Component-based software engineering (CBSE) is concerned with the development of software-intensive systems from reusable parts (components), the development of such reusable parts, and the maintenance and improvement of systems by means of component replacement and customization. Although it holds considerable promise, there are still many challenges facing both researchers and practitioners in establishing CBSE as an efficient and proven engineering discipline.

Six CBSE workshops have been held consecutively at the most recent six International Conferences on Software Engineering (ICSE). The premise of the last three CBSE workshops was that the long-term success of component-based development depends on the viability of an established science and technology foundation for achieving predictable quality in component-based systems.

The intent of the CBSE 2004 symposium was to build on this premise, and to provide a forum for more in-depth and substantive treatment of topics pertaining to predictability, to help establish cross-discipline insights, and to improve cooperation and mutual understanding. The goal of the CBSE 2004 symposium was to discuss and present more complete and mature works, and consequently collect the technical papers in published proceedings. The response to the Call for Papers was beyond expectations: 82 papers were submitted. Of those 25 (12 long and 13 short) were accepted for publication. In all 25 cases, the papers were reviewed by three to four independent reviewers. The symposium brought together researchers and practitioners from a variety of disciplines related to CBSE.

CBSE 2004 was privileged to have very competent, engaged and cooperative organizing and program committees with members involved in the forming of the symposium, its organization and in the review process. The review process, including the virtual review meetings, was organized completely electronically and succeeded thanks to the devoted work of the members and additional reviewers, and the excellent support from Richard van de Stadt who provided the electronic review system. The organizers of the ICSE 2004 conference, in particular Anthony Finkelstein, the General Chair, and Neno Medvidovic, the Workshops Chair, with great help and flexibility made it possible to organize CBSE 2004 as an adjunct event to the ICSE 2004 workshops. Springer-Verlag kindly agreed to publish the proceedings volume and helped greatly in its realisation. Finally all the contributors, the authors of the accepted papers, invited speakers and panelists contributed to the success of the symposium. We would like to thank each of them for their excellent contributions.

March 2004

Ivica Crnkovic
Heinz Schmidt
Judith Stafford
Kurt Wallanu
Many hold that component software is the way to the next level of the software field’s productivity. Others object that progress has been slow and that fundamental roadblocks continue to be in the way. Ultimately, it is the need to move from manufacturing to an industrial approach that encourages the move away from monolithic software towards component-based engineering. Yet, it is true that much remains to be done and that component technologies available today have significant shortcomings. The same holds at the level of methodologies, processes, design and implementation languages, and tools.

The successful call for contributions to CBSE 2004 was a strong sign of the growing international attention. Research in academia and industry alike is embracing component software. With a maturing understanding of how components relate to other approaches, such as services and generators, the field is moving into a phase that promises good progress on both fundamental and practical issues. The broad range of topics covered by the authors of the accepted papers is a clear indication. From fundamental concerns of correctness and extrafunctional properties of composition to the architectural embedding of components, to methods and processes, and to the implications of using commercial off-the-shelf components – this symposium covers all of these topics.

With a strong and healthy community forming and growing, it was about time for CBSE to move from being a well-attended workshop to being a fully peer-reviewed and published symposium in its own right. This year’s contributions inspire us to go that much further in the future. Hence, I am confident that we are seeing but the beginning of what I trust will develop into a successful series of events.

At this point, I would like to thank Ivica Crnkovic for running a smooth and efficient paper reviewing and selection process. Heinz Schmidt, Judy Stafford, and Kurt Wallnau supported the process greatly. I would also like to thank the two invited speakers, Hans Jonkers and Oscar Nierstrasz, who where quick to accept the invitation to speak at the newly shaped CBSE 2004 symposium, for delivering timely and thought-provoking contributions.

March 2004

Clemens Szyperski
CBSE 2004 was organized by Microsoft Research, USA, Monash University, Australia, Mälardalen University, Sweden, Carnegie Mellon University, USA and Tufts University, USA as an adjunct event to workshops at the 26th International Conference on Software Engineering (ICSE 2004).

Organizing Committee

General Chair, Clemens Szyperski (Microsoft Research, US)
Program Chair, Ivica Crnkovic (Mälardalen University, Sweden)
Program Co-chairs, Heinz Schmidt (Monash University, Australia)
Judith Stafford (Tufts University, USA)
Kurt Wallnau (SEI, Carnegie Mellon University, USA)

Program Committee

Uwe Aßmann, Linköping University, Sweden
Jakob Axelsson, Volvo Car Corporation, Sweden
Mike Barnett, Microsoft Research, USA
Judith Bishop, University of Pretoria, South Africa
Jan Bosch, University of Groningen, The Netherlands
Michel Chaudron, University of Eindhoven, The Netherlands
Wolfgang Emmerich, University College London, UK
Jacky Estublier, LSR-IMAG, France
Andre van der Hoek, University of California, Irvine, USA
Kathi Fisler, WPI, USA
Dimitra Giannakopoulou, NASA Ames, USA
Richard Hall, IMAG/LSR, France
Bengt Jonsson, Uppsala University, Sweden
Dick Hamlet, Portland State University, USA
George Heineman, WPI, USA
Paola Inverardi, University of L'Aquila, Italy
Shriram Krishnamurthi, Brown University, USA
Jeff Magee, Imperial College London, UK
Nenad Medvidovic, University of Southern California, USA
Magnus Larsson, ABB, Sweden
Rob van Ommering, Philips, The Netherlands
Heinz Schmidt, Monash University, Australia
Judith Stafford, Tufts University, USA
Dave Wile, Teknowledge, Corp., USA
Kurt Wallnau, SEI, Carnegie Mellon University, USA
## Co-reviewers

<table>
<thead>
<tr>
<th>Karine Arnout</th>
<th>Michel Jaring</th>
<th>Johan Muskens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Egor Bondarev</td>
<td>Eckhard Kruse</td>
<td>Martin Naedele</td>
</tr>
<tr>
<td>Ivor Bosloper</td>
<td>Christian Lange</td>
<td>Owen O’Malley</td>
</tr>
<tr>
<td>Reinder J. Bril</td>
<td>Henrik Lonn</td>
<td>John Penix</td>
</tr>
<tr>
<td>Marco Castaldi</td>
<td>Chris Lueer</td>
<td>Paul Pettersson</td>
</tr>
<tr>
<td>Humberto Cervantes</td>
<td>Sam Malek</td>
<td>Noël Plouzeau</td>
</tr>
<tr>
<td>Vittorio Cortellessa</td>
<td>Peter Mehlitz</td>
<td>Manos Renieris</td>
</tr>
<tr>
<td>Sybren Deelstra</td>
<td>Nikunj Mehta</td>
<td>Roshanak Roshandel</td>
</tr>
<tr>
<td>Fernando Erazo</td>
<td>Theo Dirk Meijler</td>
<td>Kevin B. Simons</td>
</tr>
<tr>
<td>Howard Foster</td>
<td>Marija Mikic-Rakic</td>
<td>Jorge Villalobos</td>
</tr>
<tr>
<td>Holger Hofmann</td>
<td>Raffaela Mirandola</td>
<td>Wang Yi</td>
</tr>
<tr>
<td>Anton Jansen</td>
<td>Mohammad Reza Mousavi</td>
<td></td>
</tr>
<tr>
<td>David N. Jansen</td>
<td>Henry Muccini</td>
<td></td>
</tr>
</tbody>
</table>

## Previous events

- CBSE 6 Workshop, Portland, USA (2003)
- CBSE 5 Workshop, Orlando, USA (2002)
- CBSE 4 Workshop, Toronto, Canada (2001)
- CBSE 2 Workshop, Los Angeles, USA (1999)
- CBSE 1 Workshop, Tokyo, Japan (1998)
# Table of Contents

## Invited Talks

Putting Change at the Center of the Software Process .................. 1  
*Oscar Nierstrasz*

Interface Specification: A Balancing Act .............................. 5  
*Hans Jonkers*

## Generation and Adaptation of Component-Based Systems

An Open Component Model and Its Support in Java ....................... 7  
*Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani*

Software Architectural Support for Disconnected Operation  
in Highly Distributed Environments ........................................ 23  
*Marija Mikic-Rakic and Nenad Medvidovic*

Using Smart Connectors to Resolve Partial Matching Problems  
in COTS Component Acquisition ............................................ 40  
*Hyun Gi Min, Si Won Choi, and Soo Dong Kim*

Correctness of Component-Based Adaptation ............................ 48  
*Sandeep S. Kulkarni and Karun N. Biyani*

Strategies for a Component-Based Self-adaptability Model  
in Peer-to-Peer Architectures ............................................. 59  
*Sascha Alda and Armin B. Cremers*

## Tools and Building Framework

Classifying Software Component Interoperability Errors  
to Support Component Adaption ............................................ 68  
*Steffen Becker, Sven Overhage, and Ralf H. Reussner*

Correct Components Assembly  
for a Product Data Management Cooperative System ................... 84  
*Massimo Tivoli, Paola Inverardi, Valentina Presutti, Alessandro Forghieri, and Maurizio Sebastianis*

The Release Matrix for Component-Based Software Systems .......... 100  
*Louis J.M. Taborda*
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewpoints for Specifying Component-Based Systems</td>
<td>114</td>
</tr>
<tr>
<td>Gerald Kotonya and John Hutchinson</td>
<td></td>
</tr>
<tr>
<td>CMEH: Container Managed Exception Handling</td>
<td>122</td>
</tr>
<tr>
<td>for Increased Assembly Robustness</td>
<td></td>
</tr>
<tr>
<td>Kevin Simons and Judith Stafford</td>
<td></td>
</tr>
<tr>
<td>A Framework for Constructing Adaptive Component-Based Applications:</td>
<td>130</td>
</tr>
<tr>
<td>Concepts and Experiences</td>
<td></td>
</tr>
<tr>
<td>Humberto Cervantes and Richard S. Hall</td>
<td></td>
</tr>
<tr>
<td>Testing Framework Components</td>
<td>138</td>
</tr>
<tr>
<td>Benjamin Tyler and Neelam Soundarajan</td>
<td></td>
</tr>
<tr>
<td><strong>Components for Real-Time Embedded Systems</strong></td>
<td></td>
</tr>
<tr>
<td>Industrial Requirements on Component Technologies for Embedded Systems</td>
<td>146</td>
</tr>
<tr>
<td>Anders Möller, Joakim Fröberg, and Mikael Nolin</td>
<td></td>
</tr>
<tr>
<td>Prediction of Run-Time Resource Consumption</td>
<td>162</td>
</tr>
<tr>
<td>in Multi-task Component-Based Software Systems</td>
<td></td>
</tr>
<tr>
<td>Johan Muskens and Michel Chaudron</td>
<td></td>
</tr>
<tr>
<td>Design Accompanying Analysis of Component-Based Embedded Software</td>
<td>178</td>
</tr>
<tr>
<td>Walter Maydl</td>
<td></td>
</tr>
<tr>
<td>Introducing a Component Technology</td>
<td>194</td>
</tr>
<tr>
<td>for Safety Critical Embedded Real-Time Systems</td>
<td></td>
</tr>
<tr>
<td>Kristian Sandström, Johan Fredriksson, and Mikael Åkerholm</td>
<td></td>
</tr>
<tr>
<td>A Hierarchical Framework for Component-Based Real-Time Systems</td>
<td>209</td>
</tr>
<tr>
<td>Giuseppe Lipari, Paolo Gai, Michael Trimarchi, Giacomo Guidi, and Paolo Ancilotti</td>
<td></td>
</tr>
<tr>
<td><strong>Extra-Funrtional Properties of Components and Component-Based Systems</strong></td>
<td></td>
</tr>
<tr>
<td>Extra-Functional Contract Support in Components</td>
<td>217</td>
</tr>
<tr>
<td>Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau</td>
<td></td>
</tr>
<tr>
<td>CB-SPE Tool: Putting Component-Based Performance Engineering into Practice</td>
<td>233</td>
</tr>
<tr>
<td>Antonia Bertolino and Raffaela Mirandola</td>
<td></td>
</tr>
<tr>
<td>Component Technology and QoS Management</td>
<td>249</td>
</tr>
<tr>
<td>George T. Heineman, Joseph Loyall, and Richard Schantz</td>
<td></td>
</tr>
<tr>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Computational Quality of Service for Scientific Components</td>
<td>264</td>
</tr>
<tr>
<td>Boyana Norris, Jaideep Ray, Rob Armstrong, Lois C. McInnes,</td>
<td></td>
</tr>
<tr>
<td>David E. Bernholdt, Wael R. Elwasif, Allen D. Malony,</td>
<td></td>
</tr>
<tr>
<td>and Sameer Shende</td>
<td></td>
</tr>
<tr>
<td>A Framework for Reliability Assessment of Software Components</td>
<td>272</td>
</tr>
<tr>
<td>Rakesh Shukla, Paul Strooper, and David Carrington</td>
<td></td>
</tr>
<tr>
<td><strong>Measurements and Prediction Models</strong></td>
<td></td>
</tr>
<tr>
<td><strong>for Component Assemblies</strong></td>
<td></td>
</tr>
<tr>
<td>Performance Prediction for Component Compositions</td>
<td>280</td>
</tr>
<tr>
<td>Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer</td>
<td></td>
</tr>
<tr>
<td>TESTEJB - A Measurement Framework for EJBs</td>
<td>294</td>
</tr>
<tr>
<td>Marcus Meyerhöfer and Christoph Neumann</td>
<td></td>
</tr>
<tr>
<td>Model-Based Transaction Service Configuration</td>
<td>302</td>
</tr>
<tr>
<td>for Component-Based Development</td>
<td></td>
</tr>
<tr>
<td>Sten Loecher</td>
<td></td>
</tr>
<tr>
<td><strong>Author Index</strong></td>
<td>311</td>
</tr>
</tbody>
</table>
Putting Change at the Center of the Software Process

Oscar Nierstrasz

Software Composition Group, University of Bern

www.iam.unibe.ch/~scg

Introduction

For over thirty years now, software components have been perceived as being essential stepping stones towards flexible and maintainable software systems. But where do the components come from? Once we have the components, how do we put them together? And when we are missing components, how should we synthesize them?

Lehman and Belady established in a classic study that a number of “Laws” of Software Evolution apply to successful software projects [10]. Of these, the two most insightful are perhaps:

- Continuing change: A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- Increasing complexity: As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

In this light we can observe that many recent trends in software engineering can actually be seen as obstacles to progress, since they offer metaphors that do not help address these issues [11]. “Software Engineering” itself can be seen as a dangerous metaphor that draws too strong an analogy between engineering of hardware and software. Similarly “software maintenance” is clearly a lie when we consider that real maintenance tasks are actually continuous development.

We know that successful software systems are doomed to change. But our programming languages and tools continue to focus on developing static, unchanging models of software. We propose that change should be at the center of our software process. To that end, we are exploring programming language mechanisms to support both fine-grained composition and coarse-grained extensibility, and we are developing tools and techniques to analyse and facilitate change in complex systems. In this talk we review problems and limitations with object-oriented and component-based development approaches, and we explore both technological and methodological ways in which change can be better accommodated.

---

Language Support for Composition

What programming languages provide specific mechanisms that either take into account or support the fact that programs change over time? It is notable that mainstream programming languages largely emphasize the construction of static software structures, and disregard the fact that these structures are likely to change. We have been experimenting with various programming languages and language extensions that address certain aspects of change.

Piccola is a small language for composing applications from software components [1,13]. Whereas we have many programming languages that are well-suited for building components, few focus on how components are put together. Piccola provides a notion of first-class namespaces that turns out to be immensely useful for expressing, packaging and controlling the ways in which software components are composed [12].

Traits are a fine-grained mechanism for decomposing classes into sets of related methods [15]. Traits overcome a number of difficulties with single and multiple inheritance, while avoiding the fragility inherent in mixins by sidestepping traditional linearization algorithms for composing features. Traits have proven to be extremely useful in refactoring complex class libraries [6].

Classboxes offer a minimal module system for controlling class extensions [4]. Class extensions support unanticipated change to third-party classes where subclassing is not an option. In classboxes, as in traits and Piccola, we note that the notion of first-class namespaces is an important means to manage and control change. We conjecture that programming languages that better support change will place more emphasis on such mechanisms.

Mining Components

Support for change is clearly not just a language issue. We also need good tools to analyze and manage code.

We have been developing a reengineering platform called Moose that serves as a code repository and a basis for analyzing software systems [7]. In this context we have developed a series of tools to aid in the understanding and restructuring of complex software systems.

CodeCrawler is a software visualization tool based on the notion of polymetric views — simple graphical visualizations of direct software metrics [8]. One of the most striking applications of polymetrics views is in analyzing the evolution of a software system [9]: an evolution matrix quickly reveals which parts of a system are stable or undergoing change. We are further exploring the use of historical data to predict change in software systems [14].

We are also exploring ways to mine recurring structures from software systems. ConAn is a tool that applies formal concept analysis to detect recurring “concepts” in models of software. We have applied this approach to detect implicit contracts in class hierarchies [3] and to detect recurring “software patterns” [2]. We are now exploring ways to assess and improve the quality of the module structure of applications with respect to various reengineering operations.
Where Are We? Where Do We Go?

To conclude, we would do well to note that change is inevitable in software. As a consequence, software components, being the stable part of software systems, can offer at most half of any equation that would help to improve software productivity.

There is a need for both languages and tools that offer better support to help us cope with and even exploit change.

Nevertheless, we should beware that any new techniques or methods carry some danger with them. Not only do metaphors sometimes blind us, but, as Berry points out [5], any technique that addresses a key difficulty in software development typically entails some painful steps that we will seek to avoid. To achieve any benefit, we must first overcome this pain.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004).

References


Interface Specification: A Balancing Act
(Extended Abstract)

Hans Jonkers
Philips Research Laboratories Eindhoven
hans.jonkers@philips.com

Interface specifications play a vital role in component-based software development. A proper component interface specification acts as a contract between component developers and application developers. It defines which properties of the component are guaranteed by the component developer and can be relied on by the application developer. This allows component development and application development to be decoupled. The application developer can create applications without knowledge of the component implementation and the component developer can create and modify implementations without breaking existing application code. In addition, interface specifications can be used for several other purposes such as component verification (black-box testing) and code generation.

A serious practical question is how much effort to put in writing interface specifications. Constructing good interface specifications is difficult and most software developers dislike writing (interface) specifications. Aiming at the highest quality interface specifications, e.g. by using formal specification techniques, is expensive and requires highly skilled developers. Spending minimal effort on constructing interface specifications can also be expensive except that the costs are incurred later in the component life cycle by increased maintenance costs, customer dissatisfaction, etc. In practice the right balance between these two extremes has to be found. In finding the balance several issues have to be taken into account such as the expected lifetime and usage of the interfaces, the skills of the readers and writers of the specifications, how critical the interfaces are, etc.

ISpec is an approach to interface specification that evolved in Philips from early work on formal specification techniques and the application of these techniques in several industrial development projects. The reality checks made it evolve from a classical closed formal methods approach to an open-ended approach supporting the “balanced” development of interface specifications. It has borrowed from many mainstream approaches such as object-oriented modeling, design by contract and design patterns.

ISpec is neither a method nor a language. It can be characterized as a coherent framework of concepts, principles and techniques aimed at improving the quality of interface specifications in a cost-effective way. In applying the framework several trade-offs can be made, such as formality versus informality, declarative versus operational style of specification, and completeness versus conciseness. The framework can also be customized to the context it is used in, e.g. to comply with project standards and local culture. The ability to make these trade-offs and customizations is essential in making interface specifications cost-effective and in gaining the acceptance of software developers.
The basic concepts and principles of ISpec can be summarized as follows:

- **Interface Suites.** The units of specification are groups of mutually related interfaces, so-called *interface suites*, rather than individual interfaces. They act as “specification components” that can be used to construct specifications of (implementation) components.

- **Contracts and Roles.** The specification of an interface suite is considered a contract identifying *roles* that are played (by component and application code) with respect to interfaces and explicit *rights* and *obligations* associated with the roles. In UML terms, roles can be viewed as abstract classes.

- **Closed World Assumption.** An interface specification is considered a closed world: it defines the complete set of rights and obligations of providers and users of the interfaces. In particular, when composing interface specifications no new rights and obligations may be associated with existing roles (contracts are immutable), which is the basis for the compositionality of interface specifications.

- **Interface-Role Models.** Interface specifications are model-based. The external signature of the model is defined by a UML class diagram (the *interface-role model*) defining the signatures of and the relations between interfaces and roles such as the “provides” and “requires” relations. The interface-role model acts as the “root” of each interface specification.

- **Specification Templates.** The model is defined by means of a number of standard *specification templates* associated with the various model elements (including roles and interfaces). The (graphical) interface-role model together with the (textual) contents of the specification templates fully define the model. All other information such as additional UML diagrams is considered derived information.

- **Language Plug-ins.** The interface role diagram and the structure of the specification templates are largely (specification) language independent. Language is an issue in those “holes” of the specification templates where constraints or expressions are expected. The concept of *language plug-ins* is used to associate languages with these holes, where languages may vary from completely informal (the default) to completely formal.

- **Extended Pre and Post-conditions.** The main declarative style of specification is based on an extension of the classical pre and post-condition style with an *action clause*, allowing more operational aspects of methods such as out-calls, synchronization, etc. to be expressed.
An Open Component Model and Its Support in Java

Eric Bruneton¹, Thierry Coupaye¹, Matthieu Leclercq², Vivien Quema², and Jean-Bernard Stefani²

¹ France Telecom R&D
² INRIA Rhône-Alpes

Abstract. This paper presents Fractal, a hierarchical and reflective component model with sharing. Components in this model can be endowed with arbitrary reflective capabilities, from black-boxes to components that allow a fine-grained manipulation of their internal structure. The paper describes Julia, a Java implementation of the model, a small but efficient run-time framework, which relies on a combination of interceptors and mixins for the programming of reflective features of components. The paper presents a qualitative and quantitative evaluation of this implementation, showing that component-based programming in Fractal can be made very efficient.

1 Introduction

By enforcing a strict separation between interface and implementation and by making software architecture explicit, component-based programming can facilitate the implementation and maintenance of complex software systems [22]. Coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance, as exemplified e.g. by the container concept in Enterprise Java Beans (EJB), CORBA Component Model (CCM), or Microsoft.Net [22].

Existing component-based frameworks and architecture description languages, however, provide only limited support for extension and adaptation, as witnessed by recent works on component aspectualization, e.g. [9,17,19]. This limitation has several important drawbacks: it prevents the easy and possibly dynamic introduction of different control facilities for components such as non-functional aspects; it prevents application designers and programmers from making important trade-offs such as degree of configurability vs performance and space consumption; and it can make difficult the use of these frameworks and languages in different environments, including embedded systems.

We present in this paper a component model, called Fractal, that alleviates the above problems by introducing a notion of component endowed with an open set of control capabilities. In other terms, components in Fractal are reflective, and their reflective capabilities are not fixed in the model but can be extended.

© Springer-Verlag Berlin Heidelberg 2004
and adapted to fit the programmer’s constraints and objectives. Importantly, we also present in this paper how such an open component model can be efficiently supported by an extensible run-time framework.

The main contributions of the paper are as follows:

- We define a hierarchical component model with sharing, that supports an extensible set of component control capabilities.
- We show how this model can be effectively supported by means of an extensible software framework, that provides for both static and dynamic configurability.
- We show that our component model and run-time framework can be effectively used to build highly configurable, yet efficient, distributed systems.

The paper is organized as follows. Section 2 presents the main features of the Fractal model. Section 3 describes JULIA, a Java framework that supports the Fractal model. Section 4 evaluates the model and its supporting framework. Section 5 discusses related work. Section 6 concludes the paper with some indications for future work.

2 The Fractal Component Model

The Fractal component model (see [6] for a detailed specification), is a general component model which is intended to implement, deploy and manage (i.e. monitor and dynamically configure) complex software systems, including in particular operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of applications at various levels of abstraction), shared components (to model resources and resource sharing while maintaining component encapsulation), introspection capabilities (to monitor a running system), and re-configuration capabilities (to deploy and dynamically configure a system). In order to allow programmers to tune the reflective features of components to the requirements of their applications, Fractal is defined as an extensible system of relations between selected concepts, where components can be endowed with different forms of control (reflective features).

A Fractal component is a run-time entity that is encapsulated, and that has a distinct identity. At the lowest level of control, a Fractal component is a black box, that does not provide any introspection or intercession capability. Such components, called base components are comparable to plain objects in an object-oriented programming language such as Java. Their explicit inclusion in the model facilitates the integration of legacy software.

An interface is an access point to a component (similar to a “port” in other component models), that supports a finite set of operations. Interfaces can be of two kinds: server interfaces, which correspond to access points accepting incoming operation invocations, and client interfaces, which correspond to access points supporting outgoing operation invocations. At the level of control immediately above the base level, a Fractal component provides a Component
interface, similar to the IUnknown in the COM model, that allows one to discover all its external (client and server) interfaces. Each interface has a name that distinguishes it from other interfaces of the component.

At upper levels of control, a Fractal component exposes (part of) its internal structure. A Fractal component is composed of a membrane, which provides external interfaces to introspect and reconfigure its internal features (called control interfaces), and a content, which consists in a finite set of other components (called sub-components). The membrane of a component can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are only accessible from the component’s sub-components. The membrane of a component is typically composed of several controller and interceptor objects. A controller object implements control interfaces. Typically, a controller object can provide an explicit and causally connected representation of the component’s sub-components and superpose a control behavior to the behavior of the component’s sub-components, including suspending, check pointing and resuming activities of these sub-components. Interceptor objects are used to export the external interface of a subcomponent as an external interface of the parent component. They intercept the oncoming and outgoing operation invocations of an exported interface and they can add additional behavior to the handling of such invocations (e.g. pre and post-handlers). Each component membrane can thus be seen as implementing a particular semantics of composition for the component’s sub-components. Controller and interceptors can be understood as meta-objects or meta-groups (as they appear in reflective languages and systems).

The Fractal model allows for arbitrary (including user defined) classes of controller and interceptor objects. This is the main reason behind the denomination “open component model”. The Fractal specification, however, contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. The following are examples of controllers.

Attribute controller: An attribute is a configurable property of a component. A component can provide an AttributeController interface to expose getter and setter operations for its attributes.

Binding controller: A component can provide the BindingController interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings (see below).

Content controller: A component can provide the ContentController interface to list, add and remove subcomponents in its contents.

Life-cycle controller: A component can provide the LifeCycleController interface to allow explicit control over its main behavioral phases, in support for dynamic reconfiguration. Basic lifecycle methods supported by a LifeCycleController interface include methods to start and stop the execution of the component.

Communication between Fractal components is only possible if their interfaces are bound. Fractal supports both primitive bindings and compos-
ite bindings. A *primitive binding* is a binding between one client interface and one server interface in the same address space, which means that operation invocations emitted by the client interface should be accepted by the specified server interface. A primitive binding is called that way for it can be readily implemented by pointers or direct language references (e.g. Java references).

A *composite binding* is a communication path between an arbitrary number of component interfaces. These bindings are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc). A binding is a normal FRACTAL component whose role is to mediate communication between other components. The binding concept corresponds to the connector concept that is defined in other component models. Note that, except for primitive bindings, there is no predefined set of bindings in FRACTAL. In fact bindings can be built explicitly by composition, just as other components. The FRACTAL model thus provides two mechanisms to define the architecture of an application: bindings between component interfaces, and encapsulation of a group of components in a composite.

An original feature of the FRACTAL component model is that a given component can be included in several other components. Such a component is said to be *shared* between these components. Shared components are useful, paradoxically, to preserve component encapsulation: there is no need to expose interfaces in higher-level components to allow access to a shared component by a lower-level one. Shared components are useful in particular to faithfully model access to low-level system resources.

The FRACTAL model is endowed with an optional type system (some components such as base components need not adhere to the type system). Interface types are pairs *(signature,role)*. Component types reflect the different interface types that a component can bear. For lack of space, we do not detail the FRACTAL type system here.

## 3 The Julia Framework

The JULIA framework supports the construction of software systems with FRACTAL components written in Java. The main design goal for JULIA was to implement a framework to program FRACTAL component membranes. In particular, we wanted to provide an extensible set of control objects, from which the user can freely choose and assemble the controller and interceptor objects he or she wants, in order to build the membrane of a FRACTAL component. The second design goal was to provide a continuum from static configuration to dynamic reconfiguration, so that the user can make the speed/memory tradeoffs he or she wants. The last design goal was to implement a framework that can be used on any JVM and/or JDK, including very constrained ones such as the KVM, and the J2ME profile (where there is no ClassLoader class, no reflection API, no collection API, etc). In addition to the previous design goals, we also made two hypotheses in order to simplify the implementation: we suppose there is only one (re)configuration thread at a given time, and we also suppose that
the component data structures do not need to be protected against malicious components.

