wu, M.S., Bentz, J., Peng, F., Sosonkina, M., Gordon, M.S., Kendall, R.A.: Integrating performance tools with large-scale scientific software. In: The 8th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2007), in conjunction with 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, California (March 2007)
33. Malony, A., Shende, S., Trebon, N., Ray, J., Armstrong, R., Rasmussen, C., Sotile, M.: Performance technology for parallel and distributed component software. *Concurrency and Computation: Practice and Experience* 17, 117–141 (2005)
34. Huck, K.A., Malony, A.D., Bell, R., Morris, A.: Design and implementation of a parallel performance data management framework. In: Proc. 2005 International Conference on Parallel Processing, ICPP 2005, pp. 473–482. IEEE Computer Society Press, Los Alamitos (2005)
35. Witten, I.H., Frank, E.: *Data mining: Practical machine learning tools and techniques* (2005), <http://www.cs.waikato.ac.nz/~ml/weka/>
36. Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., Wolfe, P.: Ccaffeine – a CCA component framework for parallel computing (2003), <http://www.cca-forum.org/ccafe/>
37. Møller, C., Plesset, M.S.: Note on an approximation treatment for many-electron systems. *Phys. Rev.* 46, 618–622 (1934)
38. U. S. Dept. of Energy: SciDAC Initiative homepage (2006), <http://www.osti.gov/scidac/>
39. Bernholdt D. (PI): TASCSCenter, <http://www.scidac.gov/compsci/TASCSC.html>
40. Lucas, R. (PI): Performance Engineering Research Institute (PERI), <http://www.peri-scidac.org>

Author Index

- Amaldi, Edoardo 163
Anselmi, Jonatha 163
- Biehl, Matthias 36
Björnander, Stefan 101
Buckl, Sabine 52
- Chan, Kenneth 86
Clerc, Viktor 130
Cremonesi, Paolo 163
Curfman McInnes, Lois 194
- Doppelhamer, Jens 177
Duchien, Laurence 1
- Edwards, George 146
- Gaenko, Alexander 194
Garcia, Joshua 146
Ghezzi, Carlo 70
Gokhale, Aniruddha 18
Gordon, Mark S. 194
Grunske, Lars 101
- Hinsman, Carl 116
Huck, Kevin 194
- Janssen, Curtis L. 194
- Kavimandan, Amogh 18
Kenny, Joseph P. 194
Koziolk, Heiko 177
- Krogmann, Klaus 52
Kuperberg, Michael 52
- Le Meur, Anne-Françoise 1
Li, Li 194
Löwe, Welf 36
Lundqvist, Kristina 101
- Martens, Anne 52
Matthes, Florian 52
Medvidovic, Nenad 146
Mori, Hirotoshi 194
- Netzloff, Heather M. 194
Norris, Boyana 194
- Perdeck, Michiel 130
Poernomo, Iman 86
Poort, Eltjo R. 130
Popescu, Daniel 146
Pramono, Agung 130
- Sangal, Neeraj 116
Schweda, Christian M. 52
Stafford, Judith 116
- Tamburrelli, Giordano 70
- van Vliet, Hans 130
- Wagnier, Guillaume 1
Weiss, Roland 177
Windus, Theresa L. 194
Wu, Meng-Shiou 194