3.1 Main Data Structures

Overview. A Fractal component is generally represented by many Java objects, which can be separated into three groups (see Fig. 1):

- the objects that implement the component interfaces, in white in Fig. 1 (one object per component interface; each object has an `impl` reference to an object that really implements the Java interface, and to which all method calls are delegated; this reference is `null` for client interfaces),
- the objects that implement the membrane of the component, in gray and light gray in the figure (a controller object can implement zero of more control interfaces),
- and the objects that implement the content part of the component (not shown in the figure).

The objects that represent the membrane of a component can be separated into two groups: the objects that implement the control interfaces (in gray in Fig. 1), and (optional) `interceptor` objects (in light gray) that intercept incoming and/or outgoing method calls on non-control interfaces. These objects implement respectively the `Controller` and the `Interceptor` interfaces. Each controller and interceptor object can contain references to other controller / interceptor objects (since the control aspects are generally not independent - or “orthogonal” - they must generally communicate between each other).

3.2 Mixin Classes

Motivations. The main design goal of JULIA is to implement a `framework` to program component membranes. To do so, JULIA provides a collection of pre-defined controller and interceptor classes and a class mixin mechanism. Mixin classes are used to build controller classes that combine several aspects.
In order to provide these frameworks, one could have thought of using class inheritance. But this solution is not feasible, because it leads to a combinatorial explosion, and to a lot of code duplication. Another solution could have been to use an Aspect Oriented Programming (AOP) tool or language, such as AspectJ [11]. But using AspectJ would have introduced a new problem, due to the fact that, in AspectJ, aspects must be applied at compile time, and that this process requires the source code of the "base" classes\(^1\). It would then be impossible to distribute JULIA in compiled form, in a jar file, because then the users would not be able to apply new aspects to the existing JULIA classes (in order to add new control aspects that crosscut existing ones).

What is needed to really solve our modularity and extensibility problem is therefore a kind of AOP tool or language that can be used at load time or at runtime, without needing the source code of the base classes, such as JAC [16]. The current JULIA version does not use JAC or other similar systems: it uses instead some kind of *mixin* classes. A mixin class is a class whose super class is specified in an abstract way, by specifying the minimum set of fields and methods it should have. A mixin class can therefore be applied (i.e. override and add methods) to any super class that defines at least these fields and methods. This property solves the above combinatorial problem. The AspectJ problem is solved by the fact that the mixin classes used in JULIA can be mixed at load time, thanks to our bytecode generator [23] (unlike in most mixin based inheritance languages, where mixed classes are declared at compile time).

**Implementation.** Instead of using a Java extension to program the mixin classes, which would require an extended Java compiler or a pre processor, mixin classes in JULIA are programmed by using patterns. For example the JAM [3] mixin class shown below (on the left) is written in pure Java as follows (on the right):

```
mixin A {
    inherited public void m () {
        public int count;
        public void m () {
            ++count;
            super.m();
        }
    }
}

abstract class A {
    abstract void _super_m () {
        public int count;
        public void m () {
            ++count;
            _super_m();
        }
    }
}
```

In other words, the _super_ prefix is used to denote the inherited members in JAM, i.e. the members that are required in a base class, for the mixin class to be applicable to it. More precisely, the _super_ prefix is used to denote methods that are overridden by the mixin class. Members that are required but not overridden are denoted with _this_.

Mixin classes can be mixed, resulting in normal classes. More precisely, the result of mixing several mixin classes \(M_1, \ldots, M_n\), *in this order*, is a normal class that is equivalent to a class \(M_n\) extending the \(M_{n-1}\) class, itself extending the

\(^1\) This is no longer true with version 1.1 of AspectJ, however this was the case in 2002 when JULIA was developed.
M_{n-2} class, ... itself extending the M_1 class (constructors are ignored; an empty public constructor is generated for the mixed classes). Several mixin classes can be mixed only if each method and field required by a mixin class M_i is provided by a mixin class M_j, with j < i (each required method and field may be provided by a different mixin).

### 3.3 Interceptor Classes

This section gives some details about the generator used to generate interceptor classes. This bytecode generator takes as parameters the name of a super class, the name(s) of one or more application specific interface(s), and one or more aspect code generator(s). It generates a sub class of the given super class that implements all the given application specific interfaces and that, for each application specific method, implements all the aspects corresponding to the given aspect code generators.

Each aspect code generator can modify the bytecode of each application specific method arbitrarily. For example, two aspect code generators A and B can modify the method void m() { impl.m() } into:

```java
void m () {
    /*pre code A*/
    try {
        impl.m();
    } finally { /*post code A*/ }
}
```

```java
void m () {
    /*pre code B*/
    try {
        impl.m();
    } finally { /*post code B*/ }
}
```

When an interceptor class is generated by using several aspect bytecode generators, the transformations performed by these generators are automatically composed together. For example, if A and B are used to generate an interceptor class, the result for the previous m method is the following (there are two possibilities, depending on the order in which A and B are used):

```java
void m () {
    /*pre code A*/
    try {
        /*pre code B*/
        impl.m();
        /*post code B*/
    } finally { /*post code A*/ }
}
```

```java
void m () {
    /*pre code B*/
    try {
        /*pre code A*/
        impl.m();
        /*post code A*/
    } finally { /*post code B*/ }
}
```

Note that, thanks to this (elementary) automatic weaving, which is very similar to what can be found in AspectJ or in Composition Filters [5], several aspects can be managed by a single interceptor object: there is no need to have chains of interceptor objects, each object corresponding to an aspect.

Like the controller objects, the aspects managed by the interceptor objects of a given component can all be specified by the user when the component is created. The user can therefore not only choose the control interfaces he or she wants, but also the interceptor objects he or she wants. JULIA only provides two
aspect code generators: one to manage the lifecycle of components, the other to
trace incoming and/or outgoing method calls. JULIA also provides two abstract
code generators, named SimpleCodeGenerator and MetaCodeGenerator, that can
be easily specialized in order to implement custom aspect code generators.

Note: each aspect bytecode generator is given a Method object corresponding
to the method for which it must generate interception code. Thanks to this
argument, a generator can generate bytecode that adapts to the specific signature
of each method. It can also generate bytecode to reify the method’s parameters
if desired (although this is less efficient than the first method).

3.4 Support for Constrained Environments

One of the goals of JULIA is to be usable even with very constrained JVMs and
JDKs, such as the KVM and the J2ME libraries (CLDC profile). This goal is
achieved thanks to the following properties.

- The size of the JULIA runtime (35KB, plus 10KB for the Fractal API),
  which is the only part of JULIA (175 KB as a whole) that is needed at run-
time, is compatible with the capabilities of most constrained environments.
- JULIA can be used in environments that do not provide the Java Reflection
  API or the ClassLoader class, which are needed to dynamically generate the
  JULIA application specific classes, since these classes can also be generated
  statically, in a less constrained environment.
- The JULIA classes that are needed at runtime, or whose code can be copied
  into application specific runtime classes, use only the J2ME, CLDC profile
  APIs, with only two exceptions for collections and serialization. For collec-
tions a subset of the JDK 1.2 collection API is used. This API is not available
in the CLDC profile, but a bytecode modification tool is provided with Ju-
ilia to convert classes that use this subset into classes that use the CLDC
APIs instead. This tool also removes all serialization related code in JULIA.
In other words the JULIA jars cannot be used directly with CLDC, but can
be transformed automatically in new jars that are compatible with this API.

3.5 Optimizations

Intra component optimizations. In order to save memory, JULIA provides opti-
mization options to merge some or most of the objects that constitute a com-
ponent into a single object. This merging is done thanks to a bytecode class
generator that can merge several controller classes into a single class provided
that these classes respect the following rules:

- Each controller object can provide and require zero or more Java interfaces.
The provided interfaces must be implemented by the object, and there must
be one field per required interface, whose name must begin with weaveable
for a mandatory interface, or weaveableOpt for an optional interface (see
below). Each controller class that requires at least one interface must also
implement the Controller interface (see below).
– In a given component, a given interface cannot be provided by more than one object (except for the Controller interface). Otherwise it would be impossible to merge these objects (an object cannot implement a given interface in several ways).

The merging process is the following. Basically, all the methods and fields of each class are copied into a new class (the resulting class does not depend on the order into which the classes are copied). However the fields whose name begins with weaveable are replaced by this, and those whose name begins with weaveableOpt are replaced either by this, if a class that implements the corresponding type is present in the list of the classes to be merged, or null otherwise.

Inter component optimizations. In addition to the previous intra component optimizations, which are mainly used to save memory, JULIA also provides an inter component optimization, namely an algorithm to create and update shortcut bindings between components, and whose role is to improve time performances. As explained above, each interface of a component contains an impl reference to an object that really implements the component interface. In the case of a server interface s, this field generally references an interceptor object, which itself references another server interface.

More precisely, this is the case with the CompositeBindingMixin. With the OptimizedCompositeBindingMixin, the impl references are optimized when possible. For example, in Fig. 2, since I1 does not have an associated interceptor object, and since component interface objects such as I2 just forward any incoming method calls to the object referenced by their impl field, I1 can, and effectively references directly the interceptor associated to I2.

4 Evaluation

We provide in this section an evaluation of our model and its implementation. We first provide a qualitative assessment of our component framework. We then
provide a more quantitative evaluation with micro-benchmarks and with an application benchmark based on a reengineered message-oriented middleware.

4.1 Qualitative Assessment

Modularity. JULIA provides several mixins for the binding controller interface, two implementations of the life cycle controller interface, and one implementation of the content controller interface. It also provides support to control component attributes, and to associate names to components. All these aspect implementations, which make different flexibility/performance tradeoffs, are well separated from each other thanks to mixins, and can therefore be combined freely. Together with the optimization mechanisms used in JULIA, this flexibility provides what we call a continuum from static to dynamic configurations, i.e., from unreconfigurable but very efficient configurations, to fully dynamically reconfigurable but less efficient configurations (it is even possible to use different flexibility/performance tradeoffs for different parts of a single application).

Extensibility. Several users of JULIA have extended it to implement new control aspects, such as transactions [20], auto-adaptability [8], or checking of the component’s behavior, compared to a formal behavior, expressed for example with assertions (pre/post conditions and invariants), or with more elaborate formalisms, such as temporal logic [21]. As discussed below, we have also built with JULIA a component library, called DREAM, for building message-oriented middleware (MOM) and reengineered and existing MOM using this library. DREAM components exhibit specific control aspects, dealing with on-line deployment and re-configuration. In all these experiences, the different mechanisms in JULIA have proved sufficient to build the required control aspects.

Limitations. There are however some limitations to JULIA’s modularity and extensibility. For example, when we implemented JULIA, it was sometimes necessary to refactor an existing method into two or more methods, so that one of this new methods could be overridden by a new mixin, without overriding the others. In other words, the mixin mechanism is not sufficient by itself: the classes must also provide the appropriate “hooks” to apply the mixins. And it is not easy, if not impossible, to guess the hooks that will be necessary for future aspects (but this problem is not specific to mixins, it also occurs in AspectJ, for example).

4.2 Quantitative Evaluation I: Micro-benchmarks

In order to measure the memory and time overhead of components in JULIA, compared to objects, we measured the memory size of an object, and the duration of an empty method call on this object, and we compared these results to the memory size of a component\(^2\) (with a binding controller and a life cycle

\(^2\) The size of the objects that represent the component’s type, which is shared between all components of the same type, is not taken into account here. This size is of the order of 1500 bytes for a component with 6 interfaces.
controller) encapsulating this object, and to the duration of an empty method call on this component. The results are given in Table 1, for different optimization options. The measurements were made on a Pentium III 1GHz, with the JDK1.3, HotSpotVM, on top of Linux. In these conditions the size of an empty object is 8 bytes, and an empty method call on an interface lasts 0.014 $\mu$s.

As can be seen the class merging options can reduce the memory overhead of components (merging several objects into a single one saves many object headers, as well as fields that were used for references between these objects). The time overhead without interceptor is of the order of one empty method call: it corresponds to the indirectness through an interface reference object. With a lifecycle interceptor, this overhead is much greater: it is mainly due to the execution time of two synchronized blocks, which are used to increment and decrement a counter before and after the method’s execution. This overhead is reduced in the “merge all” case, because an indirection is saved in this case. In any cases, this overhead is much smaller than the overhead that is measured when using a generic interceptor that completely reifies all method calls (4.6 $\mu$s for an empty method, and 9 $\mu$s for an int inc (int i) method), which shows the advantages of using an open and extensible interceptor code generator.

The time needed to instantiate a component encapsulating an empty object is of the order of 0.3 ms, without counting the dynamic class generation time, while the time to needed instantiate an empty object is of the order of 0.3 $\mu$s (instantiating a component requires to instantiate several objects, and many checks are performed before instantiating a component).

### 4.3 Quantitative Evaluation II: Re-engineering JORAM

Joram [25] is an open-source JMS-compliant middleware (Java Messaging Service [24]). Joram comprises two parts: the ScalAgent message-oriented middleware (MOM) [4], and a software layer on top of it to support the JMS API.

The ScalAgent MOM is a fault-tolerant platform, written in Java, that combines asynchronous message communication with a distributed programming model based on autonomous software entities called agents. Agents behave according to an “event $\rightarrow$ reaction” model. They are persistent and each reaction is instantiated as a transaction, allowing recovery in case of node failure. The ScalAgent MOM comprises a set of agent servers. Each agent server is made

### Table 1. JULIA performances.

<table>
<thead>
<tr>
<th>options</th>
<th>memory overhead (bytes)</th>
<th>time overhead ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lifecycle, no optimization</td>
<td>592</td>
<td>0.110</td>
</tr>
<tr>
<td>lifecycle, merge controllers</td>
<td>528</td>
<td>0.110</td>
</tr>
<tr>
<td>lifecycle, merge all</td>
<td>504</td>
<td>0.092</td>
</tr>
<tr>
<td>no lifecycle, no optimization</td>
<td>496</td>
<td>0.011</td>
</tr>
<tr>
<td>no lifecycle, merge controllers</td>
<td>440</td>
<td>0.011</td>
</tr>
<tr>
<td>no lifecycle, merge all</td>
<td>432</td>
<td>0.011</td>
</tr>
</tbody>
</table>
up of three entities. The Engine is responsible for the creation and execution of agents; it ensures their persistency and atomic reaction. The Conduit routes messages from the engine to the networks. The Networks ensure reliable message delivery and a causal ordering of messages between servers.

Using JULIA, we have built a component-based library, called DREAM, which is dedicated to the construction of asynchronous middleware. For lack of space we do not present the DREAM library here. Let us just note that DREAM components are relatively fine-grained, comprising e.g. components such as messages, message queues, message aggregators and de-aggregators, message routers, and distributed channels. Using DREAM, we have reengineered the ScalAgent MOM. Its main structures (networks, engine and conduit) have been preserved to facilitate the functional comparison between the ScalAgent MOM and its DREAM re-implementation: networks, engine and conduits are implemented as composites that control the execution of DREAM subcomponents. The layer that implements the JMS API in Joram runs unmodified on the DREAM implementation.

Performance comparisons. Measurements have been performed to compare the efficiency of the same benchmark application running on the ScalAgent MOM and on its DREAM implementation. The application involves four agent servers; each one hosts one agent. Agents in the application are organized in a virtual ring. One agent is an initiator of rounds. Each round consists in forwarding the message originated by the initiator around the ring. Each agent behaves very simply as follows: each message received by an agent is forwarded to the next agent on the ring until the message has gone full circle. We did two series of tests: messages without payload and messages embedding a 1kB payload. Experiments have been done on four PC Bi-Xeon 1.8 GHz with 1Go, connected by a Gigabit Ethernet adapter, running Linux kernel 2.4.20.

Table 2 shows the average number of rounds per second, and the memory footprint. We have compared two implementations based on DREAM with the ScalAgent implementation. Note that the DREAM implementations do not make use of the JULIA memory optimizations. The first implementation using DREAM is not dynamically reconfigurable. As we can see, the number of rounds is slightly better (∼1.2 to 2%) than in the ScalAgent implementation: this marginal improvement comes from a better design of some low-level components. Concerning the memory footprint, the DREAM implementation requires 9% more memory, which can be explained by some of the structure needed by JULIA and the fact that each component has several controller objects. This memory overhead is not significant on standard PCs. The second implementation is dynamically

<table>
<thead>
<tr>
<th>MOM</th>
<th>Number of rounds</th>
<th>Memory footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 KB</td>
<td>1 KB</td>
</tr>
<tr>
<td>ScalAgent</td>
<td>325</td>
<td>255</td>
</tr>
<tr>
<td>DREAM (non-reconf.)</td>
<td>329</td>
<td>260</td>
</tr>
<tr>
<td>DREAM (reconf.)</td>
<td>318</td>
<td>250</td>
</tr>
</tbody>
</table>
reconfigurable (in particular, each composite component supports a life-cycle controller and a content controller). This implementation is only slightly slower than the ScalAgent one ($\approx 2.2$ to $2\%$) and only requires $7\,\text{KB}$ more than the non-reconfigurable implementation made using DREAM. Note that DREAM performances are proportionally better with the $1\,\text{KB}$ messages than with the empty ones. This is easily explained by the fact that less messages are handled (more time is spent in message transmissions), thus limiting the impact of interceptors.

We have also evaluated the gain brought by changing the configuration in a multi-engine agent server. Such a server can be very useful to parallelize the execution of agents and to provide different non-functional properties to different sets of agents. In contrast to the ScalAgent MOM, such a change of configuration is trivial to make in a DREAM implementation (it can be done dynamically in a reconfigurable implementation). We have compared four different architectures: the ScalAgent one, and equivalent DREAM configurations with four mono-engine, two 2-engine, or one 4-engine agent server(s). Contrary to the previous experiment, agent servers are hosted by the same PC. Also, agents are placed so that two consecutive agents in the virtual ring are hosted by different agent servers. Table 3 shows that using two 2-engine agent servers improves the number of rounds by $18\%$ and reduces the memory footprint by $47\%$. The increase of the number of rounds can be explained by the fact that matrix clocks used to enforce the causal ordering have a $n^2$ size, $n$ being the number of agent servers. Thus, limiting the number of agent servers reduces the size of the matrix which is sent with messages. Table 3 also shows that using a 4-engine agent servers is 29 (35 for 1KB messages) times faster than using four mono-engine agent servers. This result comes from the fact that inter agent communication do not transit via the network components. Instead, the router component that implements the conduit in the DREAM implementation directly sends the message to the appropriate engine. Again, such a change is trivial to make using DREAM, but would require important changes in the ScalAgent code.

### Table 3. Impact of the number of engines by agent server.

<table>
<thead>
<tr>
<th>MOM</th>
<th>Number of rounds</th>
<th>Memory footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 KB</td>
<td>1 KB</td>
</tr>
<tr>
<td>ScalAgent</td>
<td>182</td>
<td>150</td>
</tr>
<tr>
<td>DREAM (4 agent servers)</td>
<td>188</td>
<td>153</td>
</tr>
<tr>
<td>DREAM (2 agent servers)</td>
<td>222</td>
<td>181</td>
</tr>
<tr>
<td>DREAM (1 agent server)</td>
<td>6597</td>
<td>6445</td>
</tr>
</tbody>
</table>

5 Related Work

Component models. The FRACIAL model occupies an original position in the vast amount of work dealing with component-based programming and software architecture [22,14,12], because of its combination of features: hierarchical components with sharing, support for arbitrary binding semantics between compo-
ponents, components with selective reflection. Aside from the fact that sharing is rarely present in component models (an exception is [15]), most component models provide little support for reflection (apart from elementary introspection, as exemplified by the second level of control in the FRACtAL model discussed in Section 2). A component model that provides extensive reflection capabilities is OpenCOM [7]. Unlike FRACtAL, however, OpenCOM defines a fixed meta-object protocol for components (in FRACtAL terms, each OpenCOM component comes equipped with a fixed and predetermined set of controller objects). With respect to industrial standards such as EJB and CCM, Fractal constitutes a more flexible and open component model (with hierarchical composites and sharing) which does not embed predetermined non functional services. It is however perfectly possible to implement such services in FRACtAL, as demonstrated e.g. by the development of transactional controllers in [20]. Note also that FRACtAL is targeted at system engineering, for which EJB or CCM would be inadequate.

**Software architecture in Java.** Several component models for Java have been devised in the last five years. Two recent representatives include JiaZZi [13] and ArchJava [1]. Unlike these works, our approach to component-based programming in Java does not rely on language extensions: JULIA is a small run-time library, complemented with simple code and byte-code generators. This coupled, with the reflective character of the FRACtAL model, provides for a more dynamic and extensible basis for component-based programming than JiaZZi or ArchJava. Note that FRACtAL and JULIA directly support arbitrary connector abstractions, through the notion of bindings. We have, for instance, implemented synchronous distributed bindings with an RMI-like semantics just by wrapping the communication subsystem of the Jonathan Java ORB [10], and asynchronous distributed bindings with message queuing and publish/subscribe semantics by similarly wrapping message channels from the DREAM library introduced in the previous section. ArchJava also supports arbitrary connector abstractions [2], but provides little support for component reflection as in FRACtAL and JULIA. Unlike JULIA, however, ArchJava supports sophisticated type checking that guarantees communication integrity (i.e. that components only communicate along declared connections between ports - in FRACtAL, that components only communicate along established bindings between interfaces).

**Combining aspects and components.** The techniques used in JULIA to support the programming of controller and interceptor objects in a FRACtAL component membrane are related to several recent works on the aspectualization of components or component containers, such as e.g. [9,17,19]. The mixin and aspect code generators in JULIA provide a lightweight, flexible yet efficient means to aspectualize components. In line with its design goals, JULIA does not seek to provide extensive language support as AOP tools such as AspectJ or JAC provide. However such language support can certainly be build on top of JULIA. Prose [18] provides dynamic aspect weaving, whose performance appear to be comparable to that of JULIA. However, Prose relies on a modified JVM, which
makes it impractical for production use. In contrast, JULIA can make use of standard JVMs, including JVMs for constrained environments.

6 Conclusion

We have presented the FRACTAL component model and its Java implementation, JULIA. FRACTAL is open in the sense that FRACTAL components are endowed with an extensible set of reflective capabilities (controller and interceptor objects), ranging from no reflective feature at all (black boxes or plain objects) to user-defined controllers and interceptors, with arbitrary introspection and intercession capabilities. JULIA consists in a small run-time library, together with bytecode generators, that relies on mixins and load time aspect weaving to allow the creation and combination of controller and interceptor classes. We have evaluated the effectiveness of the model and its Java implementation, in particular through the re-engineering of an existing open source message-oriented middleware. The simple application benchmark we have used indicates that the performance of complex component-based systems built with JULIA compares favorably with standard Java implementations of functionally equivalent systems. In fact, as our performance evaluation shows, the gains in static and dynamic configurability can also provide significant gains in performance by adapting system configurations to the application context.

FRACTAL and JULIA have already been, and are being used for several developments, by the authors and others. We hope to benefit from these developments to further develop the FRACTAL component technology. Among the ongoing and future work we can mention: the development of a dynamic ADL, the exploitation of containment types and related type systems to enforce architectural integrity constraints such as communication integrity, the investigation of dynamic aspect weaving techniques to augment or complement the JULIA toolset, and the formal specification of the FRACTAL model with a view to assess its correctness and to connect it with formal verification tools.

Availability. JULIA is freely available under an LGPL license at the following URL: http://fractal.objectweb.org.

References

Software Architectural Support for Disconnected Operation in Highly Distributed Environments

Marija Mikic-Rakic and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA, 90089-0781 USA
{marija,neno}@usc.edu

Abstract: In distributed and mobile environments, the connections among the hosts on which a software system is running are often unstable. As a result of connectivity losses, the overall availability of the system decreases. The distribution of software components onto hardware nodes (i.e., deployment architecture) may be ill-suited for the given target hardware environment and may need to be altered to improve the software system's availability. The critical difficulty in achieving this task lies in the fact that determining a software system's deployment that will maximize its availability is an exponentially complex problem. In this paper, we present an automated, flexible, software architecture-based solution for disconnected operation that increases the availability of the system during disconnection. We provide a fast approximative solution for the exponentially complex redeployment problem, and assess its performance.

1 Introduction

The emergence of mobile devices, such as portable computers, PDAs, and mobile phones, and the advent of the Internet and various wireless networking solutions make computation possible anywhere. One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment monitoring, traffic management, damage surveys in times of natural disaster, and so on. Such scenarios present daunting technical challenges: effective understanding of software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data and numbers of devices; and heterogeneity of the software executing across devices. Furthermore, software often must execute on “small” devices, characterized by highly constrained resources such as limited power, low network bandwidth, slow CPU speed, limited memory, and patchy connectivity. We refer to the development of software systems in the described setting as \textit{programming-in-the-small-and-many (Prism)}, in order to distinguish it from the commonly adopted software engineering paradigm of \textit{programming-in-the-large (PitL)} [8].

Applications in the Prism setting are becoming highly distributed, decentralized, and mobile, and therefore highly dependent on the underlying network. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable (involuntary) connectivity losses due to their constant location change.
and lack of wireless network coverage; the costs of wireless connectivity often also induce user-initiated (voluntary) disconnection; and even the highly reliable WAN and LAN connectivity is unavailable 1.5% to 3.3% of the time [25].

For this reason, Prism systems are challenged by the problem of **disconnected operation** [24], where the system must continue functioning in the temporary absence of the network. Disconnected operation forces systems executing on each network host to temporarily operate independently from other hosts. This presents a major challenge for software systems that are highly dependent on network connectivity because each local subsystem is usually dependent on the availability of non-local resources. Lack of access to a remote resource can make a particular subsystem, or even the entire system unusable.

A software system’s **availability** is commonly defined as a degree to which the system suffers degradation or interruption in its service as a consequence of failures of one or more of its components [11]. In the context of Prism systems, where a most common failure is a network failure, we define availability as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions.

A key observation for systems executing in the Prism setting is that the distribution of software components onto hardware nodes (i.e., **deployment architecture**) greatly influences the system’s availability in the face of connectivity losses. However, the parameters that influence the optimal distribution of a system may not be known before the system’s deployment. For this reason, the (initial) software deployment architecture may be ill-suited for the given target hardware environment. This means that a ** redeployment** of the software system may be necessary to improve its availability.

There are several existing techniques that can support various subtasks of redeployment, such as monitoring [10] to assess hardware and software properties of interest, component migration [9] to facilitate redeployment, and dynamic system manipulation [19] to effect the redeployment once the components are migrated to the appropriate hosts. However, the critical difficulty lies in the fact that determining a software system’s deployment that will maximize its availability is an exponentially complex problem. Existing approaches that recognize this (e.g., [2]) still assume that all system parameters are known beforehand and that infinite time is available to calculate the optimal deployment.

This paper presents an automated, flexible solution for disconnected operation that increases the availability of the system in the presence of connectivity losses. Our solution takes into account that limited information about the system and finite (usually small) amount of time to perform the redeployment task will be available. We directly leverage a software system’s architecture in accomplishing this task. **Software architectures** provide abstractions for representing the structure, behavior, and key properties of a software system [20] in terms of the system’s **components**, their interactions (**connectors**), and their configurations (**topologies**).

We increase a system’s availability by (1) monitoring the system; (2) estimating the redeployment architecture; and (3) effecting that architecture. Since estimating the optimal deployment architecture is exponentially complex, we provide a fast approximative solution for increasing the system’s availability, and provide an assessment of its performance. We provide a light-weight, efficient, and adaptable **architec-**
Software Architectural Support for Disconnected Operation

**tural middleware**, called Prism-MW, that enables implementation, execution, monitoring, and automatic (re)deployment of software architectures in the Prism setting. We have evaluated our approach on a series of examples.

The remainder of the paper is organized as follows. Section 2 presents overviews of the techniques that enable our approach, and of related work. Section 3 defines the problem our work is addressing and presents an overview of our approach. Section 4 introduces Prism-MW and its foundation for the disconnected operation support. Sections 5, 6, and 7 present the three stages of the redeployment process: monitoring, redeployment estimation, and redeployment effecting. The paper concludes with the discussion of future work.

## 2 Foundational and Related Work

This section presents three cases of techniques that form the foundation of our approach and a brief overview of existing disconnected operation techniques.

### 2.1 Deployment

Carzaniga et al. [4] propose a comparison framework for software deployment techniques. They identify eight activities in the software deployment process, and compare existing approaches based on their coverage of these activities:

- **Release** – preparing a system for assembly and transfer to the consumer site;
- **Install** – the initial insertion of a system into a consumer site;
- **Activate** – starting up the executable components of the system at the consumer site;
- **Deactivate** – shutting down any executing components of an installed system;
- **Update** – renewing a version of a system;
- **Adapt** – modifying a software system that has previously been installed. Unlike update, adaptation is initiated by local events, such as change in the environment of the consumer site. As will be detailed in Section 7, our approach utilizes system adaptation;
- **Deinstall** – removal of the system from the consumer site; and
- **Retire** – the system is marked as obsolete, and support by the producer is withdrawn.

### 2.2 Mobility

Code mobility can be informally defined as the ability to dynamically change the binding between a code fragments and the location where it is executed. A detailed overview of existing code mobility techniques is given by Fuggetta et al. [9]. They describe three code mobility paradigms: (1) *remote evaluation* allows the proactive shipping of code to a remote host in order to be executed; (2) *mobile agents* are autonomous objects that carry their state and code, and proactively move across the network, and (3) *code-on-demand*, in which the client owns the resources (e.g., data)
needed for the execution of a service, but lacks the functionality needed to perform the service. As detailed in Section 7, our approach leverages the remote evaluation paradigm.

Existing mobile code systems offer two forms of mobility. **Strong mobility** allows migration of both the code and the state of an execution unit to a different computational environment. **Weak mobility** allows code transfers across different environments; the code may be accompanied by some initialization data, but the execution state is not migrated. As described in Section 7, our approach utilizes strong mobility.

### 2.3 Dynamic Reconfigurability

Dynamic reconfigurability encompasses run-time changes to a software system’s configuration via addition and removal of components, connectors, or their interconnections. Oreizy et al. [19] describe three causes of dynamic reconfigurability: (1) corrective, which is used to remove software faults, (2) perfective, used to enhance software functionality, and (3) adaptive, used to enact changes required for the software to execute in a new environment. They also identify four types of architectural reconfigurability: (1) component addition, (2) component removal, (3) component replacement, and (4) structural reconfiguration (i.e., recombining existing functionality to modify overall system behavior). As described in Section 7, our approach utilizes all four types of run-time architectural reconfigurability.

### 2.4 Related Approaches

We have performed an extensive survey of existing disconnected operation approaches, and provided a framework for their classification and comparison in [18]. In this section, we briefly summarize these techniques, and directly compare our approach to I5 [2], the only known approach that explicitly focuses on a system’s deployment architecture and its impact on the system’s availability.

The most commonly used techniques for supporting disconnected operation are:

- **Caching** – locally storing the accessed remote data in anticipation that it will be needed again [13];
- **Hoarding** – prefetching the likely needed remote data prior to disconnection [14];
- **Queueing remote interactions** – buffering remote, non-blocking requests and responses during disconnection and exchanging them upon reconnection [12];
- **Replication and replica reconciliation** – synchronizing the changes made during disconnection to different local copies of the same component [13]; and
- **Multi-modal components** – implementing separate subcomponents to be used during connection and disconnection [24].

None of these techniques change the system’s deployment architecture. Instead, they strive to improve system’s availability by sacrificing either correctness (in the case of replication) or service delivery time (queueing), or by requiring implementation-level changes to the existing application’s code [24].

I5 [2], proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network. This
approach uses minimization of overall remote communication as the criterion for optimality, and therefore does not distinguish reliable, high-bandwidth from unreliable, low-bandwidth links between target hosts. Additionally, solving the BIP model is exponentially complex in the number of software components, and 15 does not attempt to reduce this complexity. This renders the approach applicable only to systems with very small numbers of software components and target hosts. 15 assumes that all characteristics of the software system and the target hardware environment are known before the system’s initial deployment. Therefore, 15 is not applicable to systems whose characteristics, such as frequencies of interactions among software components, are either not known at design time, or may change during the system’s execution.

3 Problem and Approach

3.1 Problem Definition

The distribution of software components onto hardware nodes (i.e., a system’s software deployment architecture, a concept illustrated in Figure 1) greatly influences the system’s availability in the face of connectivity losses. For example, components located on the same host will be able to communicate regardless of the network’s status; this is clearly not the case with components distributed across different hosts. However, the reliability of connectivity (i.e., rate of failure) among the “target” hardware nodes on which the system is deployed is usually not known before the deployment. The frequencies of interaction among software components may also be unknown. For this reason, the initial software deployment architecture may be ill-suited for the given target hardware environment. This means that a redeployment of the software system may be necessary to improve its availability.

The critical difficulty in achieving this task lies in the fact that determining a software system’s deployment architecture that will maximize its availability (referred to as optimal deployment architecture) is an exponentially complex problem. In addition to hardware connectivity and frequencies of software interaction, there are other constraints on a system’s redeployment, including: (1) the available memory on each host; (2) required memory for each software component; and (3) possible restrictions on component locations (e.g., a UI component may be fixed to a selected host). Figure 2 shows a formal definition of the problem. The $\text{mem}_{\text{comp}}$ function captures the required memory for each component. The frequency of interaction between any pair of components is captured via the $\text{freq}$ function. Each host’s available memory is captured via the $\text{mem}_{\text{host}}$ function. The reliability of the link between any pair of
Given:

1. A set $C$ of $n$ components and two functions $freq: C \times C \rightarrow R$ and $mem_{comp}: C \rightarrow R$
   
   $\begin{align*}
   freq(c_i, c_j) &= \begin{cases} 0 & \text{if } c_i = c_j \\
   & \text{frequency of communication between } c_i \text{ and } c_j \text{ if } c_i \neq c_j \end{cases} \\
   mem_{comp}(c) &= \text{required memory for } c
   \end{align*}$

2. A set $H$ of $k$ hardware nodes and two functions $rel: H \times H \rightarrow R$ and $mem_{host}: H \rightarrow R$
   
   $\begin{align*}
   rel(h_i, h_j) &= \begin{cases} 1 & \text{if } h_i = h_j \\
   0 & \text{if } h_i \text{ is not connected to } h_j \text{ if } h_i \neq h_j \end{cases} \\
   mem_{host}(h) &= \text{available memory on host } h
   \end{align*}$

3. A function that restricts locations of software components $loc: C \times H \rightarrow \{0, 1\}$
   
   $\begin{align*}
   loc(c_i, h_j) &= \begin{cases} 1 & \text{if } c_i \text{ can be deployed onto } h_j \\
   0 & \text{if } c_i \text{ cannot be deployed onto } h_j \end{cases}
   \end{align*}$

Problem:

Find a function $f: C \rightarrow H$ such that the system’s overall availability $A$ defined as

$$A = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} (freq(c_i, c_j)^a \cdot rel(f(c_i), f(c_j)))}{\sum_{i=1}^{n} \sum_{j=1}^{m} (freq(c_i, c_j))}$$

is maximized, and the following two conditions are satisfied:

1. $\forall i \in [1,k] \quad (\sum_{j \in [1,k]} mem_{comp}(c_j)) \leq mem_{host}(h_i)$
2. $\forall j \in [1,n] \quad loc(c_j, f(c_j)) = 1$

Note that in the most general case, the number of possible functions $f$ is $k^n$. However, note that some of these deployments may not satisfy one or both of the above two conditions.

Fig. 2. Formal Statement of the Problem.

hosts is captured via the $rel$ function. Using the $loc$ function, deployment of any component can be restricted to a subset of hosts, thus denoting a set of allowed hosts for that component. The criterion function $A$ formally describes a system’s availability as the ratio of the number of successfully completed interactions in the system to the total number of attempted interactions. Function $f$ represents the exponential number of the system’s candidate deployments. To be considered valid, each candidate deployment must satisfy the two conditions. The first condition in the definition states that the sum of memories of the components that are deployed onto a given host may not exceed the available memory on that host. Finally, the second condition states that a component may only be deployed onto a host that belongs to a set of allowed hosts for that component, specified via the $loc$ function.

Our approach relies on the assumption that the given system’s deployment architecture is accessible from some central location. While this assumption may have been fully justified in the past, a growing number of software systems are decentralized to some extent. We recognize this and intend to address this problem in our future work. At the same time, before we would be able to do so, we have to understand and solve the redeployment problem in the more centralized setting.
3.2 Our Approach

Our approach provides an automated, flexible solution for increasing the availability of a distributed system during disconnection, without the shortcomings introduced by existing approaches. For instance unlike [24], our approach does not require any re-coding of the system’s existing functionality; unlike [13], it does not sacrifice the correctness of computations; finally unlike [12] it does not introduce service delivery delays. We directly leverage a software system’s architecture in accomplishing this task. We propose run-time redeployment to increase the software system’s availability by (1) monitoring the system, (2) estimating its redeployment architecture, and (3) effecting the estimated redeployment architecture. Since estimating a system’s redeployment (step 2) is an exponentially complex problem, we provide an approximative solution that shows good performance.

A key insight guiding our approach is that for software systems whose frequencies of interactions among constituent components are stable over a given period of time T, and which are deployed onto a set of hardware nodes whose reliability of connectivity is also stable over T, there exists at least one deployment architecture (called the optimal deployment architecture) that maximizes the availability of that software system for that target environment over the period T.

![Graphical representation of the availability function.](image)

Figure 3 illustrates the system’s availability during the time period T, in terms of our approach’s three key activities. The system’s initial availability is $A_1$. First, the system is monitored over the period $T_M$; during that period the availability remains unchanged. Second, during the period $T_E$, the system’s redeployment is estimated; again system availability remains unchanged. Third, during the time period $T_R$ the system redeployment is performed to improve its availability; the system’s availability will decrease during this activity as components are migrated across hosts (and thus become temporarily unavailable). Once the redeployment is performed, the system’s availability increases to $A_2$, and remains stable for the remainder of the time period $T$.

Performing system redeployment to improve its availability will give good results if the times required to monitor the system and complete its redeployment are negligible with respect to $T$ (i.e., $T_M + T_E + T_R \ll T$). Otherwise, the system’s parameters would be changing too frequently and the system would “thrash” (i.e., it would undergo continuous redeployments to improve the availability for parameter values that change either before or shortly after the redeployment is completed).

4 Prism-MW

Our approach is independent of the underlying implementation platform, but requires scalable, light-weight support for distributed architectures with arbitrary topologies.
For this reason, we leverage an adaptable and extensible architecture implementation infrastructure (i.e., architectural middleware), called Prism-MW [16]. Prism-MW enables efficient and scalable implementation and execution of distributed software architectures in the Prism setting. Furthermore, Prism-MW’s native support for extensibility made it highly suitable for incorporating disconnected operation support. In this section we summarize the design of Prism-MW, and describe in more detail its foundation for the disconnected operation support.

### 4.1 Middleware Design

Prism-MW provides classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. Figure 4 shows the class design view of Prism-MW. The shaded classes constitute the middleware core; the dark gray classes are relevant to the application developer. The design of the middleware is highly modular: the only dependencies among classes are via interfaces and inheritance; the only exception is the Architecture class, which contains multiple Bricks for reasons that are explained below.

*Brick* is an abstract class that encapsulates common features of its subclasses (Architecture, Component, and Connector). The Architecture class records the configuration of its constituent components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system run-time. A distributed application is implemented as a set of interacting Architecture objects. Components in an architecture communicate by exchanging Events, which are routed by Connectors. Finally, Prism-MW associates the IScaffold interface with every Brick. Scaffolds are used to schedule events for delivery and to dispatch events using a pool of threads in a decoupled manner. IScaffold also directly aids architectural awareness [3] by allowing probing of the runtime behavior of a Brick, via different implementations of the IMonitor interface.

---

**Figure 4.** UML class design view of Prism-MW. Middleware core classes are highlighted. Only the relevant extensions are shown.
When using Prism-MW, the developer first subclasses the `Component` class for all components in the architecture and implements their application-specific methods. The next step is to instantiate the `Architecture` classes for each address space and define the needed instances of thus created components, and of connectors selected from the reusable connector library. Finally, attaching component and connector instances into a configuration is achieved by using the `weld` method of the `Architecture` class. This process can be partially automated using our tool support described in [16].

To support capabilities that may be required for different (classes of) Prism applications, Prism-MW provides three specialized classes: `ExtensibleComponent`, `ExtensibleConnector`, and `ExtensibleEvent`. These classes extend the corresponding base classes and, by composing a number of interfaces, provide the ability to select the desired functionality inside each instance of an `Extensible` class. If an interface is installed in a given `Extensible` class instance, that instance will exhibit the behavior realized inside the interface’s implementation. Multiple interfaces may be installed in a single `Extensible` class instance. In that case, the instance will exhibit the combined behavior of the installed interfaces. To date, we have provided support for architectural awareness, real-time computation, distribution, security, data compression, delivery guarantees, and mobility. The details of these extensions may be found in [16].

In support of distribution, Prism-MW provides the `ExtensibleConnector`, which composes the `IDistribution` interface. To date, we have provided two implementations of the `IDistribution` interface, supporting socket-based and infrared-port based interprocess communication. An `ExtensibleConnector` with the instantiated `IDistribution` interface (referred to as `DistributionConnector`) facilitates interaction across process or machine boundaries. In addition to the `IDistribution` interface inside the `ExtensibleConnector` class, to support distribution and mobility we have implemented the `Serializable` interface inside each one of the `Extensible` classes. This allows us to send data as well as code across machine boundaries.

To support various aspects of architectural awareness, we have provided the `ExtensibleComponent` class, which contains a reference to `IArchitecture`. This allows an instance of `ExtensibleComponent` to access all architectural elements in its local configuration, acting as a meta-level component that effects run-time changes on the system’s architecture.

### 4.2 Disconnected Operation Support

To date, we have augmented `ExtensibleComponent` with several interfaces. Of interest in this paper is the `IAdmin` interface used in support of redeployment. We provide two implementations of the `IAdmin` interface: `Admin`, which supports system monitoring and redeployment effecting, and `Admin`’s subclass `Deployer`, which also provides facilities for redeployment estimation. We refer to the `ExtensibleComponent` with the `Admin` implementation of the `IAdmin` interface as `AdminComponent`; analogously, we refer to the `ExtensibleComponent` with the `Deployer` implementation of the `IAdmin` interface as `DeployerComponent`.

As indicated in Figure 4, both `AdminComponent` and `DeployerComponent` contain a pointer to the `Architecture` object and are thus able to effect run-time changes to their local subsystem’s architecture: instantiation, addition, removal, connection, and
disconnection of components and connectors. With the help of DistributionConnectors, AdminComponent and DeployerComponent are able to send and receive from any device to which they are connected the events that contain application-level components (sent between address spaces using the Serializable interface).

In order to perform run-time redeployment of the desired architecture on a set of target hosts, we assume that a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW’s Architecture object that contains a DistributionConnector and an AdminComponent that is attached to the connector. One of the hosts contains the DeployerComponent (instead of the AdminComponent) and controls the redeployment process. The DeployerComponent gathers all the monitoring data from different AdminComponents and estimates the system redeployment. Then, the DeployerComponent collaborates with AdminComponents to effect the estimated redeployment architecture. The details of this process are described in Sections 5, 6, and 7.

Our current implementation assumes that the host containing the DeployerComponent will have a direct (possibly unreliable) connection with all the remaining hosts. An alternative design would require only the existence of a path between any two hosts (i.e., a connected graph of hosts). In this scenario, the information that needs to be exchanged between a pair of hosts not connected directly would need to get routed through a set of intermediary hosts (e.g., by using event forwarding mechanisms of Siena [5]). We are currently implementing and evaluating this design.

5 System Monitoring

5.1 Monitoring Requirements

System monitoring [10] is a process of gathering data of interest from the running application. In the context of system redeployment, the following data needs to be obtained: (1) frequency of interaction among software components; (2) each components’ maximum memory requirements; (3) reliability of connectivity among hardware hosts; and (4) available memory on each host. Since we assume that the available memory on each host and maximum required memory for each software component are stable throughout the system’s execution, these parameters can be obtained either from the system’s specification (e.g., [2]) or at the time the initial deployment of the system is performed. Therefore, the active monitoring support should gather the following parameters: (1) for each pair of software components in the system, the number of times these components interact is recorded, and (2) for each pair of hardware hosts, the ratio of the number of successfully completed remote interactions to the total number of attempted interactions is recorded. Furthermore, due to the limited time available to perform a system’s redeployment, the time required to complete system monitoring should be minimized (recall Figure 3).

5.2 Prism-MW’s Support for Monitoring

In support of monitoring Prism-MW provides the IMonitor interface associated through the Scaffold class with every Brick. This allows us to monitor the run-time behavior of each Brick. To date, we have provided two implementations of the IMoni-
tor interface: $EvtFrequencyMonitor$ records the frequencies of different events the associated $Brick$ sends, while $DisconnectionRateMonitor$ records the reliability of connectivity between its associated $DistributionConnector$ and other remote $DistributionConnectors$.

An $AdminComponent$ on any device is capable of accessing the monitoring data of its local components and connectors (recorded in their associated implementations of the $IMonitor$ interface) via its pointer to the $Architecture$. The $AdminComponent$ then sends that data in the form of serialized $ExtensibleEvents$ to the requesting $DeployerComponent$.

In order to minimize the time required to monitor the system, system monitoring is performed in short intervals. The $AdminComponent$ compares the results from consecutive intervals. As soon as the difference in the monitoring data between two consecutive intervals becomes small (i.e., less than a given value $\epsilon$), the $AdminComponent$ assumes that the monitoring data is stable, and informs the $DeployerComponent$.

Note that the $DeployerComponent$ will get the reliability data twice (once from each host). On the one hand, this presents additional overhead. On the other hand, this feature can be used to more accurately assess the reliability data, by taking the average from the two sets of monitoring data. Furthermore, this overhead presents only a fraction of the total monitoring overhead, since the number of hosts is usually much smaller than the number of components. The frequency data will be received by the $DeployerComponent$ only once, since each $EvtFrequencyMonitor$ only monitors the outgoing events of its associated component.

An issue we have considered deals with cases when most, but not all system parameters are stable. As described above, if any of the parameters do not satisfy their $\epsilon$ constraint, the redeployment estimation will not be initiated. There are at least two ways of addressing this situation. First is to increase the $\epsilon$ for the specific troublesome parameters and thus induce the redeployment estimation. Alternatively, a single, global $\epsilon_g$ may be used to initiate redeployment estimation as soon as the average difference of the monitoring data for all the parameters in the system becomes smaller than $\epsilon_g$. We support both these options and are currently assessing their respective strengths and weaknesses.

Our initial assessment of Prism-MW’s monitoring support suggests that continuous monitoring on each host will likely induce no more than 10% computational overhead and 5% memory overhead. We are currently studying the actual monitoring overhead caused by our solution.

6 Estimating Redeployment

6.1 Algorithms and Their Analysis

In this section we describe our two algorithms for estimating a system’s redeployment architecture. These algorithms require the data obtained during the monitoring stage. We analyze the algorithms’ performance, both in terms of time complexity (i.e., $T_\epsilon$) and of the achieved availability.
6.1.1 Exact Algorithm

This algorithm tries every possible deployment architecture, and selects the one that has maximum availability and satisfies the constraints posed by the memory and restrictions on locations of software components. The exact algorithm guarantees at least one optimal deployment. The complexity of this algorithm in the general case (i.e., with no restrictions on component locations) is \( O(k^n) \), where \( k \) is the number of hardware hosts, and \( n \) the number of software components. By fixing a subset of \( m \) components to selected hosts, the complexity of the exact algorithm reduces to \( O(k^{n-m}) \). Even with this reduction, however, this algorithm may be computationally too expensive if the number of hardware nodes and unfixed software components involved is not very small.

Figure 5 shows the performance of this algorithm. For example, if there are no restrictions on locations of software components, even for a relatively small deployment architecture (15 components, 4 hosts), the exact algorithm runs for more than eight hours.

This algorithm may be used for calculating optimal deployment for systems whose characteristics (i.e., input parameters to the algorithm) are stable for a very long time period. In such cases, it may be beneficial to invest the time required for the exact algorithm, in order to gain maximum possible availability. However, note that even in such cases, running the algorithm may become infeasible very quickly.

6.1.2 Approximative Algorithm

Given the complexity of the exact algorithm and limited time available for estimating the system’s redeployment, we had to devise an approximative algorithm that would significantly reduce this complexity while exhibiting good performance.

The redeployment problem is an instance of a large class of global optimization problems, in which the goal is to find a solution that corresponds to the minimum (or maximum) of a suitable criterion function, while it satisfies a given collection of feasibility constraints. Most global optimization problems are NP-hard [7].

Different techniques have been devised to search for approximative solutions to global optimization problems. Some of the more commonly used strategies are dynamic programming, branch-and-bound, and greedy algorithms [6]. When devising approximative solutions for global optimization problems, the challenge is to avoid getting stuck at the “local optima”. There are several techniques that can be applied to avoid this problem: genetic algorithms, simulated annealing, and stochastic (i.e., random) algorithms [6,21]. It has been demonstrated that stochastic algorithms produce good results more quickly than the other two techniques [1]. In this section, therefore,

---

1 An NP-hard problem cannot be solved, or an existing solution to it verified, in polynomial time.
we describe and assess the performance of a stochastic approximative algorithm with polynomial time complexity ($O(n^2)$).

This algorithm randomly orders all the hosts and randomly orders all the components. Then, going in order, it assigns as many components to a given host as can fit on that host, ensuring that the assignment of each component is allowed (recall the loc restriction in Figure 2). Once the host is full, the algorithm proceeds with the same process for the next host in the ordered list of hosts, and the remaining unassigned components in the ordered list of components, until all components have been deployed. This process is repeated a desired number of times, and the best obtained deployment is selected. The complexity of this algorithm is polynomial, since we need to calculate the availability for every deployment, and that takes $O(n^2)$ time.

In order to assess the performance of our two algorithms, we have implemented a tool, called DeSi [16], that provides random generation of the system parameters, the ability to modify these parameters manually, and the ability to both textually and graphically display the results of the two algorithms.

We have assessed the performance of the approximative algorithm by comparing it against the exact algorithm, for systems with small numbers of components and hosts (i.e., less than 13 components, and less than 5 hosts). In large numbers of randomly generated problems, the approximative algorithm invariably found a solution that was at least 80% of the optimal with 1000 iterations. Figure 6 shows the results of these benchmarks.

For larger problems, where the exact algorithm is infeasible, we have compared the results of the approximative algorithm for varying number of iterations to the minimum availability obtained.\(^2\) The results of this benchmark are illustrated in Figure 7. The achieved availability in these

\(^2\) The minimum availability is obtained by recording the worst availability during the approximative algorithm’s search.
four architectures was at least 70%. Furthermore, with an order of magnitude increase in the number of iterations, the output of the algorithm improved at most by 2%. Finally, the achieved availability was at least 60% greater than the minimum.

6.2 Prism-MW’s Support for Estimation

The DeployerComponent accesses its local monitoring data; it also receives all the monitoring data from the remote AdminComponents. Once the monitoring data is gathered from all the hosts, the DeployerComponent initiates the redeployment estimation, by invoking the execute operation of the installed IDeployerAlgorithm interface. To date, we have provided two implementations of the IDeployerAlgorithm interface, ExactAlgorithm and ApproximativeAlgorithm. The output of each of these algorithms is a desired deployment architecture (in the form of unique component-host identifier pairs), which now needs to be effected.

7 Effecting Redeployment

7.1 Requirements for Effecting Redeployment

Effecting the system’s redeployment is performed by determining the difference between the current deployment architecture and the estimated one. This will result in a set of components to be migrated. Thus obtained components are then migrated from their source hosts to the destination hosts. After the migration is performed, the migrant components need to be attached to the appropriate locations in their destination configurations. In order to effectively manage system redeployment, the exchange of components between hosts that are not directly connected should be supported. Furthermore, a solution for effecting redeployment would also need to address situations in which network bandwidth and reliability of connectivity between a pair of hosts restrict the maximum size of components that may be exchanged. Otherwise, it may not be possible to effect the redeployment architecture obtained during the estimation phase.

7.2 Prism-MW’s Support for Redeployment Effecting

The DeployerComponent controls the process of effecting the redeployment as follows:

1. The DeployerComponent sends events to inform AdminComponents of their new local configurations, and of remote locations of software components required for performing changes to each AdminComponent’s local configuration.
2. Each AdminComponent determines the difference between its current configuration and the new configuration, and issues a series of events to remote AdminComponents requesting the set of components that are to be deployed locally. If some of the devices containing the desired components are not directly reachable from the requesting device, the request events for those components are sent by the local AdminComponents to the DeployerComponent. The DeployerComponent then forwards those
events to the appropriate destinations, and forwards the responses containing migrant components to the requesting AdminComponents. Therefore, the DeployerComponent serves as a router for devices not directly connected (recall the discussion in Section 4.2).

3. Each AdminComponent that receives an event requesting its local component(s) to be deployed remotely, detaches the required component(s) from its local configuration, serializes them (therefore preserving the component’s state during the migration), and sends them as a series of events via its local DistributionConnector to the requesting device.

4. The recipient AdminComponents reconstitute the migrant components from the received events.

5. Each AdminComponent invokes the appropriate methods on its Architecture object to attach the received components to the local configuration.

As discussed in Section 4.2, our current implementation assumes a centralized organization, i.e., that the device containing the DeployerComponent will have direct connection with all the remaining devices. We are currently implementing and evaluating an existing decentralized solution to a similar problem [5].

To address the situations where the reliability of connectivity, network bandwidth, and component size would prevent the exchange of a migrant component between a pair of hosts from occurring atomically (i.e., using a single event to send the migrant component), Prism-MW supports incremental component migration, as follows:

1. The sending AdminComponent serializes the migrant component into a byte stream.

2. The sending AdminComponent divides the byte stream into small segments, whose size is programmatically adjustable.

3. Each segment is packaged into a separate event, numbered, and sent atomically.

4. After the last chunk is sent, the sending AdminComponent sends a special event denoting that the entire component has been sent.

5. The receiving AdminComponent reconstitutes the migrant component from the received byte code chunks, requesting that the sending AdminComponent resend any missing (numbered) segments.

8 Conclusions and Future Work

As the distribution, decentralization, and mobility of computing environments grow, so does the probability that (parts of) those environments will need to operate in the face of network disconnections. On the one hand, a number of promising solutions to the disconnected operation problem have already emerged. On the other hand, these solutions have focused on specific system aspects (e.g., data caching, hoarding, and replication; special purpose, disconnection-aware code) and operational scenarios (e.g., anticipated disconnection [24]). While each of these solutions may play a role in the emerging world of highly distributed, mobile, resource constrained environments, our research is guided by the observation that, in these environments, a key determinant of the system’s ability to effectively deal with network disconnections is its deployment architecture.
This paper has thus presented a set of algorithms, techniques, and tools for improving a distributed, mobile system’s availability via redeployment. Our support for disconnected operation has been successfully tested on several example applications. We are currently developing a simulation framework, hosted on Prism-MW, that will enable the assessment of our approach on a large number of simulated hardware hosts, with varying but controllable connectivity among them.

While our experience thus far has been very positive, a number of pertinent questions remain unexplored. Our future work will span issues such as (1) devising new approximative algorithms targeted at different types of problems (e.g., different hardware configurations such as star, ring, and grid), (2) support for decentralized redeployment (in cases where the DeployerComponent is not connected to all the remaining hardware hosts), (3) addressing the issue of trust in performing distributed redeployment, and (4) studying the trade-offs between the cost of redeployment and the resulting improvement in system availability. Finally, we intend to expand our solutions to include system parameters other than memory, frequency of interaction, and reliability of connection (e.g., battery power, display size, system software available on a given host, and so on).

Acknowledgements

The authors wish to thank Sam Malek for his contributions to the Prism project. This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9985441 and ITR-0312780.

References

Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition*

Hyun Gi Min, Si Won Choi, and Soo Dong Kim

Department of Computer Science, Soongsil University
1-1 Sangdo-Dong, Dongjak-Ku, Seoul, Korea 156-743
{hgmin,swchoi}@otlab.ssu.ac.kr, sdkim@ssu.ac.kr

Abstract. Components, especially commercial-off-the-shelf (COTS) components, are mainly for inter-organizational reuse. One of the essential tasks in component-based development (CBD) is to locate and reuse the right components that provide the functionality and interface required by component consumers. However, if a candidate component provides a limited applicability and customizability so that it does not completely satisfy the functionality and interface needed, then a component consumer cannot reuse the component in application development. We call it a partial matching problem in component acquisition. To resolve this problem, we propose smart connectors that fill the gap between candidate components and the specification of components required. By using connectors, partially matched components become reusable in application development without sacrificing the component consumer’s requirements. Consequently, the effort and cost to develop new components and applications can be greatly reduced. In this paper, we propose four types of connectors, and each connector type is specified with its applicable situation and instructions to design correct connectors.

1 Motivation

CBD has acquired a substantial acceptance in both academia and industry as an effective paradigm for building software systems with reusable assets [1]. Components, especially commercial-off-the-shelf (COTS) components, are mainly for inter-organizational reuse. One of the essential tasks in component-based development (CBD) is to locate and reuse the right components that provide the functionality and interface required by component consumers, called component acquisition. Once appropriate components are identified, they are customized for the target application through available customization mechanisms, called component adaptation [2].

However, if a candidate component provides a limited applicability and customizability so that it does not completely satisfy the functionality and interface needed, then a component consumer cannot reuse the component in application development [3]. We call it a partial matching problem in component acquisition. To resolve this prob-

* This work was supported by the Soongsil University Research Fund.

© Springer-Verlag Berlin Heidelberg 2004
lem, we propose smart connectors that fill the gap between candidate components and the specification of components required. Connectors in software architecture are mainly used to connect and enable communications among architectural elements, and so they rarely provide functionality of data manipulation, transformation, and mediation. [4]. However, smart connectors proposed in this paper are more intelligent and active since they provide richer and complex functionality of resolving the unmatched portion between candidate components and client program. Consequently, partially matched components become reusable in application development, and the effort and cost to develop new components and applications can be greatly reduced.

In this paper, we propose four types of connectors; value range transformer, interface adapter, functional transformer, and workflow handler. Each connector type is specified with its applicable situation of partial matching and mechanism for filling the gap. Then, a systematic process to design connectors is proposed, and a case study of applying connectors to the banking domain is given.

2 Related Works

Spitznagel addresses the partial matching problem between software modules in general, and proposes connector mechanisms to resolve the problem [5]. Interface incompatibility is resolved by aggregate mechanisms, behavioral incompatibility is resolved by add_a_role mechanisms, and data type incompatibility is resolved by data transform mechanisms. However, this work identifies different types of mismatches between modules and sketches the resolution mechanisms by giving precise instructions to design connectors. Moreover, issues in the partial matching problem for component acquisition, such as variability and workflow, are not addressed.

Mehta defines service categories as Communication, Coordination, Conversion and Facilitation [6]. In addition, the author suggests eight kinds of connector type: Procedure Call, Event, Data Access, Linkage, Stream, Arbitrator, Adaptor and Distributor. Data incompatibility is resolved by Communication, Conversion, Event, Data Access and Stream. Interface incompatibility is resolved by Facilitation, Arbitrator, Adaptor and Distributor, and control incompatibility is resolved by Coordination, Conversion, Procedure Call, Event and Arbitrator. However, the connector type of this work is not intended to solve the mismatch between components. These connector types classify communication between components. If the connector type is classified as the mismatch category, they are duplicated in the incompatibility classification. Moreover, this work does not support partial behavior mismatch problems.

Spitznagel suggests a wrapper mechanism for architecture engineering [7]. The No Effect wrapper relays events from the caller role to the glue role and vice versa. The Retry wrapper intercepts any timeout error sent to the client and sends out a new call event to the glue. The Failover wrapper masks the potential failure. Hence, the wrappers are mainly used to transform the communication protocols, and transforming various partial matching circumstances occurring in CBD, such as data mismatch, interface mismatch, and behavioral mismatch, are not addressed.
3 Four Types of Smart Connectors

In this section, each of the four connector types is specified with its applicable situation and mechanism. The term COMP describes a candidate COTS component under evaluation for suitability, and the term client means either an application program or another component that expects to invoke services provided by the COMP. Smart connectors resolve mismatches between COMP and client without sacrificing the component specification needed by client. Hence, both COMP and client specifications are completely intact, resulting in lower coupling between them.

3.1 Value Range Transformer

As the simplest form of connector, the Value Range Transformer is used to transform the value range of a data element provided by COMP to the value range required by client, and vice versa.

**Situation:** A client and a COMP typically exchange information through interfaces or a shared data store such as external file. If the signatures of the interface provided by COMP match those expected by client but the value ranges provided and required do not match, then there is a mismatch problem on value ranges of data elements. A data element can be an input, output or return parameter of an interface, or a data store shared by two parties. For example, in figure 1 a temperature monitor component returns the current temperature in Fahrenheit at a particular time by `getTemperature (t: Time): float`, and the client program uses the same function signature but requires the temperature in Centigrade.

**Mechanism:** First, the incompatible value ranges of data elements are examined and transformation rules including the conversion formula are defined. In figure 1, the rule has a temperature conversion formula. Then, the transformer takes a value produced by COMP, converts it to new value according to the transformation rule, and then returns the new value to client, and vice versa. In this example, a temperature in Centigrade is returned to client. As the result, an originally incompatible COMP becomes usable for client.
3.2 Interface Adapter

*Interface Adapter* is used to resolve mismatches between a signature required by *client* and the signature provided by *COMP*.

**Situation:** *COMP* has a provided interface which consists of function signatures and their semantic descriptions. If a function satisfies the behavior required by *client* but its signature does not match the signature required by *client*, then there is a mismatch problem on the interface. A mismatch may occur on function name, types of input/output parameters, the ordering of parameters and return type. For example, a customer management component provides a function of creating record for new customer with this signature:

\[
\text{RegisterCustomer}(\text{LastName}: \text{String}, \text{FirstName}: \text{String}, \text{ID}: \text{String}, \text{Address}: \text{String}): \text{integer};
\]

This function satisfies the functionality needed by *client* which is a legacy application, however *client* has to invoke the function with this particular signature.

\[
\text{AddNewCustomer}(\text{Name}: \text{NameType}, \text{Address}: \text{String}, \text{SSN}: \text{SSNType}): \text{boolean};
\]

Then, we have mismatches on function name, parameter types, ordering of parameters and return type. *RegisterCustomer* uses two parameters for the name, but *AddNewCustomer* uses one parameter *Name* which is a structure type containing both last name and first name.

![Transformation Rule]

**Mechanism:** *Interface Adapter* works in six steps as in figure 2. A *client* sends a message, the adapter constructs a new message by converting relevant elements of the message according to the transformation rule, it sends the new message to *COMP*, the *COMP* generates a return value after executing the method, the adapter possibly converts the return value, and finally returns it to *client*. The transformation rule in the figure 2 consists of all mappings between elements of the original message and the new message. The adapter implements these mappings by decomposing compound
data type into finer-grained data types, type conversion, value conversion, re-ordering parameters, and/or renaming.

There is a special case that is slightly different from the process stated above. If the signature of a function in COMP contains more parameters than those provided by a message sent by client, then the adapter should acquire values for lacking parameters by relevant data manipulation or input from the user.

3.3 Functional Transformer

When COMP satisfies most of the functionality that client expects but partially lacks some functionality, Functional Transformer can be used to resolve the partial mismatch between the functionality required by client and that provided by COMP.

Situation: There are three cases when comparing the provided functionality and the required functionality, as shown in figure 3. Functionality is determined by the set of all functions in a component. Let a predicate Fn(COMP) be the functionality provided by COMP and Fn(client) accordingly. In case i), Fn(COMP) and Fn(client) fully match and hence COMP can be used by client. In case ii), the extra functionality of COMP should be disabled for client by a connector. In case iii), COMP should be appended with some additional functionality by a connector.

![Fig. 3. Three Cases for Matching Functionalities.](image)

There is another dimension on functional matches. For a function which belongs to the area of intersection between Fn(COMP) and Fn(client), its internal logic (or algorithm) may not fully match that required by client. This mismatch is about the internal logic of a function, rather than comparing whole functionalities. We call it logic mismatch.

Mechanism: Functional Transformer works differently for three cases.
Case i) No connector is needed unless there is a logic mismatch on individual functions.
Case ii) Extra functionality of COMP should be disabled by a connector. This can be done by ignoring invocations on the excessive functions or by raising exceptions. If additional functionality is not disabled and is used by client accidentally, then unexpected side effects may be generated.
Case iii) Any functionality unsupported by COMP should be realized and provided by a connector. As illustrated in figure 4, this can be done by implementing new functions that directly manipulate data elements of COMP or indirectly manipulate
data elements through get( ), set( ) and public methods of COMP. Providing additional functionality through a connector is not always possible. That is, if COMP does not expose data elements, e.g. they are declared with private, or if COMP does not provide the necessary get( ) and set( ) methods, the connector cannot manipulate the necessary data and provide the additional functionality.

For the logic mismatch as explained above, a functional transformer can be equipped with new logic which fully or partially replaces the original logic provided by COMP. However, implementing newer logic in connector only becomes possible when COMP provides necessary public data elements and/or methods.

3.4 Workflow Handler

Workflow is a sequence of method invocations among components to carry out a function in an interface. Workflow Handler is used to resolve mismatch between workflow required by client and workflow provided by COMP. Workflow mismatch is distinguished from logic mismatch defined in section 3.3; workflow mismatch is determined by examining the orders of multiple method invocations where logic mismatch is about the behavior of a single method.

Situation: A workflow nature function of COMP is implemented by invoking methods of objects in COMP. A function realized with the mediator pattern is mostly workflow intensive. If the workflow taken by COMP does not match that required by client, a connector is used to implement the required workflow.

Mechanism: Workflow Handler implements the new workflow by invoking public methods of COMPs. Figure 5 illustrates the five steps the connector takes. It receives a message foo( ) sent by client and constructs possibly multiple messages with the parameters of foo( ). COMPs receive messages and execute corresponding methods. Then, the handler constructs a single return value from return values received and finally client receives the return value.
Note that it is possible to implement a new workflow in a connector only if COMPs provide all necessary functions in their interface. Also, side effects such as state changes and database updates caused by invoking new workflow must be carefully examined to maintain the integrity of COMPs.

4 Process and Assessment

Process: The process to implement smart connectors is given in figure 6. After candidate components are collected by examining component specifications, all potential mismatches between COMPs and client are identified. Using the situations and mechanisms specified in this paper, smart connectors are identified, designed and realized. Finally, components and connectors are integrated to satisfy the precise requirement set by client.

Assessment: Smart connectors proposed in this paper are compared to other representative works in table 1. The comparison criteria are selected from the view point of component acquisition, rather than connectors for gluing architectural elements. Smart connectors are shown to effectively support all the essential criteria for resolving mismatch problems in CBD.
Table 1. Comparison of Representative Works on Connectors,  
(● for Supports, ○ for Partially Supports)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Glueing Components</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Transforming Value Ranges of Data</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Adapting Component Interfaces3</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Modifying Functional Behavior</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Adding new Functionality</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Modifying Workflows</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Combining Different Connectors</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

5 Concluding Remarks

One of the essential tasks in CBD is to locate and reuse the right components that provide the functionality and interface required by component consumers. If a candidate component provides a limited applicability and customizability, it cannot be reused in application development, resulting in a partial matching problem. To resolve this problem, we proposed four types of smart connectors, value range transformer, interface adapter, functional transformer and workflow handler, which fill the gap between candidate components and the specification of components required. By using connectors, partially matched components become reusable in application development without sacrificing component consumer’s requirement. Consequently, the effort and cost to develop new components and applications can be greatly reduced.

References

Correctness of Component-Based Adaptation*

Sandeep S. Kulkarni and Karun N. Biyani

Software Engineering and Network Systems (SENS) Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824 USA
{sandeep,biyanika}@cse.msu.edu
Tel: +1-517-355-2387

Abstract. Long running applications often need to adapt due to changing requirements or changing environment. Typically, such adaptation is performed by dynamically adding or removing components. In these types of adaptation, components are often added to or removed from multiple processes in the system. While techniques for such adaptations have been extensively discussed in the literature, there is a lack of systematic methods to ensure the correctness of dynamic adaptation. To redress this deficiency, in this paper, we propose a new method, based on the concept of proof lattice, for verifying correctness of dynamic adaptation in a distributed application. We use transitional-invariant lattice to verify correctness of adaptation. As an illustration of this method, we show how correctness of dynamic adaptation is obtained in the context of a message communication application.

Keywords: Dynamic Adaptation, Correctness, Verification

1 Introduction

Long running applications are often subjected to adaptations due to changing requirements and/or execution environment. Adaptive software provides techniques [1, 2, 3, 4] that allow the software to modify its own functional behavior or non-functional behavior (e.g., its fault-tolerance or security requirements). These modifications may include reconfiguration of some parameters, addition or removal of filters and, in the extreme, addition or removal of arbitrary code from the running application.

The approaches in [1, 2, 3, 4] address several syntactic issues in adaptation, e.g., parameter checking, dynamic loading, reflection, etc. However, these approaches do not focus on the semantic issues related to the application during or after adaptation. Specifically, these approaches do not address the correctness of the application during and/or after adaptation. Thus, there is a need to develop

* This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

© Springer-Verlag Berlin Heidelberg 2004
techniques that can enable us to reason about the correctness of the application that is being adapted using techniques such as those in [1, 2, 3, 4].

One of the problems in arguing about correctness of adaptation is that adaptation is often performed due to changing requirements where the specification of the application is also changing. Hence, existing techniques for verifying correctness of programs cannot be applied directly as these techniques assume that the program and the specification are fixed. In addition, it is also necessary to consider the specification during adaptation as it may be different from the specification before and after adaptation.

With the above motivation, in this paper, we focus on the methods for verification of programs during and after adaptation. Our approach is based on the idea of proof lattice [5]. Since we focus on the verification during and after adaptation, we assume that the original program (before adaptation) begins in a state from where its specification would be satisfied. Our approach ensures that after adaptation is complete, the program would be in a state from where its new specification (after adaptation) is satisfied. In addition, it ensures that the specification during adaptation is also satisfied.

**Organization of the Paper.** The rest of the paper is organized as follows: In Sect. 2, we introduce some formal definitions to model an adaptive system, adaptation, and define specification during adaptation. In Sect. 3, we introduce the notion of transitional-invariant lattice to verify adaptation and illustrate its use in Sect. 4 by verifying correctness of adding the proactive component to the message communication application. Finally, we conclude in Sect. 5.

## 2 Modeling Adaptation

In this section, we describe how we model the adaptive programs in order to show their correctness. We consider adaptations where the program needs to add, remove or replace distributed components. A distributed component consists of one or more *component fractions* [6]. Each fraction is associated with one process of the program. To add such a component to a distributed program, each fraction of the component needs to be added at processes of the program. Similarly, to remove a distributed component, each fraction of the component needs to be removed from the corresponding process of the program. In other words, adaptation in distributed programs involves multiple steps. We divide this multi-step adaptation into multiple *atomic adaptations* each involving only one process. Each atomic adaptation is represented by a name and has a *guard* (defined later in this section). An atomic adaptation is performed when the guard corresponding to it becomes true.

Now, we discuss how we model such distributed programs and the multi-step adaptation performed in them. We note that general purpose languages such as C++/Java that are used in existing adaptation techniques (e.g., [1, 6, 2, 3, 4]) are not suitable for our task as verification of programs in these languages is difficult even in the absence of adaptation. For this reason, while proving correctness of programs, we often consider their abstract version where several implementation details are omitted.
2.1 Modeling Adaptive Program

Program and Process. A program $P$ is specified by a set of global constants, a set of global variables and a finite set of processes. A process $p$ is specified by a set of local constants, a set of local variables and a finite set of actions (defined later in this section). Variables declared in $P$ can be read and written by the actions of all processes. The processes in a program communicate with one another by sending and receiving messages over unbounded channels that connect the processes. We use the notation $ch.p.q$ to denote the content of the channel from process $p$ to process $q$, and $\#ch.p.q$ to denote the number of messages in the channel from $p$ to $q$.

State and State Predicate. A state of process $p$ is defined by a value for each variable of $p$, chosen from the predefined domain of the variable. A state of program $P$ is defined by a value for each global variable of $P$, the state of all its processes and the contents of all channels. A state predicate of $p$ is a boolean expression over the constants and variables of $p$. A state predicate of $P$ is a boolean expression over constants and variables of all processes, global constants and global variables, and the contents of all channels.

Action. An action of $p$ is uniquely identified by a name, and is of the form

$$\langle \text{name} \rangle : \langle \text{guard} \rangle \to \langle \text{statement} \rangle$$

A guard is a combination of one or more of the following: a state predicate of $p$, a receiving guard of $p$, or a timeout guard. A receiving guard of $p$ is of the form $rcv \langle \text{message} \rangle \text{ from } \langle q \rangle$. A timeout guard is of the form $\text{timeout} \langle \text{state predicate of } P \rangle$. The statement of an action updates zero or more variables and/or sends one or more messages. An action can be executed only if its guard evaluates to true. To execute an action, the statement of that action is executed atomically. A sending statement of $p$ is of the form $\text{send} \langle \text{message} \rangle \text{ to } \langle q_1, \ldots, q_n \rangle$, which sends a message to one or more processes. We say that an action of $p$ is enabled in a state of $P$ iff its guard evaluates to true in that state.

Computation. A computation of $P$ is a sequence of states $s_0, s_1, \ldots$ such that for each $j, j > 0, s_j$ is obtained from state $s_{j-1}$ by executing an action of $P$ that is enabled in the state $s_{j-1}$, and satisfies the following conditions: (i) Non-determinism: Any action that is enabled in a state of $P$ can be selected for execution at that state, (ii) Atomicity: Enabled actions in $P$ are executed one at a time, (iii) Fairness: If an action is continuously enabled along the states in the sequence, then that action is eventually chosen for execution, and (iv) Maximaliity: Maximaliity of the sequence means that if the sequence is finite then the guard of each action in $P$ is false in the final state.

Closure. Let $S$ be a state predicate. An action $ac$ of $p$ preserves $S$ iff in any state where $S$ is true and $ac$ is enabled, atomically executing the statement of $ac$ yields a state where $S$ continues to be true. $S$ is closed in a set of actions iff each action in that set preserves $S$.

Specification. The program specification is a set of acceptable computations. Following Alpern and Schneider [7], specification can be decomposed into a safety
specification and a liveness specification. Also, as shown in [8], for a rich class of specifications, safety specification can be represented as a set of bad transitions that should not occur in program computations. Hence, we represent this safety specification by a set of bad transitions that should not occur in program computations. We omit the representation of liveness specification here as it is not required for our purpose.

**Satisfies.** $\mathcal{P}$ satisfies specification from $S$ iff each computation of $\mathcal{P}$ that starts from a state where $S$ is true is in the specification.

**Invariant.** The state predicate $S$ of $\mathcal{P}$ is an invariant iff (i) $S$ is closed in $\mathcal{P}$, and (ii) $\mathcal{P}$ satisfies specification from $S$.

**Specification during Adaptation.** As mentioned in the introduction, we need to consider the specification of the program before, during and after adaptation. We argue that while the specification before and after adaptation can be arbitrary, the specification during adaptation should be a safety specification. This is due to the fact that one often wants the adaptation to be completed as quickly as possible. Hence, it is desirable not to delay the adaptation task to satisfy the liveness specification during adaptation. Rather, it is desirable to guarantee that, after adaptation, the program reaches states from where its (new) safety and liveness specification is satisfied. Thus, the implicit liveness specification during adaptation is that the adaptation completes. While our method does not make any additional assumptions about the specification during adaptations, we expect it to be weaker than the specification before (respectively after) adaptation. For example, it may be acceptable if requirements such as bounded fairness are not satisfied during adaptation.

**Notation.** We use the notation $X \vee Y \wedge Z$ to imply that only one of $X$, $Y$, or $Z$ is true at a time.

## 3 Verifying Adaptation

In this section, we introduce the notion of transitional-invariant lattice to verify the correctness of adaptation. The idea of transitional-invariant lattice is based on the concept of proof lattice [5], which is used to prove liveness properties of a concurrent program. We first define adaptation lattice and then introduce transitional-invariants to define transitional-invariant lattice.

**Adaptation Lattice.** An adaptation lattice is a finite directed acyclic graph in which each node is labeled with one or more predicates and each edge is labeled with an atomic adaptation, such that,

1. There is a single entry node $P$ having no incoming edges. The entry node is associated with predicates over the variables of the old program, i.e., the program before adaptation.
2. There is a single exit node $Q$ having no outgoing edges. The exit node is associated with predicates over the variables of the new program, i.e., the program after adaptation.
3. Each intermediate node $R$ that has at least one incoming and outgoing edge is associated with predicates over the variables of both the old and the new program.

In the context of adaptation, the program adds and removes components, and thus, the program during adaptation consists of actions of the old program and the new program. Therefore, we consider intermediate programs obtained after one or more atomic adaptations. Now, similar to the invariants that are used to identify “legal” program states and are closed under program execution, we define transitional-invariants.

**Transitional-Invariant.** A transitional-invariant is a predicate that is true throughout the execution of an intermediate program and is closed under the old program actions that are not yet removed and the new program actions that are already added. Note however that the atomic adaptations do not necessarily preserve the transitional-invariant.

**Transitional-Invariant Lattice.** A transitional-invariant lattice is an adaptation lattice with each node having one predicate and that satisfies the following five conditions (see Fig. 1 for an example):

1. The entry node $P$ is associated with an invariant $S_P$ of the program before adaptation.
2. The exit node $Q$ is associated with an invariant $S_Q$ of the program after adaptation.
3. Each intermediate node $R$ is associated with a transitional-invariant $TS_R$, such that any intermediate program at $R$ (i.e., intermediate program obtained by performing adaptations from the entry node to $R$) satisfies the (safety) specification during adaptation from $TS_R$.
4. If a node labeled $R_i$ has an outgoing edge labeled $A$ to a node labeled $R_j$, then performing atomic adaptation $A$ in any state where $TS_{R_i}$ holds and guard of $A$ is true results in a state where $TS_{R_j}$ holds, and the transition obtained by $A$ satisfies the safety specification during adaptation.
5. If a node labeled $R$ has outgoing edges labeled $a_1, a_2, ..., a_k$ to nodes labeled $R_1, R_2, ..., R_k$, respectively, then in any execution of any intermediate program at $R$ that starts in a state where $TS_R$ is true, eventually, the guard of at least one atomic adaptation $a_i$, $1 \leq i \leq k$, becomes true and remains true thereafter and, hence, eventually some $a_i$ will be performed.

![Fig. 1. An example of a transitional-invariant lattice.](image-url)
Remark. An intermediate node $R$ could be reached by multiple paths. Therefore, it is required that $TS_R$ be met for each intermediate program corresponding to the path.

Now, to prove the correctness of adaptation, we need to find a transitional-invariant lattice corresponding to the adaptation. This is stated formally by the following theorem: (Due to limited space, we omit the proofs of this and other theorems in the paper. We refer readers to [9] for the proofs.)

**Theorem 1.** Given $S_P$ as the invariant of the program before adaptation and $S_Q$ as the invariant of the program after adaptation, if there is a transitional-invariant lattice for an adaptation with entry node associated with $S_P$ and exit node associated with $S_Q$, then the adaptation depicted by that lattice is correct (i.e., while the adaptation is being performed the specification during adaptation is satisfied and after the adaptation completes, the application satisfies the new specification).

\[
\square
\]

## 4 Example: Message Communication

In this section, we present an example that illustrates how transitional-invariant lattice can be used to verify correctness of adaptation in the context of a message communication program. We first describe the intolerant message communication program. Next, we discuss adaptation of adding a FEC-based proactive component to the intolerant message communication program.

**Specification of Communication Program.** An infinite queue of messages at a sender process $s$ is to be sent to two receiver processes $r_1$ and $r_2$ via a channel and copied in corresponding infinite queues at the receivers. Receiver queue contains one copy of each message sent by the sender. Faults may lose messages in the channel.

The message communication program with process $s$ and $r_1$ is shown in Fig. 2. Process $r_2$ is similar to process $r_1$. Only `send` and `receive` actions of the program are shown, since only those actions are considered for adaptation.

Processes $s$, $r_1$ and $r_2$ maintain queues $sQ$, $rQ_1$ and $rQ_2$ respectively. $sQ$ contains messages that $s$ needs to send to $r_1$ and $r_2$. $rQ_1$ (respectively, $rQ_2$) contains messages that $r_1$ (respectively, $rQ_2$) receives from $s$. Let $mQ$ be the queue of all messages to be sent. ($mQ$ is an auxiliary variable that is used only for the proof.) Initially, $sQ = mQ$. The function $\text{head}(sQ)$ returns the message at the front of $sQ$, and $\text{head}(sQ, k)$ returns $k$ messages from the front of $sQ$. The notation $sQ \circ d$ denotes the concatenation of $sQ$ and $\langle d \rangle$.

**Invariant.** The invariant of the communication program is $S1 \land S2$, where

$S1 = \forall i : (m_i \in rQ1 \lor m_i \in rQ2) \Rightarrow m_i \in mQ$, and $S2 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \lor ((m_i \in \text{ch.s.r1} \lor m_i \in rQ1) \land (m_i \in \text{ch.s.r2} \lor m_i \in rQ2)))$.

In the above invariant, $S1$ indicates that messages received by the receivers are sent by the sender. $S2$ indicates that a message $m_i$ is not yet sent by the sender, or it is in the channel, or it is already received by the receiver.
4.1 Adaptation: Addition of Proactive Component

In this section, we first describe the FEC-based proactive component. Then, we briefly discuss how the adaptation is implemented. Next, we describe the abstract version of the proactive component and the adaptation using guarded commands. Finally, we prove correctness of the adaptation using transitional-invariant lattice.

**Proactive component.** The proactive component consists of two types of fractions: encoder and decoder. The encoder fraction is added at the sender process and the decoder fraction is added at the receiver process. The encoder takes \((n - k)\) data packets and encodes them to add \(k\) parity packets. It then sends the group of \(n\) (data and parity) packets. The decoder needs to receive at least \((n - k)\) packets of a group to decode all the data packets.

**Adaptation to add the proactive component.** Our adaptation of adding the proactive component is based on the Java implementation in [6]. To add the proactive component this adaptation starts when the interceptor at the sender process traps the `send` method and blocks it. The interceptor at the receiver process replaces the `receive` method with the decoder fraction after the `send` method is blocked and channel from the sender to the receiver is empty. After the decoder is added at both the receivers, the encoder is added at the sender process by replacing the `send` method.

We now describe an abstract representation of the proactive component and adaptation. Figure 3 shows how the sender process \(s\) is adapted to use the encoder fraction. The adaptation starts when the original program action (send)
process s
var sQ : queue of integer
begin

Original Program Action

<table>
<thead>
<tr>
<th>send</th>
<th>¬Empty(sQ) → send head(sQ) to r1, r2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FEC Encoder Fraction</td>
<td>e_add</td>
</tr>
<tr>
<td>const n, k</td>
<td></td>
</tr>
<tr>
<td>var u, l, m : integer {initially, u = l = m = 0}</td>
<td></td>
</tr>
<tr>
<td>encQ : array [integer, 0..n-1] of integer {initially, encQ = Empty}</td>
<td></td>
</tr>
<tr>
<td>encode : ¬Empty(sQ) → encQ[u, 0..n-1] := fec_encode(head(sQ, n - k));</td>
<td></td>
</tr>
<tr>
<td>u := u + 1</td>
<td></td>
</tr>
<tr>
<td>fec_send : ¬Empty(encQ[l, m]) → send encQ[l, m] to r1, r2;</td>
<td></td>
</tr>
<tr>
<td>m := (m + 1) mod n;</td>
<td></td>
</tr>
<tr>
<td>if m = 0 then</td>
<td></td>
</tr>
<tr>
<td>l := l + 1</td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
</tbody>
</table>

end

Fig. 3. Adapting sender to add proactive component.

at s is blocked. This is shown in the figure by atomic adaptation s_block. The addition of encoder fraction at s is shown by atomic adaptation e_add. Likewise, Figure 4 shows how the receiver process r1 is adapted to use the decoder fraction.

Specification of program using the proactive component. Program using the proactive component satisfies the same specification as the communication program (discussed earlier in this section). Additionally, it can recover lost packets if at most k packets from a group are lost.

The invariant of the program using the proactive component is \( S_1 \land SF \), where

\[
SF = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \lor (m_i \in rQ1 \lor m_i \in data(encQ \cup ch.s.r1 \cup rbufQ1)) \land (m_i \in rQ2 \lor m_i \in data(encQ \cup ch.s.r2 \cup rbufQ2)))
\]

We use the notation \( m_i \in data(encQ \cup ch.s.r1 \cup rbufQ1) \) to imply that message \( m_i \) can be generated from the data in \( \{encQ \cup ch.s.r1 \cup rbufQ1\} \). Finally, the specification during adaptation is that \( S_1 \) continues to be true during adaptation.

**Theorem 2.** The adaptation lattice of Fig. 5 is a transitional-invariant lattice for the adaptation of adding the proactive component (shown in Fig. 3 and Fig. 4). Hence, the adaptation is correct. \( \square \)
process r1
var rQ1 : queue of integer
begin

Original Program Action

receive : rcv data from s → rQ1 := rQ1 o data

FEC Decoder Fraction ↓ r1_replace

const n, k
var x, y, p1 : integer {initially, p1 = 0}
rbufQ1 : array [integer, 0..n − 1] of integer {initially, rbufQ1 = Empty}

fec_receive : rcv data(x, y) from s → rbufQ1[x, y] := data(x, y)
decode : count(¬Empty(rbufQ1[p1, 0..n − 1])) >= (n − k) →
         rQ1 := rQ1 o fec_decode(rbufQ1[p1, 0..n − 1]);
         p1 := p1 + 1

end

Fig. 4. Adapting receiver to add proactive component.

5 Conclusion

In this paper, we presented an approach to verify the correctness of adaptation. We introduced the notion of transitional-invariant lattice to verify the correctness of adaptation. We demonstrated the use of our approach in verifying an example adaptation of adding the proactive component to a message communication application. For reasons of space, we refer readers to [9], where we discuss more examples. In [9] we also extend transitional-invariant lattice to transitional-faultspan lattice for verification of fault-tolerance properties during adaptation.

Our method also has the potential to tradeoff between the concurrency during adaptation and the difficulty in verifying adaptation. For example, if more concurrency is provided during adaptation then the size of the lattice would be large. By contrast, if the atomic adaptations were performed sequentially then the size of the adaptation lattice would be small. Thus, our method has the potential to trade off between the permitted concurrency among atomic adaptations and the difficulty in proving correctness of adaptation.

Our method is also more general than the approach used in [6] where one identifies the dependency relations among the component fractions for their correct addition and removal. Specifically, our method can be used even if the dependency relation is cyclic whereas the method in [6] assumes that the dependency relation is acyclic. Also, our method enables us to ensure that the safety specification during adaptation is satisfied. By contrast, the methods in [10] assume that the program being adapted is stabilizing fault-tolerant, and ensure that after adaptation is complete eventually the new program will recover to states from where its specification would be satisfied.
Fig. 5. Transitional-invariant lattice for addition of proactive component.

There are several possible extensions to this work. In this paper, we did not discuss techniques to generate transitional-invariant lattice. We note that techniques that are developed to calculate invariants can also be used to find transitional-invariants. In this context, one extension to this work is to generate the lattices automatically, given the invariants of the application before adaptation and after adaptation. Based on the adaptation requirements, the automatic generation of the lattices would enable us to identify different paths to achieve adaptation and also ensure correctness of those paths. Further, cost analysis techniques used in [11] can be used to choose the best path among all the correct paths for adaptation.

References


Strategies for a Component-Based Self-adaptability Model in Peer-to-Peer Architectures

Sascha Alda and Armin B. Cremers

Department for Applied Computer Science, University of Bonn, 53117 Bonn, Germany
{alda,abc}@cs.uni-bonn.de

Abstract. Current peer-to-peer architectures are hardly resistant against unanticipated exceptions such as the failure of single peers. This can be justified by the absence of sophisticated models for exception detection and resolution in peer-to-peer architectures. On the other hand, existing generic models for such self-adaptable architectures are rather theoretical and less suitable for the usage by end-users. In this work, strategies for a new self-adaptability model in peer-to-peer architecture are presented incorporating the component technology as the conceptual foundation. The claim of this approach is that through the intuitive nature of the component technology the process of self-adaptability becomes more applicable and more comprehensible even for less experienced end-users.

1 Introduction

The ability to adapt software architectures during use time nowadays is a crucial requirement, in order to deploy software for working contexts and tasks that cannot be anticipated during design time. In this regard, it must be taken into account that software adaptation is mostly tackled by (inexperienced) end-users rather than by highly sophisticated experts such as administrators. The process of end-user adaptability of software (end-user tailorability [8]) can be enhanced for instance by the comprehension of adaptation tools or helping systems. The adaptability of distributed software architectures marks a couple of further challenges for developers. Recent architectures like peer-to-peer architectures [3] assume an unstable, dynamic topology as an important constraint. The consequences of such dynamic architectures are unanticipated exceptions within the architecture like the failure of single peers or the unavailability of services. These exceptions constitute further reasons why software architectures have to be adapted, in order to avoid any kind of misbehavior of the architecture.

Though conventional adaptation environments could already be utilized to resolve occurred exceptions, they are usually not designated to detect and to report them to the system or to end-users. In this context, so-called self-adaptable architectures constitute a more promising approach. Self-adaptable architectures are capable of identifying circumstances that necessitate adaptation, and accordingly, to select and effect an
appropriate course of action [10]. However, the analysis of recent peer-to-peer architectures such as the JXTA architecture [12] shows that sophisticated options to detect and to resolve these circumstances are hardly implemented, making them less resistant against exceptional cases. On the other side, existing self-adaptability models for software architectures [4] [5] [9] are rather generic and complex. Most of them describe a fully autonomous architecture capable of detecting and resolving exceptions stand-alone. End-users being accomplished to impact the adaptation of an architecture are almost disregarded. These issues make it difficult and not worthwhile to adopt existing models to peer-to-peer architectures that can be maintained by end-users.

This paper presents intermediate results of a research project [2] that examines, to what extend the widely accepted component technology [13] is suitable to serve as a conceptual foundation for a self-adaptation environment for peer-to-peer architectures. The fundamental idea is to adopt the intuitive strategies for the creation of component-based architectures also for the adaptation of these architectures. Likewise to existing assembly tools that allow composing applications by defining connections between pre-defined components, self-adaptation environments could assume these operations as a possibility to change the behaviour of a given composition in case of exceptions. The claim is that a component-based approach for self-adaptability would in particular enable end-users to maintain self-adaptable peer-to-peer architectures. It will be figured out to what extent existing component-based adaptation strategies and environments stemming from previous research [11] [14] have to be reconceived, and during which phases they can actually facilitate the overall process of self-adaptability.

The rest of this paper is structured as follows: chapter 2 motivates the demand as well as the requirements for a self-adaptable peer-to-peer architecture by means of a simple scenario. Chapter 3 presents the DeEVOLVE architecture that features a component-based self-adaptability model. Chapter 4 finally concludes this paper.

2 A Simple Scenario: A Distributed Groupware Application

In figure 1, a simple scenario of a peer-to-peer architecture is depicted representing a distributed groupware system.

![Fig. 1. A basic peer-to-peer architecture representing a groupware.](image-url)
This architecture consists of four different peers. Peers A and B both provide services for checking the spelling of a text and for sending emails over an IMAP service, respectively. The owner of peer C has composed these services with his local email client service towards an extended mail client. At the same time, the user of peer D has integrated the email client service of C in his local groupware suite. This example demonstrates the two roles of a peer: peer C does function as a client and a server at the same time. The user of a peer can even adopt three roles: as the provider or consumer of a service, and as a composer, who composes services with other services.

The services displayed in figure 1 are supposed to be realized by software components, which are deployed on the respective peers. Software components can be seen as self-contained units of composition with contractually specified interfaces [13]. The interaction among components takes place only through these interfaces. The interaction between two or more components may yield to syntactic and semantic dependencies between the respective components [1]. Syntactic dependencies assume that a communication between two components actually takes place, such as event or data flows. Semantic dependencies describe semantic constraints among components. An example for a semantic dependency could be a constraint between two attributes. The peer-to-peer architecture visualized in figure 1 abstracts from these different occurrences of dependencies; a thin line between two components simply indicates either a syntactical or semantic dependency. Within a peer-to-peer architecture, syntactic dependencies are mostly violated, if peers fail or if services become unavailable. Semantic dependencies can also be violated during the regular usage of an application. The violation of a dependency can be considered as an exception on an architectural level. If these exceptions are not detected and handled adequately, the affected component will not behave as intended. The failure for instance of Peer A or B would affect not only the functionality of the component of peer C, but also of peer D. The handling of exceptions raises some problems that have to be faced:

- **How can exceptions be detected?**
  Exceptions within a component-based architecture often cannot be detected on the level of components. This can be reasoned by the fact that developers of components cannot anticipate all contexts, in which third parties will deploy a distinct component and, thus, the possible exceptions it may cause or it has to detect. Consider the email client component of peer C: the integration of the remote IMAP service requires detecting the potential failure of a remote peer. A component that does assume a local IMAP service would definitely be inappropriate for this remote usage.

  Besides, exceptions might also occur due to context-sensitive interactions between components that cannot be assigned to a single component [5]. An example would be a deadlock situation among three components. This work claims that exceptions should preferably be detected on an architectural level by the runtime environment, in which the components are deployed rather than on component level.

- **Who should define exception handlers?**
  Exception handlers represent plans to resolve an occurred exception. Analogous to the detection of an exception, developer of components cannot foresee the required steps to resolve an exception. A resolution plan strongly depends on the intended
use of a component. Consider for instance the failure of peer B in figure 1: the resolution plan for peer C could be to establish a connection to a redundant spell service residing on another peer. For the case that no redundant service is available, not only peer C, but also all dependent peers (here: peer D) have to be notified about the lost service. An email component that does assume a single user mode would probably not incorporate the notification of other remote peers. This work claims that the end-user or composer of a composition himself should determine and influence the necessary resolution plan. The end-user could be involved in the process of exception resolution, for instance, if multiple handlers have been defined.

• **How can exception handling be inserted into a given composition?**

Usually, components are available only in binary form, making it impossible to enhance the code of single components. The composition of single components is often declared by means of formal notations such as architecture description languages (ADLs). These notations could be enhanced by a formalism for the handling of exceptions. In fact, existing ADLs (see [7] for an overview) provide only minor support for the explicit handling of exceptions on an architectural level and are, thus, rather inappropriate for the flexible composition of services within a peer-to-peer architecture.

A fully self-adaptable peer-to-peer architecture would not only detect an exception, but would also select one of the pre-defined handlers to resolve the exception. However, the proposed instrumentation of compositions with exception handlers and the selection of handlers through end-users, requires end-user involvement in the process of self-adaptation. As end-users cannot be expected to have proficient adaptation skills on a technical level, one has to account for intuitive adaptation strategies and auxiliary tools that utilize these routines. Adaptation strategies should be provided on different levels of complexity [6] [8]. Users with humble experience can adopt to less complex strategies, while more acquainted user can revert to more complex strategies.

3 **DEEVOLVE: A Self-adaptable Peer-to-Peer Architecture**

This chapter introduces DEEVOLVE, our notion of a component-based, self-adaptable peer-to-peer architecture. DEEVOLVE is a further development of the FREEVOLVE tailoring architecture [11] aimed to deploy component-based client-server applications. Both architectures adopt the component technology as the foundation for the flexible adaptation of deployed compositions. DEEVOLVE extends the original adaptation environment by accomplishing the detection of exceptions on an architectural level. An exception can be resolved either by the system itself or in interaction with an end-user. In the following sections, we briefly present the structure of DEEVOLVE and elaborate the self-adaptability model as conceived for this architecture.
3.1 The Structure of DEEVOLVE

The DEEVOLVE peer-to-peer architecture mainly serves as a runtime environment for deploying so-called peer services. Peer services are considered as compositions of single components. These compositions can be advertised and published (that is, made available) to other peers, as well as discovered and composed with other peer services by other third-party peers. A component model called FLEXIBEAN does thereby prescribe the structure and the valid interaction primitives for both local interaction within a single peer service and remote interaction between two distributed services. There are two interaction primitives for components, event notification and interaction through a shared object. Shared objects serve as an abstraction for a data flow between two components. The remote interaction between two remote components is accomplished by the explicit integration of the Java RMI technology into the FLEXIBEAN model. A language called CAT formulates the composition of components to define peer services. Another composition language called PeerCAT allows the flexible composition of different peer services towards concrete applications.

DEEVOLVE is built on top of the JXTA framework [12] to realize basic operations like the advertisement, publishing, and discovery of peer services. Furthermore, DEEVOLVE adopts the peer group concept of JXTA that enables peers to organise into self-governed peer groups independent of any organizational or network boundaries. The purpose of peer groups is to restrict the access to certain peer services for authorized peers only. DEEVOLVE is accompanied by a couple of useful tools supporting the discovery and composition of peer services, the definition and advertisement of single peer services, and the management of peer groups. Besides, tools for the tailoring of compositions during runtime are yet provided. The current efforts to realize DEEVOLVE towards a self-adaptable architecture will be outlined in the next section.

3.2 Component-Based Self-adaptability

Component-based tailoring environments as originally conceived in [11] enable users to compose software behaviour as needed. These environments together with their fundamental tailoring strategies serve also as the base to have a self-adaptable peer-to-peer architecture that is enabled to modify compositions in case of exceptions. What is primarily missing for a self-adaptable architecture is the ability to detect and to report exceptional cases on an architecture level during use time. The resulting phase model for detecting and resolving exceptions is pictured in figure 2.

The rationale of this approach is to define exception handlers in the corresponding PeerCAT description of the composed application. The insertion of handlers in a compositional description is referred as instrumentation and should be mastered by an end-user. Exception handlers define the type of an exception, the context in which an exception occurs, as well as the actual handlers to resolve the exception in a declarative manner. Basic exception types are the failure of remote services, the time-out of a request to a service, the violation of semantic constraints of attributes, or the loss of
access rights to services\textsuperscript{1}. Further, aggregated exceptions types allow the definition of composed exceptions like the failure of two services within a composition. The context indicates the peer service, in which an exception might arise. For handling the violation of attribute constraints, the respective attributes of a service have to be declared additionally. In the exception handler part, handlers are defined, explaining how to resolve an exception. For the phase of exception resolution we refer to the same adaptation strategies as realized in the tailoring environments. The idea is to take over the principles for the building of component-based architectures also for the modification of these architectures in the case of exceptions. The following types of adaptation or resolution strategies can be applied:

- Changing attributes of single components in the case of violated constraints
- Changing connections between two components, for instance, if a remote component has failed due to the failure of the hosting peer
- Changing the component composition by inserting, replacing, or deleting components within a composition. This type does also include the discovery of an alternative composition within a peer-to-peer architecture.

In order to handle the failure of a remote peer, DEEVOLVE keeps track to which peers a distinct peer has made up a connection for the consumption of a service. These peers are pinged in regular intervals. If no response occurs, an exception is assumed and all registered exception handlers of the affected services will be invoked. For a given exception context (i.e. service) multiple handlers can be declared. After the system has detected and reported an exception for a distinct context, the user can select, which handler is to be executed by the system to resolve the exception. This makes sense, since users normally cannot anticipate the necessary steps to resolve an exception in advance. Alternatively, if no handlers could have been defined, the user himself can

\textsuperscript{1} The access right to a service is realized by the JXTA Membership Protocol, which allows peer group owner not only to grant, but also to resign users the access to a service within a group.
pursue the adaptation individually. Then, the system simply acts as an indicator to
notify users about the occurrence of an exception. If only one handler has been de-
clared, the system can be allocated to execute the defined adaptation routines without
any confirmation from the user. This leads to a completely autonomous system. The
composition is re-started, after the adaptation has been finished and confirmed.

For handling exceptions within the composed email client service depicted in figure
1, the IMAP and spell checker service constitute two primary contexts to be consid-
ered. A necessary exception type for the important IMAP service would be the failure
of this service. In case of such a failure, the immediate discovery and integration of an
alternative mail service should be declared in the respective exception handler part. A
suitable handler for the spell checking service could be a notification to all dependent
consumers, indicating that this service is temporarily not available. An attribute con-
straint for the email service could be defined for the length or the structure of the mail
message, which is to be transmitted to either remote service. If the length falls above a
maximum value or if the message contains illegal strings (e.g. html strings), then the
users are also to be notified about these exceptions and advised to adjust the message.

Exception handling as explained so far does not require extending the code of sin-
gle components, but only the declarative compositional description of a composition
by an end-user. On the other hand, developers of components can concentrate on the
implementation of the business logic of components, but do not have to care about the
handling of exceptions. However, under some circumstances it might be more reason-
able to handle an exception locally within a component, for instance to ensure a cor-
rect and exact course of action to resolve an exception. In order to notify components
about the event of an exception, dedicated interfaces (ports) of a component can be
bound to predefined system event ports of DeEvolve. If an exception occurs, an event
object, which does encapsulate all necessary information about an exception (e.g. the
context), is sent to the component. For this exception handling on a service level, the
constituting components have to be instrumented by additional statements on a code-
level. For the actual resolution, components can utilize the same adaptation strategies
as brought above through a dedicated interface called TAILORING API, which is pro-
vided by DeEvolve. The binding between the system ports and ports of component is
described in the PeerCAT description of the respective composition.

A self-adaptable architecture as considered in this work necessitates the adaptation
of a given composition in two ways: firstly, the instrumentation with exception han-
dlers and, secondly, the declaration of adaptation strategies during exception resolu-
tion. For both, adequate graphical and textual tools will be supplied in the future.
Moreover, both instrumentation and adaptation strategies fulfill the demand imposed
for adaptation environments to provide strategies on different levels of complexity
(section 2). In accordance to the distinction brought in [6], the adaptation strategies do
in particular enhance the selection of alternative behaviour and the construction of
new behaviour on the base of existing elements (see [14] for an extensive analysis).
Instrumentation also supports the re-implementation of components (see table 1).
Table 1. Summary of the different types for the instrumentation of compositions.

<table>
<thead>
<tr>
<th>Tailoring Strategy [6]</th>
<th>Type of Instrumentation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative Selection</td>
<td>Instrumentation of Components</td>
<td>Useful to define constraints for attributes belonging to a single component. An exception might occur, if the value of an attribute falls below a prescribed minimum value. A handler could adjust this value.</td>
</tr>
<tr>
<td>Construction of new behaviour on the base of existing elements</td>
<td>Instrumentation of Dependencies</td>
<td>Useful for the exception handling in case of broken or violated dependencies between two components (e.g. due to the failure of peers, time-out of requests). A possible handler could be the definition of one or more alternative links between two components.</td>
</tr>
<tr>
<td>Instrumentation of Compositions</td>
<td></td>
<td>Useful for an advanced exception handling within a composition, for instance to define aggregated exceptions for multiply broken dependencies.</td>
</tr>
<tr>
<td>Re-Implementation</td>
<td>Instrumentation of Components (Code-Level)</td>
<td>Useful to handle exceptions without the involvement of users. This can ensure the exact and correct course of action to resolve an exception.</td>
</tr>
</tbody>
</table>

4 Conclusion

This work has elucidated the demand for self-adaptability within peer-to-peer architectures. The adoption of the component technology as the foundation accomplishes to utilize the intuitive strategies of this technology not only for the resolution of occurred exceptions, but also for the preliminary instrumentation of component compositions with exception handlers. This approach enables in particular end-users to maintain peer-to-peer architectures. Based on these considerations, the peer-to-peer architecture DEEOLVE has been presented that implements a component-based self-adaptability model. At the time of writing this position paper, the implementation is not yet finished. Hence, the prototypical realization constitutes the most crucial milestone for future work in this research project. Another integral part will be a user evaluation to assess, how the adaptation strategies are actually appreciated.

References

Classifying Software Component Interoperability Errors to Support Component Adaption

Steffen Becker¹, Sven Overhage², and Ralf H. Reussner¹

¹ Software Engineering Group, University of Oldenburg
OFFIS, Escherweg 2, 26121 Oldenburg, Germany
² Dept. of Software Engineering and Business Information Systems
Augsburg University, Universitätsstraße 16, 86135 Augsburg, Germany
{steffen.becker,ralf.reussner}@informatik.uni-oldenburg.de
sven.overhage@wiwi.uni-augsburg.de

Abstract. This paper discusses various classifications of component interoperability errors. These classifications aim at supporting the automation of component adaptation. The use of software components will only demonstrate beneficial, if the costs for component deployment (i.e., acquisition and composition) are considerably lower than those for custom component development. One of the main reasons for the moderate progress in component-based software engineering are the high costs for component deployment. These costs are mainly caused by adapting components to bridge interoperability errors between unfitting components. One way to lower the costs of component deployment is to support component adaptation by tools, i.e., for interoperability checks of (semi-)automated adaptor generation. This automation of component adaptation requires a deep understanding of component interoperability errors. In particular, one has to differentiate between different classes of interoperability errors, as different errors require different adaptors for resolving. Therefore, the presented classification of component interoperability errors supports the automation of component adaptation by aiding automated interoperability problem detection and semi-automated adaptor generation. The experience gained from already implemented solutions for a specific class of interoperability errors provides hints for the solution of similar problems of the same class.

1 Introduction

The vision of building software by simply composing existing software components available on component marketplaces or intra-corporate component repositories is quite old. Nevertheless, we are still struggling to enable the simple and cost-effective use of software component. The well-known paper of Garlan et al. [1] seems to be a bit outdated from a technical point of view, but regarding the major aspects of architectural mismatches between components it is still up to date.

Although the component interoperability errors are not new (as the above reference shows), and have been reported as architectural mismatches, interoperability errors, semantic heterogeneities, etc., a comprehensive and systematic classification of such problems is still lacking.
In particular, the aspect of building adaptors to bridge incompatibilities between two components is still under research. This field of research is aiming at generating adaptors which solve the interoperability problems encountered. The envisioned generation process is supposed to create as much bridging code as possible using available architectural information. In case of insufficient information to a fully automated generation of the adaptors, the generator helps by generating a code skeleton. This skeleton helps the developer to write the missing code parts efficiently.

One of the major challenges when attempting to write a suite of generators for a given problem space is to understand the problem in deep and to conquer it by dividing it into several subproblems. In general, this results in a classification of the subproblems of a given problem. This classification can then be used for writing a generator for each of the classified subproblems, i.e., to generate a standardised solution to a well known class of difficulties. By identifying interoperability problems of a specific class, the appropriate generator is then used to tackle the interoperability problem. If it is impossible to provide a generator for a given problem, the classification is also useful as it can be used to search knowledge bases for similar difficulties and their solution. Note that the focus of this paper is therefore the development of a classification of interoperability problems. A deeper investigation of the solutions to the classified problems - either with or without generated code - is beyond the scope of this paper and subject of future research.

A good classification framework is supposed to support the problem analysis phases as efficiently as possible. Therefore it is essential that the framework is designed methodical. It is important that one knows for which kind of problem one has to search. Thus, the classification schema should determine the problems to look for and the sequence of steps used while comparing the interoperability of two components.

The contribution of this paper is a presentation and an evaluation of several classifications of component interoperability errors. We investigate two well-known schemas as a base for two new classification schemes developed by the authors. By an example we show the usage of the presented classification schemes. After a comparison of the results the advantages and disadvantages of the approaches are presented and we recommend the usage of one of those schemes for classifying software component interoperability problems.

This paper is organised as follows. In the next section, possible classification approaches are presented to the reader. In section 3, we demonstrate the application of the classifications by examples. Afterwards, we compare the applicability of the presented schemes in section 4. Finally, we present related work to the one presented in this paper and conclude with an overview on future research directions.

2 Classification Approaches

The difficulty of correct component interoperation stems from the complexity and diversity of component interaction. In particular, most often the correct component interaction relies on assumptions component designers implicitly made about their component’s environment. Components might offer services with different names and/or parameters than expected, they might disagree on the order of the service calls or there could be incompatible expectations regarding Quality of Service (QoS). Even worse, there might be a mismatch caused by different understandings of the underlying domain.
Adaptors on various levels are used to bridge such interoperability problems. However, the application of adaptors requires that interoperability errors are detected and classified first. Hence, there must be sufficient information on a component, its intended interaction with other components and the assumptions on its environment. Although a component’s code implicitly contains some of this information, due to the black-box nature of components, this information should be provided in the component’s interfaces. This view is also supported by the fact that assessing components by testing them is not appropriate. But to get comprehensive information on the interaction of components, rich interface specifications are needed which contain the required attributes to detect as much interoperability problems as possible. However, the mere specification of interaction in component interfaces does not imply the feasibility of automated interoperability checks, due to reasons of computability. For example, if component interaction protocols are specified by push-down automata (a model capable of specifying most practical relevant protocols), the inclusion and equivalence of such protocols are in general undecidable, i.e., no interoperability or substitutability check can be performed. To summarise, a classification schema for software component adaptors highly depends on the underlying component interface model.

In the following we first present two well-known schemes. One is based on a classification schema often used to classify linguistic communication problems (section 2.1). The other one is based on a more technical view by using different layers of interface descriptions as the classifying dimension (section 2.2). The first schema developed by us is based on a combination of a technical and a quality oriented classification dimension (section 2.3). The last schema is even more detailed by classifying the domain related problems further than the one before (section 2.4). The order in which the schemes are introduced is based on the complexity and classification power - from basic to comprehensive. Always keep in mind that these classifications distinguish problems on a very high abstraction level in order to be as complete as possible. It is clear that each class of problems can be easily divided in more detailed problem classes and that this might be necessary when constructing a real world generator suite to solve some of these problems. Future work is going to investigate in some of the presented classes in more detail.

2.1 A Linguistic Classification

A classification schema often used for business applications is based on the distinction between syntactical and semantical specifications. Semantical information is further divided into static and dynamic semantics (often called pragmatics).

As an example for such a classification we take a look at some proposed enhancements to the UDDI specification framework. This proposal introduces the three mentioned classes and adds further on the classes technical and general specification data.

General information is used to capture marketing information, i.e., publisher and conditions of the component’s usage. The technical specification is used to describe technological details about the underlying component framework, i.e., communication protocols or standards like CORBA or COM.

The syntax layer refers to the interface of the component and contains information on the signatures of the interface-methods and the data types used in those methods’ parameters or return types.
On the static semantical layer a characterisation of the component's behaviour is
given. Its concern is on the one hand the definition of the implemented concepts of the
component. This is where one should find exact definitions of the used terminology and
its meaning, e.g., if the meaning of price is understood containing VAT or not. On the
other hand a more technical viewpoint describes the static semantics containing pre- and
postconditions like the ones used in the design by contract paradigm.

On the dynamic semantic layer there should be a specification of the implemented
processes of the component. This is also differentiated into a conceptual and a technical
view. In the conceptual view the workflow for which the component was designed is
specified. Analogously, the technical view specifies the component’s protocol, i.e., the
valid order in which the component’s services can be called. Moreover there is often the
demand for the specification of the external service calls performed by the component
and the constraints on this interaction.

2.2 A Classification Based on Hierarchical Interface Models

A possible approach is to take a closer look on the interfaces known nowadays (see figure
1, left side) [6,7]. The approach is comparable to the one proposed by Beugnard (see
figure 1, right side, according to [8]).

![Fig. 1. Hierarchies of Interface Models.](image)

The specification used in this approach is based on a hierarchy of interface models.
In the proposed classification those models have been structured in three layers. The
hierarchy is based on the fact, that checking for incompatibilities on a higher layer
requires interoperability on the lower layers.

On the signature list layer the main concern is the specification of single
methods. Each method is specified by its signature, i.e., by its name, its ordered list of typed
parameters, its return type and its unordered list of exceptions potentially thrown by
the method. This kind of interfaces is standard in object oriented systems (examples are
CORBA-IDL [9] or Java interfaces [10]).

While signature lists specify each single method, they lack information about the
methods’ relations. Interface models on the protocol layer specify constraints on the
sequences of calls of the methods. Protocols can be specified in a number of different ways ranging from the use of finite state machines over the use of temporal logic to petri-nets.

Protocols specify one aspect of the component’s behaviour, namely ordering dependencies between services. This means, protocols do not specify the complete behavioural semantics of the component. However, this concentration on one aspect of the behaviour allows to use non-turing universal formalisms, such as finite state machines, having a decidable inclusion problem. As the inclusion check of two interfaces allows substitutability checks and interoperability checks, a protocol description with a decidable inclusion problem is of practical interest. While finite state machines have the benefit that substitutability and interoperability checks refer to language inclusion, their descriptive power is rather limited. For example the provides protocol of a stack (no more pop-operations than push-operations) cannot be described by finite state machines. Therefore, weakened substitutability relations defined for more powerful calculi of protocol specification, such as process algebra based approaches [11,12] and predicate based approaches [13,14] or counter-constrained finite state machines [15] have been investigated. However, the more powerful a protocol description technique is, the more costly are interoperability and substitutability checks, in particular if model checkers or theorem provers have to be involved. The use of interactive theorem-provers makes an automated check of interoperability or substitutability impossible.

Besides the protocol layer, Beugnard introduced an additional layer on which he proposed to add synchronisation aspects of the interface - e.g., re-entrance of methods, mutual exclusions of concurrent method calls, etc.

Interface methods not only include functional aspects but also non-functional ones. Nowadays there is an increasing research interest in the field of Quality of Service (QoS) attributes. For this reason, a layer enhancing the former layers by Quality of Service information is added on top of the hierarchy.

2.3 An Enhanced Classification on Interface Models

Based on the approach in the previous section we propose a more systematic classification of the heterogeneities in question by introducing an additional classification dimension. The first dimension distinguishes between functional and non-functional aspects (some might also say extra-functional aspects). The other dimension is based on the granularity of the interface description - quite similar to the previous approach. The introduction of the additional dimension allows a more detailed differentiation of problems which were interwoven in previous approaches. Using these dimensions results in the classification matrix depicted in figure 2.

We present a short explanation of the classes introduced by these dimensions. Regarding single methods of an interface specification there are signature information on the functional view and method related QoS on the non-functional view. Signature specifications are similar to those explained in section 2.2. Quality attributes of single methods are described on the non-functional view - e.g., as introduced by QML [16]. Examples for possible attributes contain efficiency of the method (with respect to CPU and memory utilisation) or reliability. Others can be found in the ISO 9126 [17].
On the next level of detail there are attributes of the interface. Dependencies in the call-order of single methods and similar coordination constraints (e.g., re-entrance and similar concurrency constraints, transactional execution, etc.) are relevant on the functional view. On the non-functional view we consider interface related quality attributes like maintainability or the possibility of reusing the component in different contexts.

The last class of interoperability problems results from different understandings of the underlying domain. The functional view is caused by different semantics of the used specification languages. In particular, when using natural languages for the domain specification, misunderstandings by different interpretations of the same specification occur frequently. Other reasons can be induced by unequal work flow processes or a varying understanding of the tasks of the respective domain. The non-functional view on the domain contains heterogeneities resulting from constraints in the specific domain which have an impact on the used components - i.e., an online banking component is expected to support a special encryption method or to perform the business operations within certain bounds for its reliability.

2.4 The Unified Specification of Components Framework

A novel classification of component heterogeneities can be derived from the Unified Specification of Components (UnSCom) framework, which is currently under development by the working group 5.10.3 of the German Computer Society (G.I. e.V.). This specification framework forms the basis of a composition methodology (an introduction is given in [18]). The framework aims at providing a common, standardised source of information about component characteristics that can be utilised by multiple development tools. Accordingly, the provided information can also be used to assess component incompatibilities and generate appropriate adaptors (if possible).

The framework builds upon the fundamental assumption that component characteristics relevant for the composition process manifest themselves at the component interfaces, which (following the black-box principle) completely characterise components. In order to systematically identify and structure component characteristics, the framework analyses the interfaces of components and uses a classification schema to identify various component characteristics.

The introduced classification schema aims at providing a systematic and complete structuring of component characteristics. Therefore, it distinguishes between different perspectives on components (see figure 3). First of all, it differentiates between different development perspectives which correspond to the main stages of the component development process. Orthogonal to that, it distinguishes between three design views which...
are commonly used to (completely) describe the characteristics of software artefacts [19,20,21,22,23]. By putting these different kinds of perspectives into relation, relevant component characteristics are identified and described.

The development perspectives listed on the horizontal dimension comprise a domain-related, a logical, and a physical quality perspective. Thereby, the domain-related perspective, which corresponds to the domain (or conceptual) modelling phase of the development process, characterises the functionality that is either being provided or required at a component interface. The logical perspective, which corresponds to the architectural design stage, yields information about the architectural design of a component interface, which is utilised by component models like, e.g., the CORBA component model. The physical quality perspective contains non-functional characteristics, which determine the quality that is either being required or provided at an interface.

The design views denoted on the vertical dimension respectively put the focus on specific parts of a development perspective. The static view resembles the structure (arrangements) of a software artefact, the operational view yields the effects of its operations, and the dynamic view details the interactions that a software artefact participates in. Note that the operational view is sometimes being neglected by software engineers: It is either commonly addressed together with the dynamic view as artefact behaviour or, alternatively, together with the static view as artefact form.

The combination of development and design perspectives yields a total of nine classes of component characteristics. These classes are used to assess the compatibility of components and identify possible heterogeneities. Accordingly, the UnSCom framework yields nine classes of component adaptors.

*Functional incompatibilities.* Between components arise from misunderstandings of the domain-related functionality, i.e., different interpretations of domain-related concepts between components. According to the introduced design views, functional incompatibilities refer to the structural information objects. Such objects can be conceptual entities (e.g., a different, homonym understanding of the concept "price", which either includes or excludes VAT). Furthermore, functional heterogeneities refer to the effect of domain-related operations (e.g., does creating a balance mean creating a US-GAAP or IAS balance). Finally, there are process-related functional heterogeneities (e.g., a good is ordered on account or debit advice).

*Architectural incompatibilities.* Stem from incompatible architectural component designs, i.e., different interface layouts. Architectural mismatch refers to the structure of an interface, i.e., to type and property (attribute) declarations (e.g., differing names). In
addition, architectural incompatibilities refer to an incompatible understanding of the interface effects, i.e., to the declarations of events, exceptions, and methods as well as to the associated pre- and post-conditions (assertions). Finally, architectural mismatch originate from mis-fitting method invocation sequences or synchronisation constraints, i.e., from diverse interaction protocols.

*Non-functional incompatibilities.* stem from differing assumptions on the non-functional characteristics of components. Following the ISO 9126 quality model [17] and the introduced design views, non-functional incompatibilities may originate from differing structural characteristics, assumptions on design aspects, and mismatching dynamic characteristics. Thereby, structural quality characteristics are primarily relevant to software engineers who reuse components during an application development project. They can be classified into characteristics relating either to the component usability (e.g., the documentation or the time to use), maintainability (e.g., testability), or portability (e.g., the utilised implementation platform) [17]. In contrast to this, differing assumptions on design aspects and incompatible dynamic characteristics interfere with the interaction of components. Assumptions on design aspects may be classified according to their respective concern (e.g. security, persistency etc.). Dynamic characteristics refer to the component reliability (e.g., meantime between failures, recovery etc.) and its efficiency (e.g., response time, throughput etc.) [17].

### 3 Example

To demonstrate the application of the presented classification schemes the following example is introduced. Note that only the interoperability errors are illustrated and classified – the solution of those errors is beyond the scope of this paper and will be investigated in future work.

At universities there is the need for managing resources like lecture halls. Hence, a component is required that abstracts from a direct view on a database and offers the needed domain functionality for resource management on its provides interface.

Given the following domain analysis the component should enable its users to book reservations for certain rooms, cancel those reservations and lookup the reservations already known to the system. We limit the example to that functionality for simplicity reasons. It should be easy to expand it with additional tasks like printing a reservation overview for every room, getting proposals for free rooms after specifying the needed room capacity and time, etc.

#### 3.1 Signature Mismatch

As all the schemes classify in one or the other way syntactical differences. An investigation of the signatures identified for the required methods is done. These signatures are compared with the signatures of the component being checked for interoperability with the identified requirements (see code examples 1 & 2).

It becomes clear that the signatures of the single methods would mismatch without adaptation. For example notice the different names of the methods, the different parameter types or the additional exception which can be thrown by the method
ReservationID ReserveRoom(DBID database, RoomID room, DateTime startTime, DateTime endTime) throws RoomBlockedException;
void CancelReservation(DBID database, ReservationID id);
ReservationID[] QueryRoomReservations(DBID database, RoomID room, DateTime startTime, DateTime endTime);

Code example 1. Requires Interface.

void OpenDatabase(DBID database);
void CloseDatabase(DBID database);
ReservationID Reserve(ResourceID res, DateTime startTime, TimeSpan duration) throws RoomBlockedException;
void Cancel(ResourceID res, DateTime startTime) throws NoReservationFound;
ReservationID[] LookupReservations(ResourceID res, DateTime startTime, DateTime endTime);

Code example 2. Provides Interface.

CancelReservation. If we assume for a moment that the methods intentionally do the same then all of those problems are bridgeable by adaptors, e.g., calculating the duration of the reservation from the startTime and endTime parameters of ReserveRoom is a simple task for an adaptor. Those problems are subsumed by the presented schemes as syntactical, signature related or method related problems respectively.

3.2 Method Specific Quality Mismatch

In addition to the problems mentioned above more interoperability problems arise when looking at the QoS of the single methods. The required and provided QoS usually is specified by the use of QML but here a more informal description is used for simplicity reasons. Consider that the booking system is expected to be quite reliable as it is more important to have the booking system available than a quick response time. So a reliability of 99% is being demanded and as a trade off response times up to 10 seconds are acceptable. The components are unable to interact correctly if the component being checked is unable to operate that reliable. Problems of this kind can be classified as signature/non-functional, dynamic/implementation or as QoS conflict by the respective classification schemes.

3.3 Protocol Mismatch

After investigating the interoperability problems on a signature basis the next class of problems can arise on a dynamic view. Therefore we focus on the protocols required respectively provided by the example components used. There is no hint in the requirements that a specific kind of protocol should be supported - therefore the requires protocol simply states that every function can be called at any time. Looking at the provides protocol of the component being analysed it can be seen that a different method for activating the reservation database has been implemented than the one the needed component expected.
In the design for the requires interface we made the assumption that the database ID has to be passed on each method invocation. The producer of the component offering the reservation services implemented that in a different way. The component expects a database selection (OpenDatabase) before any method reservation related functionality can be called. Additionally the component expects a call to CloseDatabase after the relevant actions have been taken. The resulting protocols can be seen in figure 4.

![Diagram of protocol mismatch](image)

**Fig. 4. Mismatching component protocols.**

Note that it is again possible to bridge the resulting incompatibility by the use of adaptors\(^1\). Problems of the kind depicted here can be categorised as semantic problems, protocol, interface/protocol or architectural/dynamic be the respective classifications.

### 3.4 Interface Specific Quality Mismatch

A different issue that can be seen from the former example is related to non-functional aspects of the outlined interfaces. The producer of the component offering the reservation services had a different reuse idea in mind. Therefore the component was designed with additional Open-/CloseDatabase functions with the aim to produce a component which might be used in additional reuse contexts. It is well known that the interface design influences reusability and maintainability. As a result the enhanced interface model classification and the UnSCom classification matrix consider these attributes as interface/non-functional respective implementation/dynamic.

### 3.5 Domain Objects

After checking interoperability on the signature and interface level there is still a chance that the components might not work as expected if composed. The problems that still might arise can result from a different understanding of the underlying domain. Belonging to the domain are the domain entities, functions and processes which can be

---

\(^1\) Note that if the requires interface resulted from a design decision one might also consider adapting the design to the available component.
seen in the UnSCom matrix in the first column. In our example there might be a different understanding of the word "resource". The required interface implies lecture hall reservations when talking about resource reservations. The provided component might use the word in a more wider sense to support a larger amount of customers. For that component a resource might be something more abstract which can represent rooms, cars, video projectors, etc. Especially with cars it might be possible to have more than one reservation for a given time, e.g., if there is some kind of car sharing established which is impossible for lecture halls.

So it can be seen that there are domain related interoperability problems. The linguistic approach classifies these problems as either semantic or pragmatic. In the original interface model based system it is impossible to detect such problems at all. In the enhanced version they are classified as domain/functional and as said before in the UnSCom matrix they can be found in the first column.

3.6 Domain Constraints

The last two schemes also support a non- or extra-functional view on the domain related interoperability problems. In the enhanced interface model they are called domain constraints and in the UnSCom matrix they can be found as functionality aspects (operational/implementation). Problems of that kind result from aspects of the domain not directly related to the functionality but to additional constraints on the components - e.g., a banking software in Germany is expected to support the HBCI protocol resulting in a standardised security architecture.

4 Comparison

After presenting the different classification approaches in section 2 and the examples in the previous section we take a closer look at the presented schemes, compare the approaches, and highlight their advantages or disadvantages respectively.

In so doing, it becomes obvious that the linguistic classification approach does not contain all the kinds of component heterogeneities that are being covered by the other approaches. Especially, the linguistic classification lacks support for different development views and only distinguishes between a conceptual (domain-related) and technical view at the semantic levels. As a consequence, it does not cover non-functional component characteristics and the form of heterogeneity that arises from them. Moreover, the technical view mixes architectural and technical component characteristics and does not appear to provide a systematic way of structuring for them. For these reasons, the linguistic classification is based on an empirical understanding only.

Some of these shortcomings are avoided by the classification of component characteristics that is based on hierarchical interface models. This classification uses different development perspectives and distinguishes between functional and non-functional characteristics. However, in its original fashion, the classification approach does not cover the component functionality (although this is the most important criterion to select a component for composition during application development). Moreover, it does not systematically structure the distinguished development perspectives: while the classification
based on interface models discriminates between signature lists and protocols, it does not equally structure non-functional characteristics and just summarises them as quality.

The remaining shortcomings are fixed by the introduced classification schema that enhances interface models ex post. Here a explicit distinction between functional and non-functional aspects is introduced as an additional dimension which is as important as the technical view. This leads to a clearer classification of the problem space in which the important non-functional aspects can be seen as separate part of each of the technical oriented aspects. When looking at figure 5 it is obvious that this approach is therefore superior to the first two schemes. Nevertheless this approach lacks a more detailed differentiation of the domain related problems which is remedied in the last approach.

The *Unified Software Component Specification Framework* uses such a classification schema to determine relevant component characteristics and thus provides a systematic approach to classify component characteristics (and heterogeneities). Since the provided classification schema is founded in software engineering theory and uses well-established development perspectives and design views to classify component characteristics, the UnSCom framework provides a matured basis that, moreover, includes all of the component characteristics that were identified by the pre-mentioned classification approaches. That fact can be clearly seen in figure 5.

<table>
<thead>
<tr>
<th>Functional</th>
<th>UnSCom</th>
<th>Enhanced Interface Model</th>
<th>Interface model (Beugnurd)</th>
<th>Linguistic Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Information Objects</td>
<td>Domain Objects</td>
<td></td>
<td>Semantic</td>
</tr>
<tr>
<td>Operational</td>
<td>Functions</td>
<td></td>
<td></td>
<td>Pragmatic</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Processes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Architectural | | | | |
| Static | Types, Attributes | Signatures | Syntax, Synchronisation | Syntax |
| Operational | Events, Methods, etc. | | | |
| Dynamic | Protocols | Protocols | Behavior | Semantic |

| Implementation | | | | |
| Static | Usability, Portability, etc. | Interface related QoS | QoS |
| Operational | Functionality | Domain Constraints | |
| Dynamical | Efficiency | Methods related QoS | QoS |

**Fig. 5.** Comparison of the introduced Schemes.

For these reasons, we recommend this schema to be used as a basis to classify component interoperability problems and for an analysis of code generators that solve adaptation problems. Regardless there might be situations in which the other approaches might render useful for reasons of simplicity. Especially the enhanced interface model approach might be useful when talking about more technical problems and neglecting domain related aspects because of the often inherent impossibility to solve domain related problems solely with generators.

## 5 Related Work

Component based software engineering was proposed already in 1968 [24]. Nevertheless, the focus on systematic adaptation of components in order to bridge interoperability
problems is still a field of research. Most papers are based on the work done by Yellin and Strom [25,26] who introduced an algorithm for the (semi-)automatic generation of adaptors using protocol information and an external adaptor specification. Canal et. al propose the use of some kind of process calculus to enhance this process and generate adaptors using PROLOG [27,28]. Schmidt and Reussner present adaptors for merging and splitting interface protocols and for a certain class of protocol interoperability problems [29]. Besides adapter generation, Reussner’s parameterised contracts also represent a mechanism for automated component adaptation [30].

An overview on adaptation mechanisms including non-automated approaches can be found in [31,32] (such as delegation, wrappers [33], superimposition [34], metaprogramming (e.g., [35])). Bosch [31] also provides a general discussion on requirements to component adaptation mechanisms. He lists properties, such as compositionality, configurability, reusability, efficiency, power and transparency. Although these properties classify adaptation mechanisms, they are not designed and too general for classifying interoperability errors.

There are very few classification approaches specific for interoperability issues, similar to the one presented here. A classification schema structuring interaction incompatibilities is proposed in [36]. It is based on two dimensions. The first one differentiates between syntactic mismatches - similar to the signature layer of the model from section 2.2 - and several fine grained semantical mismatches. On the other dimension there is a distinction between the system itself and its environment, both with respect to the software and hardware interacting. The hierarchical classification presented in section 2.2 was a result of discussions at the object interoperability workshop on ECOOP 1999 and 2000 [7,6].

A different approach is based on identifying so called Problematic Architecture Interactions (PAI) [37]. Those PAIs are introduced as interoperability conflicts which can be identified by the comparison of architectural characteristics. Several possible PAIs are introduced and explained in [37]. It is also explicitly mentioned that the availability of such a classification can be used to reuse experience in order to solve similar architectural mismatches of the same kind.

6 Conclusions

Interoperability problems between components hinder component reuse. The lack of systematic approaches to identify and deal with component interoperability results in a higher difficulty when assembling a system from pre-produced components. An important step to systematically tackle the difficulties is to classify the interoperability problems.

Therefore we presented in this paper a survey of existing classification schemes for classifying component interoperability problems. Further on, enhancements for the existing schemes have been introduced. In particular, an improved variant of the interface model schema and the research project leading to the UnSCom framework were presented in detail.

The use of the classification schemes has been demonstrated by a lecture hall reservation system. The example shows the importance of checking interoperability considering
several different aspects as there might be a lot of different problems. Especially those problems which are not easily detectable like domain related mismatches must be inspected carefully. Only if all aspects of a component’s interface are compatible, there will be no interoperability problems.

The comparison between the classification schemes resulted in a recommendation for the UnSCom framework as that schema supersedes the other approaches and is capable of differentiating domain related problems in greater detail. If it is sufficient to abstract from domain related questions, it is also acceptable to use the enhanced interface model.

The need for a classification schema resulted from deeper research in the generation of software component adaptors. The generation process in mind might be semi- or fully automated where signature or protocol related problems are already solved with generators today. On the other hand it is doubtful if it will ever be possible to solve domain related problems automatically. The solution of quality oriented problems by automatically generated code is currently investigated.

Adaptor generation is desirable for several reasons. It can be assumed the code generation produces more performant and reliable code. Further on, there is the additional advantage that the generated code can be analysed in advance. In particular, when dealing with QoS of a system assembled from components, hand-written adaptors often screw up predictions about the assemblies QoS attributes. By using generated adaptor code, we aim at including the effect of adaptors explicitly in QoS prediction models by utilizing knowledge on the generated code. The goal is to speed up the solution process for prediction models and to get more realistic predictions.

References

10. Sun Microsystems Corp.: The JAVA homepage. (http://java.sun.com/)
33. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA (1995)
Correct Components Assembly for a Product Data Management Cooperative System

Massimo Tivoli¹, Paola Inverardi¹, Valentina Presutti², Alessandro Forghieri³, and Maurizio Sebastianis³

¹ University of L’Aquila
Dip. Informatica
via Vetoio 1, 67100 L’Aquila
{inverard,tivoli}@di.univaq.it
fax: +390862433057
² University of Bologna
Dip. Informatica
Mura Anteo Zamboni 7, 40127 Bologna
presutti@cs.unibo.it
fax: +390512094510
³ Think3 company
via Ronzani 7/29, 40033 Bologna
{alessandro.forghieri,maurizio.sebastianis}@think3.com
fax: +39051597120

Abstract. In this paper we report on a case study of correct automatic assembly of software components. We show the application of our tool (called Synthesis) for correct components assembly to a software system in the area of CSCW (Computer Supported Cooperative Work). More specifically we consider a product data management (PDM) cooperative system which has been developed by the company Think3 in Bologna, ITALY (www.think3.com). In the area of CSCW, the automatic enforcing of desired interactions among the components forming the system requires the ability to properly manage the dynamic interactions of the components. Moreover once a customer acquires a CSCW system, the vendor of the CSCW system has to spend many further resources in order to integrate the CSCW system with the client applications used by the customer organization. Thus the full automation of the phase of integration code development has a great influence for a good setting of a CSCW system on the market. We present the application of our approach and we describe our experience in automatic derivation of the code which integrates the components forming the PDM cooperative system above mentioned. The case study we treat in this paper represent the first attempt to, successfully, apply Synthesis in real-scale contexts.

1 Introduction

Correct automatic assembly of software components is an important issue in CBSE (Component Based Software Engineering). Integrating a system with...
Correct Components Assembly

85

reusable software components or with COTS (Commercial-Off-The-Shelf) components introduces a set of problems. One of the main problems is related to the ability to properly manage the dynamic interactions of the components. In the area of CSCW (Computer Supported Cooperative Work) [5,10,9], the management of dynamic interactions of the components forming the application can become very complex in order to prevent and avoid undesired interactions. A CSCW application constitutes an integrated environment formed by one or more CSCW servers and many CSCW clients [10,9]. In general, both servers and clients are heterogeneous components built by different organizations of software development. CSCW servers are black-box components providing the main functionalities concerning a cooperative activity (e.g. repository management functionalities as data check-in and check-out, data persistence, concurrent accesses to data, group-aware information management, etc.). CSCW clients are black-box and third-party components which exploit the services provided by the CSCW servers in order to execute a group-aware task. Typically, the servers are sold by the vendor of the CSCW framework. The clients are used by the customer organization of the CSCW framework which is the organization acquiring the CSCW framework on the market. A very important issue concerns the integration between the CSCW servers and CSCW clients. Depending on the customer organization and on the typology of its product manufacture, a CSCW client can be a text editor or a spreadsheet or a computer aided design (CAD) application. Given the huge diversity of applications that could be clients of a CSCW server, it is worthwhile noticing that a CSCW server has to be integrated with a CSCW client by following ad-hoc strategies. This means that once the customer acquires the servers forming the CSCW framework, the vendor will have to implement the code integrating the clients with the servers. Moreover the vendor will have to repeat this heavy phase of deployment of the CSCW framework for each different customer. Thus an issue of great influence on the good setting of the vendor on the market concerns the full automation of the phase of integration code development.

Our approach to the integration problem in the CBSE setting is to compose systems by assuming a well defined architectural style [8,6,11,12] in such a way that it is possible to detect and to fix integration anomalies. Moreover we assume that a high level specification of the desired assembled system is available and that a precise definition of the coordination properties to satisfy exists. With these assumptions we are able to automatically derive the assembly code for a set of components so that, if possible, a properties-satisfying system is obtained (i.e. the integration failure-free version of the system). The assembly code implements an explicit software connector which mediates all interactions among the system components as a new component to be inserted in the composed system. The connector can then be analyzed and modified in such a way that the coordination (i.e. functional) properties of the composed system are satisfied. Depending on the kind of property, the analysis of the connector is enough to obtain a property satisfying version of the system. Otherwise, the property is due to some component internal behavior and cannot be fixed without di-
rectly operating on the component code. In a component based setting in which we are assuming black-boxes components, this is the best we can expect to do. We assume that components behavior is only partially and indirectly specified by using bMSC (basic Message Sequence Charts) and HMSC (High level MSC) specifications [2] of the desired assembled system and we address behavioral properties of the assembly code together with different recovery strategies. The behavioral properties we deal with are the deadlock freeness property [7] and generic coordination policies of the components interaction behavior [8].

In this paper, by exploiting our approach to correct and automatic components assembly [8,6,11,12], we describe our experience in automatic derivation of the assembly code for a set of components forming a product data management (PDM) cooperative system. The PDM system we refer to has been developed by Think3 company [1] in Bologna, ITALY.

The paper is organized as follows. Section 2 briefly describes our tool for correct and automatic components assembly called Synthesis. Synthesis implements our theoretical approach to correct and automatic components assembly presented in [8,6,11,12]. Section 3 describes a realistic application of our tool to the PDM cooperative system ThinkTeam. Section 4 concludes and discusses future work.

2 Synthesis: A Tool for Correct and Automatic Components Assembly

In this section we describe our tool for correct and automatic components assembly called Synthesis. For the purposes of this paper, we briefly recall the theory underlying our approach to correct and automatic components assembly implemented in Synthesis. For a formal description of the whole approach we refer to [8].

2.1 The Reference Architectural Style

The architectural style Synthesis refers to, called Connector Based Architecture (CBA), consists of components and connectors which define a notion of top and bottom. The top (bottom) of a component may be connected to the bottom (top) of a single connector. Components can only communicate via connectors. Direct connection between connectors is disallowed. Components communicate synchronously by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. A top-domain of a component or of a connector is the set of requests sent upward and of received notifications. Instead a bottom-domain is the set of received requests and of notifications sent downward. Connectors are responsible for the routing of messages and they exhibit a strictly sequential input-output behavior\(^1\). The CBA style is a generic layered style. Since it is always possible to decompose a

\(^1\) Each input action is strictly followed by the corresponding output action.
n-layered CBA system in n single-layered CBA systems, in the following of this paper we will only deal with single layered systems. Refer to [8] for a description of the above cited decomposition.

2.2 Configuration Formalization

Synthesis refers to two different ways of composing a system. The first one is called Connector Free Architecture (CFA) and is defined as a set of components directly connected in a synchronous way (i.e. without a connector). The second one is called Connector Based Architecture (CBA) and is defined as a set of components directly connected in a synchronous way to one or more connectors. Components and system behaviors are modelled as Labelled Transition Systems (LTS). Synthesis derives these LTS descriptions from “HMSC (High level Message Sequence Charts)” and “bMSC (basic Message Sequence Charts)” [2] specifications of the system to be assembled [8]. This derivation step is performed by applying a suitable version of the translation algorithm from bMSCs and HMSCs to LTS (Labelled Transition Systems) presented in [14]. HMSC and bMSC specifications are common practice in real-scale contexts thus LTL can merely be regarded as internal to the Synthesis specification language.

2.3 Synthesis at Work

Synthesis aims at solving the following problem: given a CFA system T for a set of black-box interacting components, C1, and a set of coordination policies P automatically derive the corresponding CBA system V which implements every policy in P.

The CBA system V is obtained by automatically deriving the connector assembling the components forming the CFA system T. This connector coordinates the assembled components interactions by following the behaviors corresponding to every coordination policy in P. In order to automatically derive the code implementing the connector in CBA system, Synthesis refers to a specific development platform. This platform is Microsoft COM with ATL [13]. This means that the connector will be derived by generating the ATL code implementing the COM server corresponding to it. Synthesis assumes that a specification of the system to be assembled is provided in terms of bMSCs and HMSCs specification. By referring to the COM framework [13], Synthesis also assumes that a specification of components interfaces and type libraries is given in terms of Microsoft Interface Definition Language (MIDL) and binary type library (.tlb) files respectively. Moreover it assumes that a specification of the coordination policies to be enforced exists in terms of Linear-time Temporal Logic (LTL) formulas or directly in terms of Büchi automata² [4]. With these assumptions Synthesis is able to automatically derive the assembly code for the components forming the

² A Büchi automata is an operational description of a LTL formula. It represents all the system behaviors satisfying the corresponding formula.
specified software system. This code implements the connector component. *Synthesis* implements the connector component in such a way that all possible interactions among components only follow the behaviors corresponding to the specified coordination policies.

Figure 1 shows the input and output data performed by *Synthesis*.

The method performed by *Synthesis* proceeds in three steps as illustrated in Figure 2.

The first step builds a connector (i.e. a coordinator) following the CBA style constraints. The second step performs the concurrency conflicts (i.e. deadlocks) detection and recovery process. Finally, by exploiting the usual automata-based model checking approach [4], the third step performs the enforcing of the specified coordination policies against the model for the conflict-free connector and then synthesizes the model of the coordination policy-satisfying connector. From the latter we can derive the code implementing the coordinator component which is by construction correct with respect to the coordination policies.

Note that although in principle we could carry on the three steps together we decided to keep them separate. This has been done to support internal data structures traceability.
3 The PDM System ThinkTeam

In this section, we use our tool described in Section 2 to automatically derive the code integrating components forming a PDM system. The PDM system we consider has been developed by Think3 company [1] in Bologna, ITALY. This system is called ThinkTeam. ThinkTeam has been developed by using Microsoft Component Object Model (COM) [13] with Active Template Library (ATL) in Microsoft Visual Studio development environment. ThinkTeam is a PDM solution that provides a solid platform for a successful product life-cycle management implementation. For engineering departments, ThinkTeam provides the data and document management capabilities required to manage all product documentation, including 3D models, 2D drawings, specifications, analysis and test results. Multiple users will always have access to updated, released and work in progress product information. Also provided is a changes management solution that enables engineers to interface and communicate with the rest of the organization. ThinkTeam is packaged into five modules to provide specific capabilities and solution features. For the purposes of this paper the module we are interested on is the ThinkTeam client (TTClient) component. This is a stand-alone application that is integrated into a CAD application and Microsoft Office applications and provides features to manage documents, versions, data attributes, and relationships among documents. We are interested in applying our approach in order to automatically derive the integration code assembling the TTClient component with the distributed instances of the third-party CAD application. This code is derived to force the composed system to satisfy the coordination policies that will be described later.

3.1 ThinkTeam Architecture

In Figure 3 we show a ThinkTeam network.

The following are the interacting components in a ThinkTeam network:

– the TTClient component which provides general purposes PDM functionalities such as documents, multiple users, and changes management, ver-
sions controlling, data attributes and relationships among documents management. *ThinkTeam* owns the documents and metadata flow and mediates between the applications and the stores. The backing store is the *RDBMS* component;

- the distributed applications used by the *ThinkTeam*’s customer organization. Depending on the kind of customer organization manufacture, these applications can be either a CAD application or a text editor application or any other kind of application managing the product data;

- a centralized repository (i.e. the *Vault*) for the documents related to the products of the customer organization and for the relationships among hierarchical documents. A hierarchical document can be either a document with all information itself contained or a document making use of references to other hierarchical documents. *Vault* operations pertain to document data and allow reservation (i.e. check-out), publishing (i.e. check-in) and unrestricted read access. The backing store is a filesystem-like entity. Actually an entity corresponds to a file into *Vault*.

- a centralized repository (i.e. the *RDBMS*) for the metadata.

In a *ThinkTeam* network, the *TTClient* component manages many types of data related to documents, to the work flow, to product parts and to the organization. All these data are classified in terms of *entity* types and their attributes. Thus an *entity* represents any data which has an associated set of attributes.

### 3.2 ThinkTeam/Application Integration Schema

As showed in Figure 3, the distributed applications used by the customer share an instance of the *TTClient* component. One of the goals of Think3 company is to automatically derive the code integrating the *TTClient* component with a particular application which, in our case study, is a CAD system. This is an important goal for the company because an automatic and correct integration of *TTClient* with the customer application makes the *ThinkTeam* system more competitive on the market. In Figure 4, we show the integration schema for *TTClient* component and the CAD system used by the customer organization.

![Fig. 4. Integration between ThinkTeam and the customer’s application.](image-url)
By referring to Sections 2.1 and 2.2, the integration schema of Figure 4 represents the CBA version of the system we are considering for the case study of this paper. On the left side we show the component provided by Think3. It is the TTClient black-box component plus an auxiliary component (i.e. the integration component on the TTClient side) which has the only function to export to its clients the services provided by the TTClient component. In the following of this paper, we consider this composite component (i.e. TTClient plus the auxiliary component) as a single component called ThinkTeam component. ThinkTeam component is a black-box component which provides to its clients a specified set of services. The following is the subset of all services provided by the ThinkTeam component relevant to our purposes: 1) afterinit: this method has to be called when the application integrated with the ThinkTeam component is completely initialized; 2) checkout: locks the specified file into Vault for writing operations; 3) checkin: releases the lock activated for writing operations on the specified file into Vault; 4) get: gets a local copy of a file; 5) import: copies a local file into Vault; 6) getattrib: obtains a read only copy of the attributes of an entity into Vault; 7) setvalue: sets/modifies the value of a certain entity attribute; 8) setvalues: set/modify all entity attributes values; 9) remove: removes the entity from Vault; 10) start: starts-up the integration between ThinkTeam and the application used by the customer; 11) stop: shuts-down the integration between ThinkTeam and the application used by the customer. On the right side of Figure 4 we show the application used by the customer organization. In our case the customer’s application is a CAD system and the ThinkTeam component has to be integrated into it. In the following of this paper, we refer to the customer’s application as CAD component. The CAD component is a black-box component which provides to its clients the following services: 1) ttready: this method has to be called when the ThinkTeam component integrated into the customer’s application is completely initialized; 2) openfile: opens a file; 3) save: saves the changes made on a file; 4) closefile: closes a file. Between ThinkTeam and CAD components we show the connector component whose aim is to integrate the ThinkTeam component and the CAD component. The connector mediates the interaction between ThinkTeam and the distributed instances of CAD by following specified coordination policies. We use Synthesis to automatically derive the code implementing the connector component. As described in Section 2.3, we start from the following input data: i) a bMSCs and HMSCs specification of the CFA version of the system to be assembled; ii) MIDL and Type Libraries files for ThinkTeam and CAD components; iii) a Büchi automata specification of the desired coordination policies. In Section 3.3, we show the application of Synthesis to the automatic and correct integration of the ThinkTeam and CAD components.

3.3 Synthesis for Integrating ThinkTeam and CAD

In this section we apply our tool Synthesis in order to automatically derive the code of the connector component showed in Figure 4. This code is derived in order to limit all possible interactions among ThinkTeam and CAD to a subset of
interactions corresponding to a set of specified and desired coordination policies. In Figure 5 we show the bMSCs representing the execution scenarios of the composed system formed by ThinkTeam (i.e. TT in Figure 5) and the CAD components.

**Fig. 5.** bMSCs specification for the CFA version of ThinkTeam/CAD system.

The scenarios in Figure 5 are defined in terms of the messages exchanged between ThinkTeam and CAD. These messages correspond to the methods provided by ThinkTeam and CAD components. By referring to the methods definitions listed in Section 3.2, the scenarios of Figure 5 do not need further explanations. In Figure 6 we show the HMSCs specification of the composed system formed by ThinkTeam and the CAD components.

**Fig. 6.** HMSC specification for the CFA version of ThinkTeam/CAD system.

Informally, a HMSC is a graph where each node, except two special nodes (i.e. the starting and the final node), is a reference to a bMSC or to a sub-HMSC. An arc from a node $n_1$ to a node $n_2$ represents that the execution of the scenario corresponding to $n_2$ follows the execution of the scenario corresponding to $n_1$. The starting node is the grey triangle with the downward arrow. Conversely, the final node is the grey triangle with the upward arrow. An arc into a HMSC from a bMSC $b_i$ to a sub-HMSC $h_j$, means that the system’s execution goes from $b_i$ to all bMSCs $b_k^j$ reachable in one step from the starting node of $h_j$. An arc into a HMSC from a sub-HMSC $h_j$ to a bMSC $b_i$ means that the system’s execution goes from all bMSCs $b_k^j$ reaching in one step the final node of $h_j$ to $b_i$. The HMSC of Figure 6 is defined in terms of three sub-HMSCs (i.e. H_WRITE, H_READ and H_ADD_REMOVE). In all HMSCs we have showed in Figure 6, each bMSC
is reachable from every other bMSC into the HMSC. For the sake of brevity, we do not show the MIDL files for ThinkTeam and the CAD components. This is not a limitation for the understanding of the paper. Actually by referring to Section 3.2, we can consider as MIDL files for ThinkTeam and CAD the two set of services provided by ThinkTeam and CAD respectively. The .tlb files for ThinkTeam and CAD are binary files and they are used internally to Synthesis. In addition to the bMSCs and HMSCs specification plus the MIDL and .tlb files, Synthesis needs to know how many instances of ThinkTeam and CAD components have to be considered. This information is provided by interacting with a dialog control of the Synthesis’s user interface. In our case study we consider two instances of the CAD component sharing an instance of the ThinkTeam component. From this additional information (i.e. components instances) and from the two input data considered above (i.e. i) bMSCs and HMSCs specification and ii) MIDL + .tlb files), Synthesis is able to automatically derive a graph representing the component’s behavior for each component’s instance forming the specified composed system. This graph is called AC-Graph. In order to automatically derive these AC-Graphs from the bMSCs and HMSCs specification, Synthesis executes our implementation of the algorithm developed in [14]. Figure 7 is a screen-shot of Synthesis in which we show the automatically derived AC-Graph for the instance of the ThinkTeam component.

Refer to [8] for a formal definition of AC-Graph. Informally, an AC-Graph describes the behavior of a component instance in terms of the messages (seen as input and output actions) exchanged with its environment (i.e. all the others components instances in parallel). Each node is a state of the behavior of the component’s instance. The node with the incoming arrow (i.e. S1721 in Figure 7) is the starting state. An arc from a node \( n_1 \) to a node \( n_2 \) is a transition from \( n_1 \) to \( n_2 \). The transitions labels prefixed by “!” denote output actions, while the transitions labels prefixed by “?” denote input actions. For the sake of brevity we do not show the AC-Graph for the two instances of the CAD component (i.e. C1 and C2432 on the left panel of the Synthesis’s user interface in Figure 7). The last input data for Synthesis is the Büchi automata specification of the coordination policies to be enforced on the composed system through the automatically synthesized connector component. The coordination policy we want to enforce in our case study is the following: “a document cannot be removed if someone has checked it out. Moreover, the attributes cannot be modified if someone is getting a copy of the entity as a reference model.”. Figure 8 is a screen-shot of Synthesis in which we show the provided automaton for the above coordination policy.

Informally, the automaton in Figure 8 describes a set of desired behaviors for the composed system formed by the ThinkTeam’s instance and the two CAD’s instances in parallel under the point of view of a hypothetical observer. Each node is a state of the observed composed system. The node with the incoming arrow is the initial state. The black nodes are the accepting states. Once the automaton execution reaches an accepting state, it restarts from the initial state.
Fig. 7. Synthesis screen-shot of an instance of ThinkTeam component.

Input\(^3\) transition labels are prefixed by “?”, instead output\(^4\) transition labels are prefixed by “!”. Each transition label (except for a particular kind of transition) is postfixed by “_” followed by a number. This number is an identifier for a component’s instance. Referring to Figure 8, 1 identifies an instance of the CAD component, 2432 identifies the other instance of the CAD component and 2 identifies the instance of the ThinkTeam component. For each state \(s_i\), we can label the outgoing transitions from \(s_i\) with three kinds of action: i) a real action (e.g. \(?\text{checkout}_1\) from the initial state in Figure 8) which represents the action itself, ii) a universal action (e.g. \(?\text{true}\) from the initial state in Figure 8) which represents any action different from all actions associated to the other outgoing transitions from \(s_i\) and iii) a negative action (e.g. \(?\text{-remove}_2432\) from the “S11765” state in Figure 8) which represents any action different from the real action corresponding to the negative action itself (i.e. the negative action without “-”) and from all actions associated to the others outgoing transitions from \(s_i\). From the AC-Graph for each component and from the automaton of the desired coordination policy, Synthesis derives the model of the connector component

\(^3\) Input for the hypothetical observer.

\(^4\) Output for the hypothetical observer.
Fig. 8. Synthesis screen-shot of the coordination policy to be enforced.

that assembles ThinkTeam and the two CADs by implementing the coordination policy of Figure 8. For the sake of brevity we do not show the model of the connector synthesized before the coordination policy enforcing step. Figure 9 is a screen-shot of Synthesis in which we show the model of the connector after the coordination policy enforcing step.

The sink black states identify the achievement of a desired behavior. Once the connector component’s execution achieves a desired behavior, it restarts from the initial state. From this model by exploiting the information stored in each node and arc, Synthesis automatically derives the code implementing the policy-satisfying connector. This code implements an ATL COM component. It is constituted by a MIDL file (.idl), an ATL header file (.h) and an ATL source file (.cpp). In order to produce the code implementing the COM connector component Synthesis uses also the MIDL and .tlb files for ThinkTeam and CAD provided in input. In the following we only report the meaningful parts of the implementing code of the ATL COM connector component produced by Synthesis. For the sake of brevity we do not report the MIDL code (.idl) specifying the interface of the ATL COM connector component. That interface is defined by exploiting the interfaces of ThinkTeam and CAD components. The following is the ATL header file (.h):
The class declaration for the ATL COM connector component exploits the class declarations of the *ThinkTeam* and *CAD* components. The ATL COM
connector component encapsulates references to ThinkTeam and CAD objects and uses a set of private members in order to identify a caller of a service and to store the state reached during the execution. This is needed in order to reflect the behavior of the connector model in Figure 9. The following is the ATL source file (.cpp):

```cpp
... STDMETHODIMP TTConnector::get(...) {
    HRESULT res;
    if(sLbl == S4640_S11760)
    {
        if(chId == 1) // it corresponds to an instance of CAD
        {
            res = ttObj->get(...);
            sLbl = S6087_S12299;
            return res;
        }
        else if(chId == 2) // it corresponds to the other instance of CAD
        {
            res = ttObj->get(...);
            sLbl = S5007_S12997;
            return res;
        }
    }
    return E_HANDLE;
}
...
```

For the sake of brevity, we have only reported the code for the get connector method. It reflects the structure of the model in Figure 9. All other methods are synthesized analogously to the get method. The only difference is that while get, setvalue, setvalues, remove, checkout and checkin contain delegations of the corresponding methods on the ThinkTeam object (i.e. ttObj), the methods openfile and closefile contain delegations of the corresponding methods on the CAD object (i.e. cadObj). All remaining methods (i.e. afterinit, import, getattrib, start, stop, ttready and save) are synthesized as simple delegations toward the object (i.e. ttObj or cadObj) which provides them.

4 Conclusion and Future Work

In this paper we have applied the tool Synthesis implementing our connector-based architectural approach to component assembly to integrate a PDM system into a CAD application. Synthesis focusses on enforcing coordination policies on the interaction behavior of the components constituting the system to be assembled.

A key role is played by the software architecture structure since it allows all the interactions among components to be explicitly routed through a synthesized connector. By imposing this software architecture structure on the composed system we isolate the components interaction behavior in a new component (i.e. the synthesized connector) to be inserted into the composed system. By acting on the connector we have two effects: i) the components interaction behavior can satisfies the properties specified for the composed system and ii) the global system becomes flexible with respect to specified coordination policies.

Synthesis requires a bMSC and HMSC specification of the system to be assembled. Since these kinds of specifications are common practice in real-scale
contexts, this is an acceptable assumption. Moreover we assumed to have a LTL or directly a Büchi automata specification of the coordination policies to be enforced.

By referring to the case study described in Section 3, the main advantage in applying Synthesis is the full automation of the integration phase of ThinkTeam into CAD application used by the customer organization. This reduces the cost and the effort needed for the ThinkTeam component deployment phase making ThinkTeam more competitive on the market. The code of the synthesized ATL COM connector component could be not fully optimized. Think3 can modify by hand the synthesized code in order to apply all the requested optimizations. This is obviously better than write the whole adaptation code by hand.

Limits of the current version of Synthesis are: i) Synthesis completely centralizes the connector logic and it provides a strategy for the connector source code derivation step that derives a centralized implementation of the connector component. We do not think this is a real limit because even if the connector logic is centralized we are working on a new version of Synthesis which derives a distributed implementation of the connector component if needed; ii) Synthesis assumes that a HMSC and bMSC specification for the system to be assembled is provided. It is interesting to investigate the usage into Synthesis of UML2 Interaction Overview Diagrams and Sequence Diagrams [3] instead of HMSCs and bMSCs respectively. This aspect would improve the applicability in real-scale contexts of the tool; iii) Synthesis assumes also an LTL (or directly a Büchi automata) specification for the coordination policy to be enforced. We are currently working also in this area trying to find more user-friendly coordination policy specifications; for example by extending the HMSC and bMSC notations to express more complex system's components interaction behaviors.

Acknowledgements

This work has been supported by Progetto MURST CNR-SP4.

References

2. Itu telecommunication standardisation sector, itu-t recommendation z.120. message sequence charts. (msc'96). Geneva 1996.


The Release Matrix
for Component-Based Software Systems

Louis J.M. Taborda

Macquarie Graduate School of Management,
Macquarie University, NSW 2109, Australia
louis.taborda@procentric.com

Abstract. The challenge of managing the development and evolution of complex, component-based software is increasingly being recognized as the development of such systems becomes more common. This paper attempts to understand the relevance of current management best practices by utilizing a simple meta-model to illustrate the impact that architectural complexity and reusable components have on management patterns. The model serves as a heuristic device and supports the view that products based on a framework of reusable software components pose new challenges and have to be managed simultaneously at a number of different levels. This provides a rationale for the Release Matrix, a generalization of a software release plan, previously proposed as a technique for managing software product lines. The Release Matrix has practical applications for tracking the evolution of complex component-based systems and is shown via the model to be a natural consequence of increased architectural complexity and component reuse. This paper has relevance for developers seeking simple techniques to help them manage challenging component-based programs, as well as researchers interested in the conceptual basis and limits of current management practices.

1 Introduction

Software intensive systems continue to grow in complexity and while component-based architectures provide benefits in terms of productivity and quality; they also represent a growing management challenge. Beyond traditional software development expertise necessary to assure component quality, component-based development (CBD) must address new challenges including component selection, reuse, assembly, as well as the integration testing and evolution of a configuration of inter-dependent components [1]. These challenges are markedly different from one-off systems development and new methods, and techniques are needed to tackle the management of such componentized systems [2, 3, 4].

A significant benefit offered by CBD is the potential to reuse components across a number of products (alternatively, applications or systems depending upon the terminology preferred). CBD methods, like their object-oriented predecessors, encourage the design of components for future reuse and means of easily identifying and utilizing
these components in the construction of software systems is an area of active research [3, 5]. This body of work increasingly recognizes that the success of CBD and product line architectures (PLA) are intimately connected to the management of a configuration [4, 5, 6], and hence renewed interest in the discipline of Configuration Management (CM) applied at an architectural or coarse-grained, component or sub-systems level [7, 8] rather than at the software source level. The emphasis of this paper is on the management of configurations of such coarse-grained components that may subsequently be decomposed into the lower level entities that can be queried for more detailed management information.

In systems development programs, exemplified by early military and aerospace initiatives, a complex system is typically decomposed into sub-systems in a manner that has a close, but imperfect correspondence with CBD where decomposition is not the only means of identifying components. In that domain, the Systems Engineering discipline [9] ensures that the sub-systems integrate into the desired system. For all the formalism that supports such systems development programs, managing CBD can be shown to be inherently more complex. This is due to the fact that, as well as the system integration and program coordination challenges of complex systems development, the reuse of components adds a new level of management complexity. One where there is not a single customer to be satisfied, and thus the demands of the separate products reusing the component has to be juggled.

Traditional project management (PM) techniques, such as Work Breakdown Structure (WBS), the Critical Path Method (CPM), the Program Evaluation and Review Technique (PERT) and Gantt charts, were not conceived to address the significant interdependencies that often make planning a product development in isolation irrelevant. The release of a product is therefore no longer a single, isolated event but rather one that involves careful choreography of a series of smaller, “fractal” component releases. Providing accurate and reliable program status in such environments is extremely difficult, often causing management to become intractable and reactive.

While many CBD practitioners routinely have to grapple with these issues in the workplace, there are few management techniques available to guide their activities. This research has been motivated by such real-world conditions where the effectiveness of current management best practices may appear inadequate. In exploring practical new techniques more suited to these conditions, the meta-model presented in this paper was found valuable for describing the conditions that gave rise to “classical” management best practices and illustrating why the adoption of CBD and systematic reuse necessitates new paradigms to address this new level of complexity.

The remainder of this paper is structured as follows. Section 2 presents an overview of the meta-model to study the effects of product complexity on management patterns. Section 3 investigates the relevance of the most complex pattern and its realization in modern CBD and product lines. Section 4 describes the Release Matrix and shows it to be the natural result of the interdependencies between products and components with practical applications. Section 5 concludes by reviewing the future direction of the research, as well as discussing current limitations and the validation of the model in the real world.
2 Modeling Product Complexity

In an effort to arrive at new insights into CBD, it is instructive to go back to fundamentals and review the management patterns that apply to the development of increasingly complex products. A simple meta-model acts as a heuristic device to investigate the effect of growing architectural complexity and to understand the appropriateness of different management techniques. The model itself was motivated by a desire to understand the challenges of complex software development and offers a rationale for the previously proposed Release Matrix [10].

Fig. 1. Four stages of increasing development complexity.

2.1 Monolithic Architecture

We start by considering the simplest architecture (Fig. 1a) where the one-to-one correspondence between a product and component represents the development of a monolithic software system. In such a simple scenario there is little need to distinguish between the terms product and component (better called the architecture) since they are one and the same.

---

1 An entity-relationship diagram is used to illustrate the model as standard data-modeling techniques can be applied to normalize the many-to-many relationship.
This stage represents the earliest development paradigm when the focus of all effort is a tightly coupled architecture that has only to meet the needs of a single product. As such, it offers a textbook example of early development where PM techniques would ably support the planning and tracking of activities. A single development organization is likely to have been responsible for the release of the complete product, requiring little packaging given its monolithic nature, and using CM to provide the baseline control for the delivered product.

2.2 Product Decomposition

As systems grew in complexity, the natural response was to apply the “divide and conquer” maxim to decompose the product into sub-systems or components. This results in the one-to-many relationship between products and components (Fig. 1b). Irrespective of how the decomposition of the system is achieved (whether by top-down structured methods or object-oriented techniques such as domain engineering) they have the effect of “breaking-down” the monolithic architectures of the past and enabling different components to be developed separately.

Under these conditions, the organization of the work would usually follow the architectural boundaries (which might also represent different skill-sets) so that the individual components could be developed and managed as separate (sub) projects. While these are useful devices to manage complexity, the trade-off is that the separate component development schedules now have to be coordinated so that they can be integrated into the required product. As customer requirements would in general span or be allocated to the different components, achieving a product release necessitated the coordinate of the separate component releases. This requires significant management coordination that was often possible because the component teams were dedicated (if not contractually bound) to the product’s release schedule.

In particular, aligning the separate component release schedules is essential to integrate, test and deliver the final product, but achieving this and tracking the progress of the program can be problematic. Thus, it is common for the program to be under the control of a system integrator (or prime contractor in military programs) responsible for the product or system integration, while the components that are not commercial off the shelf (COTS) are developed autonomously by teams that can be in-house, remote, sub-contracted or out-sourced.

System Engineering techniques help address these challenges by ensuring that the separate components are specified and designed to be consistent with the overall system concept and that the customer requirements are traceable across these components. CM must provide the baseline controls to ensure that the different component releases are clearly identified and made available for integration and testing at the product level.

This pattern is manifest in a number of different real-world contexts including military development, CBD, PLAs and all manner of information systems (IS) development. While the nature of the components and their granularity may vary, the management pattern remains consistent and relevant.
2.3 Reusable Components

The potential to reuse or separately market software capability is at the heart of CBD principles. The aim is to identify and isolate such capabilities so they can be managed as separate components that may be reused by multiple products or applications (Fig. 1c). Increased complexity only serves to drive specialization, making it more likely that critical components will mature into reusable components or sub-systems. Indeed, domain engineering attempts to identify such components early so that a shared framework of reusable components and services exists for products.

There is a spectrum of possible reuse, and COTS products could be seen as the extreme case that best represents this pattern. Here potentially numerous customers have to be supported and, depending upon the maturity of the component/product, this could entail maintenance patches and potential development activity. The level of “shrink-wrap” that the product has achieved will determine the degree of direct influence a customer will have on the component’s evolution or direction. COTS vendors would always hope to be responsive to market needs but reusable components developed in-house as part of (say) a domain engineering effort, can expect a different level of feedback and demand for responsiveness from its customers. The close interaction and greater flexibility afforded by in-house reuse efforts represents the other extreme of the reuse spectrum, often characterized by iterative development and short release cycles.

Irrespective of the environment, a decision by a product group to reuse a common component represents a significant change in the balance of power within a program. The fact that there are multiple customers for a reusable component means that it is elevated beyond the control of any single product group and now must act as a separately managed entity responsible for its own destiny. Thus, the reused component (and its development group) must introduce mechanisms for the collection, prioritization and conflict resolution across its different customers. The release planning and scheduling can become fractious as they directly affect the product plans and capabilities. No single product can dictate the reused component’s evolution or release plan as they have to satisfy the demands of several product markets concurrently, or its reusability (or market) is threatened. The products that use a common component must therefore share influence and must negotiate with the component team to achieve an acceptable component release plan.

A product’s loss of control has to be traded-off against the benefits of utilizing a reusable component that can include shared development and maintenance costs, improvements in quality, the shrinking of schedules and the reduction of risk. Each product group must carefully weigh these considerations before making the decision of whether to utilize a reusable component. In the case of a COTS component, this corresponds to the classic build vs. buy decision; whereas for in-house components this relates to the decision to identify reusable components instead of recreating software capabilities.
2.4 Concurrent and Systematic Reuse

The ultimate challenge in our study of increasing product complexity is the many-to-
many relationship between products and components (Fig. 1d). This represents the
concurrent demands of multiple products impacting a configuration of reusable com-
ponents; exemplified by domain engineering or product line architectures. The critical
differentiator between this relationship and the “simpler” product decomposition pat-
tern is that every reuse of a common component creates an implicit coupling between
two products using it. What were independent products can now have their plans and
schedules intertwined as a result of decisions made regarding the reusable components
they share. This situation is illustrated in the Release Planning scenario described later
in this paper.

Such a highly interdependent environment creates a challenge for all stakeholders
in the enterprise. As described before, each product can place competing and poten-
tially conflicting demands upon the reused components. Component development
teams have to plan releases as best they can and also balance the priorities across the
different products they serve. Product managers, on the other hand, may have re-
quirements that necessitate modification to multiple components and so have to nego-
tiate with each of these groups to ensure that they can meet the product schedule. As
new agreements are reached, however, they have to be reviewed and approved by the
other product and component stakeholder groups that may be affected.

![Diagram of Products and Components](image)

*Fig. 2. Products use and reuse of components.*

The interdependencies inherent in this pattern can therefore be seen to be somewhat
outside the realm of traditional management disciplines. PM and CM techniques can-
not be assumed to be applicable under these conditions which are significantly different to those they were designed to address. Specifically, planning cannot be conducted for any single component or product in isolation. Attempting to arbitrarily favor one stakeholder over another can make reuse unsustainable and subject to the observed tensions [11]. Instead, an acceptable master plan for the entire enterprise has to be negotiated that meets the enterprise’s business objectives and encompasses the individual plans for the component and product groups.

With this level of interdependency, no single product or component can plan its release schedule in isolation. The multiplicity of conflicting demands and priorities can only be resolved at the business, or enterprise level. All stakeholders in the many-to-many relationship have to be represented and involved in the release planning process and must accept any compromises necessary as a result of agreed business priorities. A necessary feature of this planning is that it has to be done holistically, so that a master plan for the enterprise is developed with each individual stakeholder’s plan synchronized and compatible with the whole. Achieving this, however, is no simple task. Too often in such complex environments it can be expected that releases will address the needs of the most dominant group or individual rather than the business priorities.

3 Implications of Product Complexity

The last stage of the model described is the most general case and encompasses all previous “stages of evolution” of software architectures. Stepping through the different stages of this heuristic device can be viewed as an historical tour of the increasing architectural complexity and its associated management patterns. The research question that this poses is: How does reality fit the model? This topic is one that is worthy of further investigation, however, the fact is that current system complexity can be shown to have reached the most complex level in the model – without the benefit of correspondingly sophisticated management techniques.

3.1 Pattern Recognition in PLAs

The many-to-many relationship between products and components in complex software architecture are illustrated in Fig. 2. Such a relationship is readily recognizable in PLAs [3, 10] and it has been proposed that it be elevated to the status of a management pattern. This has been termed the “Marketplace Pattern” [10] in recognition of the number of potential stakeholders that could contribute to a PLA, both internal and external, to an organization.

While further research is necessary to determine the prevalence of the Marketplace Pattern outside of PLAs (and its wider acceptance within the PLA community), there are a number of scenarios where it can be observed in practice. To recognize these we have to identify situations where the following two, necessary conditions are satisfied:
(i) A configuration of (partially) reusable components or sub-systems exists or will be brought into existence; and,
(ii) There are multiple products (or more generally, customers) that make concurrent demands impacting this configuration.

The above criteria suggest that the pattern may be identified in many complex software development environments. The increasing utilization of COTS products or components [12], such as the middleware necessary in modern CDB technologies like the .NET and Java platforms, makes this management pattern almost unavoidable. Similarly, the systematic reuse inherent in domain engineering and the resulting software frameworks can be expected to be subject to such a pattern, along with the general category of business IS architectures.

3.2 Related Management Techniques

Modern software architectures have been shown to be susceptible to the highest levels of complexity described by the heuristic model introduced in this paper. The consequence of this complexity is that it creates a challenge that, it is argued, can exceed the capabilities of traditional management disciplines like PM and CM when applied in isolation.

The study of the growing complexity the model allows suggests that more integrated approaches to managing these highly complex situations is required. These approaches must extend the “classic” single-product, single-project disciplines so that they address the growing interdependencies that characterize modern systems architectures. PM and CM need to play multi-dimensional roles in enterprises employing large-scale component reuse, and need to be applied concurrently, both at the original, single-product level they were designed for, as well as at the enterprise level.

It is increasingly recognized that the CM problem in CBD and PLA environments significantly increase [5, 7, 13], and that common CM techniques are often inadequate or have to be “stretched” to tackle the different levels at which the challenges manifest. Similarly, there have been calls for new paradigms in PM to address increasing complexity [14] and simultaneity [15] of projects. This, and the increasing interest in Agile Methods [16] that question traditional plan-driven development approaches, indicate a growing dissatisfaction with the adequacy of the classical management techniques.

Yet the situations where these disciplines fall short relate to relatively common everyday scenarios for CDB practitioners. Even minor changes to a component-based architecture can, because of reuse, impact the plans of numerous related products – and even seemingly unrelated components. The simple example below highlights this situation and indicates the need for the entire enterprise (comprising products and components) to be managed as a configuration itself so that any changes can be analyzed holistically and their ramifications understood and anticipated.
3.3 Release Planning Scenario

As a real-world example of the inter-dependencies that have to be addressed, consider this simple scenario. A Billing System has a high-priority requirement to support a more elaborate bill-print format that displays a variable customer greeting. Achieving this capability might require coordinated changes to (say) the display, storage and print-manager components. Changing just one component alone is inadequate since all three capabilities (and so components) have to be released together to achieve the desired business result.

Add to this scenario the fact that the display and print-manager components are reused by another product that has a low priority request to support a new output device. It becomes evident that whenever there is reuse there arises the possibility of conflicting priorities that have to be managed. Thus the question: with limited resources, which requirement should the print-manager component team work on first? Irrespective of the apparent urgency, it may be that longer timescales for changing the display and storage components make it pointless for the print-manager component team to schedule the corresponding formatting changes with the same priority. In that case the smarter thing for the team to do would be to work on support of the new output device first. After that task is completed, then the formatting changes can be implemented, still allowing the synchronized release of the new bill-print format at the earliest possible time.

Juggling the priorities and balancing the competing concerns across products and components is what release planning in a product line environment is primarily about. To manage limited resources optimally requires the ability to easily recognize and adjust for the inter-dependencies between the change activities and the components that they affect.

4 Holistic Management

The previous scenario highlights the fact that a web of dependencies, not always recognized or managed, joins all products based upon a shared software framework. Management of these relationships has been shown to be a difficult exercise that can lead to problems [17]. The argument has been made that dependencies in software need to be separately managed [18], and the explicit recording of the component relationships is a key requirement of software release management [7]. But component releases must also be kept consistent so that changes do not render them incompatible with each other. This requires that release planning and management take place holistically, at the enterprise-level, taking into consideration the entire architecture and all the product and component stakeholders involved.

The challenge of managing complex, product-line architectures both at the individual component or product level as well as at the enterprise-level, gave rise to the concept of a Release Matrix that has been introduced as the multi-dimensional generalization of traditional release plans [10] and is summarized below.
4.1 The Release Matrix

In situations where concurrent product development is based upon a reusable, component-based framework, there are two distinct, orthogonal management views that can be taken of development – the first based on the products and the second based on the components. A matrix representation has been proposed to capture these two viewpoints, thus explicitly identifying the product and component stakeholders so that their perspectives and interdependencies can be clarified. These matrices can be seen to be a manifestation of the most general, many-to-many case of the meta-model described, and as such, provide an organizing principle to help consider different facets of a complex software program. The matrices offer the ability to record relevant, lifecycle information in their cells, while capturing the relationships that information has with other members of the ecosystem. This traceability provides a context that can support the incremental planning and evolution of the component-based architecture by early identification of the possible impacts of a change.

In particular, the Release Matrix has been proposed as a means of planning and tracking the evolution of the system architecture over time. As shown in Fig. 3, the Release Matrix records the components (x-axis) and the products (y-axis) that use these components, integrating them into the market offering. The matrix can be seen to correspond to the relationships shown in Fig. 2 where the existence of a relationship between a product (P<sub>i</sub>) and component (C<sub>j</sub>) results in an entry in the intersecting cell (r<sub>ij</sub>). When no relationship exists between a product and component there is a zero or null entry in the corresponding cell.

The content of the cells of a Release Matrix can be simply regarded as the scheduled dates of the set of dependent releases, however a family of similar matrices can be employed to record different lifecycle data depending upon the utilization of the matrix. For example, in order to derive and coordinate the release schedules for all products and components, a separate matrix can be used to record the product requirements that have been allocated to the different components. With reference to the release planning scenario previously described, the P2 row could represent the bill-print project where C2, C3 and C4 would be the display, storage and print-management components.

![Fig. 3. The Release Matrix consolidates the product and component perspectives.](image-url)
The multiplicity of releases that are a feature of CBD environments can benefit from the clarification offered by the Release Matrix. It provides a succinct and explicit means of recording the dependencies between products and components, while supporting the “separation of concerns” principle by extricating the two perspectives. Each row of the Release Matrix represents a product’s release plan that is derived from, and must be compatible with the component releases that the product is reliant upon. Similarly, each column represents a component’s release plan, based upon the total set of product requirements that are to be implemented in the release. As a whole, the Release Matrix represents a master release plan that consolidates and synchronizes the individual release plans of both the products and the components.

By way of example, the highlighted column in Fig. 3 represents the perspective of the team responsible for component C3 that needs to balance the demands of products P2 and P3. The highlighted row corresponds to the reliance that project P2 has on three components, C2, C3 and C4, that may need to have coordinated releases to effect a business change. The intersection of these two perspectives indicates the specific plan or contract between the P2 and C3 teams. Similar plans must be negotiated for all non-null cells as they point to a dependency between the stakeholders that requires coordinated attention in order to achieve an agreed and consistent set of releases.

In general, Fig. 3 shows that each product group must attempt to align the components it relies upon across the row, while each component producer must weigh and prioritize the requirements of its customers shown in that column. These orthogonal perspectives represent the different tensions that have to be balanced in a complex component-based environment.

4.2 Validating the Release Matrix

While the Release Matrix offers a simple, integrating technique to visualize and balance the competing priorities of the different stakeholders, it actually resulted from the recognition of the orthogonality of the product and component perspectives and concerns.

![Fig. 4. Normalizing the product-component relationship.](image)

The heuristic model has shown that the most complex management pattern is characterized by the many-to-many relationship between products and components. From
a data modeling perspective, this is a candidate for normalization and standard techniques for resolving such relationships leads to the introduction of a new associative entity [19]. Therefore normalization of the meta-model introduces the Release Matrix shown in Fig. 4, that represents the time-based record of the evolving software architecture. While this result could be anticipated from the existence of the many-to-many relationship, the refined model helps establish the matrix representation, and the Release Matrix, as appropriate constructs for today’s architectural complexity.

4.3 Application of the Release Matrix

The Release Matrix is proposed as a necessary generalization of an individual product’s (or component’s) release. It captures a snapshot of the evolution of complex, component-based architectures, where an adjustment to any element of the plan can have multiple impacts requiring intense communication and collaboration to achieve an agreed master plan.

The Release Matrix can therefore provide a consolidating mechanism for planning or negotiating incremental release schedules, documenting the agreements and expectations of the stakeholders involved [10]. It can be used as a visualization tool during the planning process, recording and presenting critical management information, including:

- the component releases upon which each product is reliant
- the products that are reliant upon a component’s scheduled release
- the scheduled release dates of all parties forming the matrix
- the component that is released last and so drives the schedule
- the impact of any schedule delays on other stakeholders
- the component and product versions that constitute an architectural baseline

5 Conclusion

This paper has utilized a meta-model as a heuristic device to discuss the effects of increased architectural complexity on familiar management disciplines. The model is described in stages that correspond to the growing adoption of CBD and systematic software reuse. As such, it also provides a chronology of the management patterns that have been applied to the development and release of increasingly complex software products.

In this context, current management best practices, characterized by planning and management control, can be seen to be more suited to the early, simpler stages of the model. The introduction of systematic reuse reduces the effectiveness of management control and requires, instead, greater collaboration between the stakeholders and holistic planning at the enterprise level. The aim of this exploration is to provide further rationale for the Release Matrix that has previously been proposed as new technique for managing the evolution of complex software architectures.
This research is ongoing, and the Release Matrix is presented here as a concept that requires further exploration and detailing. While the simplicity of the technique is compelling, software architectures can be notoriously complicated, with the potential for inconsistent and even circular dependencies [4] rather than the orderly, hierarchical structures described in this overview of the matrix representation. Therefore the technique should be viewed as a first-order approximation of a complex reality, and more suitable for the coordination of large-scale, component-based programs where organizational boundaries define the elements of the Release Matrix rather than the architecture.

Further study of the technique is necessary to validate the concept, determine its applicability, and adapt it as appropriate. Current research is focused on the real-world application of the Release Matrix and trials have been conducted with a case study in preparation. Investigation into the use of the matrix representation in describing the development lifecycle is also planned, with the goal of providing a more comprehensive model for complex software development.

References


Viewpoints for Specifying Component-Based Systems

Gerald Kotonya and John Hutchinson
Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{gerald,hutchinj}@comp.lancs.ac.uk

Abstract. There is a conceptual gap between the way we currently articulate requirements and the reuse-driven paradigm embodied in component-based system development. The principal challenge in requirements engineering for component-based systems is to develop models and methods that allow us make the best use of the available component technology by balancing aspects of requirements and business concerns, with the architectural assumptions and capabilities embodied in blackbox software components. This paper proposes a method for requirements engineering based on the notion of viewpoints that provides an explicit framework for expressing component-based system requirements from initial formulation through to detailed specification.

1 Introduction

Component-based system development proceeds by integrating pre-fabricated software components [3], [5], [13]. A software component is typically a blackbox implementation whose visibility is exclusively through its interface. However, because it’s subject to third party composition a component’s interfaces must be clearly specified. Traditional requirements engineering models are inappropriate for component-based system development [2], [10]. In addition, features supported by third party software components may greatly in quality and complexity. This inconsistency together with the variability in application contexts means that specifications delivered with software components are likely to be incomplete or inadequate [3]. These problems underscore the need for requirements engineering models that can balance aspects of system requirements and business concerns, with the architectural assumptions and capabilities embodied in software components.

Our proposed solution is a service-oriented requirements approach that interleaves the process of requirements with component verification, negotiation and planning.

2 Background

It is generally acknowledged that good requirements engineering is essential for successful component-based system development [12]. However, the critical problem of requirements elicitation and modelling is often ignored in specification methods for component-based systems [10]. Current approaches for requirements specification are based on procurement models in which the requirements process is driven almost exclusively by the availability of software components [12, 14]. This strong focus on
component selection ignores the dependencies and subtle interactions between requirements. It reduces the scope for requirements negotiation, makes it difficult to address quality attributes and system level concerns, and makes it difficult to assess the impact on the system of new components. Critically, it ignores the important role architecture plays in formulating requirements for component-based systems. In component-based system development, requirements definition and system design are not linear but highly interlaced activities. A common reason for this is that the system being specified may be part of an environment made up other software components. The components in the environment are likely to impose requirements on the system being specified and to constrain its design.

3 The Requirements Method

The Component-Oriented Requirements Expression (COREx) method is set in the context of a generic CBSE process [10]. Fig. 1 shows part of the process.

![Fig. 1. Component-oriented requirements engineering process.](image)

The process starts with the planning phase and iterates through development, verification and negotiation. System management cuts across the development phase, which implements the agenda set out in the planning phase. The verification and negotiation phases are intended to ensure that there is an acceptable match between selected software components and the system requirements. A colour scheme has been used to show the correspondence between the development phase and aspects of verification that apply to them. The paper is mainly concerned with the requirements stage of the system development phase.
COREx is primarily intended to support black-box development but makes allowances for custom development in cases where black-box development is not feasible. The method interleaves requirements definition with system planning, component verification and negotiation. The COREx has 3 iterative steps: requirements elicitation, ranking, and modelling.

3.1 Requirements Elicitation

All requirements methods must address the basic difficulty of identifying the problem domain entities for the system being specified. The majority of methods provide little guidance in this, relying instead on the method user’s judgement and experience. COREx uses the notion of viewpoints to support this elicitation process [8]. The notion of viewpoints has also been used in requirements engineering to support conflict management [7]. We have generalised potential requirements sources into a set of viewpoints classes that can be used as a starting point for finding viewpoints specific to the problem domain (Fig. 2). The root of the tree represents the general notion of a viewpoint. Information can be inherited by subclasses, and so global requirements are represented in the more abstract classes and inherited by subclasses.

COREx identifies the following abstract viewpoints:

- **Actor viewpoints** are analogous to clients in a client-server system. The proposed system (or required component) delivers services (functional requirements) to viewpoints, which may impose specific constraints (non-functional requirements) on them. There are two main types of Actor viewpoints:
  - **Operator viewpoints** map onto classes of users who interact with the proposed system. They represent frequent and occasional system users.
  - **Component viewpoints** correspond to software components and hardware devices that interface with the proposed system.
- **Stakeholder viewpoints** are entities that do not interact directly with the intended system but which may express an interest in the system requirements. Stakeholder viewpoints provide a mechanism for expressing critical ‘holistic’ requirements, which apply to the system as a whole.

![Fig. 2. Abstract viewpoint structure.](image-url)
A viewpoint template has the following structure:

**Viewpoint id:** <A unique viewpoint identifier>
**Type:** <Viewpoint type (e.g. operator, system, component, organisation etc)>
**Attribute:** <An optional set of data attributes for the Actor viewpoint>
**Role:** <Role of the viewpoint in the system>
**Requirements:** <Set of requirements generated by the viewpoint>
**History:** <Development history>

A requirement can be considered at different levels of abstraction to allow for scoping and ease of understanding. A requirement has the following structure:

**Requirement id:** <Requirement identifier>
**Rationale:** <Justification for requirement>
**Description:** <Natural language definition>|<Service description>|<Other>

Levels of abstraction may map the requirement description to different representations and levels of detail.

### 3.2 Requirement Ranking

COREx ranks requirements in terms of their perceived *benefit* to the user (i.e. as essential, important or useful). The output from the ranking process is a list of prioritised requirements that together with potential components and services form input to the component verification process.

### 3.3 Component Verification

In the early stages of requirements definition, verification is used as a filter for establishing the availability of candidate components and services, and to provide the project manager a rough indication of the viability of a component or service-based solution. In design verification is used to establish how well selected components and services match the desired system functionality and how compatible the component interfaces are with the designed sub-systems. Limitations of commercial software components and architectural considerations may mean that the design has to be modified (i.e. services reassigned to different components) or requirements modified. In extreme cases of incompatibility, parts of the architecture may be viewed as “place-holders” for custom development.

### 3.4 Component Selection

In COREx, component selection is achieved by formulating selection filters to match requirements to a checklist of component and service properties. Table 1 shows an example of a checklist table. The requirement(s) on the top left of the table are matched against the candidate components and services in column 2 using selection filters. For each checklist question, the specifier determines the extent to which the response is positive in relation to the component or service. A positive response scores 2. A weakly positive response or one for which there is inadequate information
scores 1. A score of 0 is given for a negative response. Filters are reusable entities with the following structure:

- **Identifier:** <Filter name>
- **Description:** <Description of filter and its effect>
- **Predicate:** <Predicate over checklist questions>

The formulation and selection of filters is likely to be influenced by the nature of the application and the business goals of an organisation. For example, if we assume that our starting set of components is $T_1$, such that: $T_1 = \{C_1, C_2, C_3, C_4, S_1\}$

We can define a filter $f_1$ such that only components that support the selected requirement, or can be configured to support it are selected. Filter $f_1$ is defined by the predicate, where $c$ represents the general component and $\text{checklist}(i)$ represents the checklist item $i$:

$$\forall c: T_1 \land (c.\text{checklist}(1) \geq 1)$$

$T_2$ represents the result of applying $f_1$ to $T_1$: $T_2 = \{C_2, C_3, C_4\}$

Filters can be combined to refine a selection. Filter $f_2$

$$\forall c: T_1 \land (c.\text{checklist}(1) = 2 \lor (c.\text{checklist}(1) = 1 \land c.\text{checklist}(7) = 2))$$

<table>
<thead>
<tr>
<th>Requirement: 1. Requirement_xyz</th>
<th>Checklist</th>
<th>Components/Services</th>
<th>Related Aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td><strong>Question</strong></td>
<td><strong>C_1</strong></td>
<td><strong>C_2</strong></td>
</tr>
<tr>
<td>1</td>
<td>Does component/service support requirement? Yes explicitly = 2; Not explicitly, but be configured to support requirement = 1; Don’t know/does not support feature = 0;</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Is the component/service specification provided? Yes, detailed = 2; Yes, limited = 1; No = 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Are release notes provided? Yes, detailed = 2; Yes, limited = 1; No = 0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Are installation notes provided? Yes, detailed = 2; Yes, limited = 1; No = 0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Is component/service available for evaluation? Yes, full functionality = 2; Yes, restricted functionality = 1; No = 0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Is component/service certified? Yes, independent certification = 2; Yes, local certification = 1; No = 0;</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Is help desk support available? Yes, continuous = 2; Yes, limited = 1; No = 0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>What is vendor’s market share? Good = 2; Moderate = 1; Don’t know/Poor = 0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>What is maturity of producer development process? CMM Level $\geq$ 3 = 2; CMM Level 2 = 1; CMM Level 1/Don’t know = 0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Are system resources needed by component/service available? Yes = 2; Not sure = 1; No = 0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>Is component/service cost within estimated cost? Yes = 2; No, but acceptable = 1; No = 0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
may be applied to $T_2$ to ensure that all components that need to be reconfigured also have helpdesk available. Thus the set $T_2$ contracts to set $T_3$: $T_3 = \{C_2, C_3\}$

By relating development aspects such risk and effort to appropriate checklist items we can formulate, for example, a low risk, low effort filter, to minimize their effects:

$$\forall c: T_1 \bullet (c.\text{checklist}(1) = 2 \land c.\text{checklist}(2) = 2 \land c.\text{checklist}(4) = 2 \land c.\text{checklist}(6) = 2 \land c.\text{checklist}(10) = 2 \land c.\text{checklist}(11) = 2$$

Where several development aspects are involved, component selection can be combined with a process of negotiation to find an acceptable trade-off [11].

3.5 Requirements Modelling

The concept of a service is common to many areas of computing, including digital libraries, distributed computing, data management and electronic commerce [4, 6]. In many of these areas the term service has several common characteristics, for example, functionality, quality and delivery [1], [6], [8]. In COREx, a service description is characterised by at least one identifiable function, a trigger by which the service commences, a recipient (Actor viewpoint), service provider (proposed system/component), conditions for service delivery and quality of service constraints (Fig. 3). Service descriptions may be partitioned into abstract sub-systems at design, which may be realised using concrete software components or services.

![Fig. 3. COREx service model.](image)

Service descriptions provide a mechanism for modelling viewpoint requirements and for mapping requirements to concrete software components and services. A service description comprises the following elements:
Invocation <Set of parameters required by a service and how the parameter values are used by service. Parameters correspond to attributes in the COREx service model >

Behaviour <Specification of the system behaviour resulting from the invocation of the service. This can be described at different levels of abstraction (e.g. use cases, state diagrams, sequence diagram etc.>

Constraints <Description of constraints on service >

Evaluation criterion <Tests that should be carried out to evaluate a component’s conformance with service>

Constraints define the overall qualities or attributes of the resulting system and are derived from non-functional requirements [9]. Constraints define the design (solution) space that includes a set of possible architectures that may be used to implement the proposed system. A security constraint may, for example, give rise to an architecture where security-critical services are held in a single component at the lower levels of layered architecture ensure a certain level of security. Performance needs may result in a dynamic architecture where popular service components are replicated with increasing load. This viewpoint/service-centric approach provides a framework for:

- Reasoning about the events that are responsible for triggering or suspending services. Because the events arise from actors in the system environment, it is possible to specify the required control information from the point of view of the actors. COREx represents this information as data attributes in viewpoints.
- Integrating functional and non-functional requirements. A viewpoint can impose quality constraints on the services it requires from the target system. For example, a viewpoint may require that a service have an availability of 98% (say between 8pm – 6pm, Monday to Friday). Alternatively a viewpoint may require that a service be provided within 30 seconds following a request (performance), or that a service is provided in a certain format (e.g. XML). A constraint description comprises the following elements:

  Identifier <Constraint identifier>
  Type <Constraint type (e.g. availability, response time, format, safety, security etc)>
  Rationale <Justification for constraint>
  Specification <Specification of constraint>
  Scope <Identifiers of services affected by constraint>
  Evaluation criterion <Tests to evaluate a component’s conformance with constraint>

Constraints related to effort, vendor and system resource requirements may provide the specifier with a mechanism for establishing the availability and viability of a component-based solution. Other types of constraint, for example dependability, filter down to the design stage where they form the basis for identifying appropriate architectures as part of a negotiated design process.

4 Conclusions

Component-based development is a highly a iterative process requiring simultaneous consideration of the system context (system characteristics such as requirements, cost, schedule, operating and support environments), capabilities of the software component, the marketplace and viable system designs. Current methods for specifying component-based systems have focused on later stages of specification and ignored the critical early stages of requirements definition.
Our solution has been to develop COREx, a viewpoint-oriented approach that supports component and service-oriented development. COREx provides an intuitive framework for eliciting and modelling requirements and for mapping these to component architectures, and provides the developer with a “pluggable” basis for custom development where available components or services are inadequate or inappropriate. The requirements process is supported by verification and negotiation at different levels of abstraction. It is impossible in a paper this size to describe the requirements method in detail, so we have tried to strike a balance between detail and the supported features. Future systems are likely to be hybrid with components and services co-existing in the same system. Adequate requirements engineering methods are absolutely necessary if such systems are to be reality, and if the risk associated with developing such systems is to be minimised. Our research partners have successfully applied COREx to the specification of non-trivial ERP systems.

Acknowledgement

The work described in this paper is being undertaken as part of the project ECO-ADM (IST 20771) Funded Under The EU IST Programme. We are grateful for the contribution of our partners In ECO-ADM.

References

CMEH: Container Managed Exception Handling for Increased Assembly Robustness

Kevin Simons and Judith Stafford

Department of Computer Science
Tufts University
Medford, MA USA 02155
{ksimons, jas}@cs.tufts.edu

Abstract. Component containers provide a deployment environment for components in a component-based system. Containers supply a variety of services to the components that are deployed in them, such as persistence, enforcement of security policies and transaction management. Recently, containers have shown a large amount of potential for aiding in the predictable assembly of component-based systems. This paper describes an augmentation to the component container, called the Container-Managed Exception Handling (CMEH) Framework, which provides an effective means for deploying exception handling mini-components into a component-based system. This framework promotes a more effective handling of exceptional events, as well as a better separation of concerns, yielding a more robust component assembly.

1 Introduction

The goal of this ongoing research is to develop a container-managed exception handling (CMEH) framework that facilitates the creation and deployment of modular exception handling mini-components in order to promote proper separation of concerns in COTS-based systems. Component developers are generally not aware of the components with which their software will interact when used in an assembly, and are therefore written to be as reusable as possible. The Java™ 2 Enterprise Edition (J2EE™) framework allows for binary implementations of Enterprise JavaBean™ (EJB) components to be directly “wired” together in a deployment without any sort of “glue code”. This can be accomplished via EJB metadata and reflection. Components can be directly connected based on the interfaces they provide and require. Directly connecting commercial-off-the-shelf (COTS) components provides a great many well known benefits, but also yields several problems with predicting the behavior of the system once it is assembled [2]. One such predictable assembly problem arises due to current exception-handling practices in component-based systems. COTS components are designed with no knowledge of the components with which they interact, and therefore have no knowledge of the exceptional behavior of such components resulting in three possible exception-related situations: (1) Components making calls to other components will be catching generic exceptions with very little useful exception handling. (2) The result of a component operation may be considered exceptional by the system developer in the context of the current application, but the exceptional result is allowed by the calling component. (3) There may be exception results that
CMEH: Container Managed Exception Handling for Increased Assembly Robustness

could be easily handled by the developer without requiring exception propagation back to the calling component, which would most likely handle the exception poorly in the first place.

Component containers are a receptacle into which components are deployed, providing a set of services that support component execution [4]. With these services implemented in the container, the developer is allowed to concentrate on writing components in their domain of expertise. Containers, therefore, provide an excellent model for the separation of concerns. Furthermore, all calls made to components must be relayed through the containers, so containers provide an excellent means of crosscutting method invocations. Containers are currently showing a great deal of promise in aiding in the problem of predictable assembly [4].

The basis of our research is an augmentation the J2EE container known as the Container-Managed Exception Handling (CMEH) framework. The ability to handle exceptions in the container alleviates the problem of improper exception handling in commercial components. Giving the system developer the ability to deal with exceptions in an application-specific context leads to useful handling of exceptions and more robust system performance. Furthermore, abstracting the exception handling into the container helps alleviate the tangle that occurs in exception handling code within commercial components [1]. For this research, the EJB container used was the container provided as a part of the JBoss open-source application server1. Before quality attributes of a system can be accurately predicted, exception behavior within the system must be handled in an application specific context.

2 The CMEH Framework

The CMEH framework allows system developers to quickly and easily deploy and manage exception handling components on a Java application server, allowing for more appropriate handling of component exceptions. The CMEH framework provides an event-driven model for handling exceptional behavior. By intercepting component method calls at a variety of points during method invocation and dispatching exception events at these points, the CMEH framework allows event handling code to correct the exceptional behavior of the system. There are three main events dispatched during a method invocation: method-called, method-returned and method-exception. When a component method is called, the invocation is intercepted before it reaches the component and the method-called event is fired. Handlers listening for this event have the opportunity to perform any necessary processing, before the invocation is released and allowed to reach the component method. If the method returns properly, the container stops the propagation of the return value and fires the method-returned event, again allowing the appropriate handlers to perform their processing. If instead the component method throws an exception, the propagation of the exception is paused and the method-exception event is fired. There are two additional events, test-component-state and recover-component-state that are used to handle cases where exceptional behavior results in a component being left in an invalid state.

1 www.jboss.org
Handling of Component Method-Exception Event

When an application-level exception is thrown by a method of a component, system developers have an opportunity to handle the exception after it is caught by the CMEH framework. This can be done in a variety of ways. In the simplest case, the exception-handling code can simply re-throw the exception, and it will be propagated back to the calling component. This is the generic behavior of the EJB container before the introduction of this new exception mechanism.

One useful option when handling the exceptions thrown by a component is to convert the exception into a different subclass of the Java `Exception` class [4]. This method allows for the exception to be translated into an exception that the calling component knows how to handle properly. Of course, knowing what exceptions a component can handle is not immediately obvious, and often must be discovered through use [2].

Another possible use of this mechanism is to stop the propagation of the exception altogether. Rather than propagating an exception back up the stack to the caller, the system developer may instead wish to return a value of the correct type to the caller. This will effectively allow the developer to return a default value to the calling component in the event of erroneous behavior, or to perform simple modifications to the return values in order to correct any possible errors that will occur when the return value reaches the calling component.

Handling of Component Method-Called and Method-Returned Events

The container-managed exception handling mechanism allows a system developer to check the arguments passed to a component method before the method has been executed. This provides the developer with several useful options. First, the developer can test the value of the arguments to ensure they are acceptable in the application and to the component being called. If they are, the method call continues as normal with the arguments being passed along to the called method. If the arguments are in any way out of range, the container exception handling code can raise an exception that will be propagated back to the calling component, effectively ending the method call. Again, the developer will be able to throw an exception that can be usefully handled by the calling component. Furthermore, the system developer can modify the values of the arguments, then allow the method call to proceed as normal, thus eliminating any erroneous behavior in the component receiving the method call.

Similar to the monitoring of arguments, this container model also provides the developer with the means to verify all return values returned by a component method. Once again, the return value can also be modified by the container exception code in order to ensure correct functioning of the system, or an exception can be propagated back to the caller.

Handling of Test-Component-State and Recover-Component-State Events

During the execution of a component method, a component may be left in a state that is invalid in the context of the application. Often the application is notified of this by way of an exception. After a component method throws an exception, the exception
either 1) propagates back to the caller of the method or 2) is translated or caught by a
handler of the method-exception event. In the CMEH framework, after an exception is
thrown by a component method, the framework fires a test-component-state
event after the method invocation has concluded and before the control flow of the
application progresses. By handling this event, the developer can write code to test the
state of the component in question. If the developer’s event handling method deter-
mines that the component’s state is invalid, a recover-component-state event
is fired. By handling this event, the system developer has an opportunity to correct the
state of the component before the application flow resumes. Exactly how to handle
this event is beyond the scope of this research, but the CMEH framework provides a
means of handling invalid component states in a modular fashion.

3 Implementation Details

The following sections assume a fair working knowledge of the J2EE framework. For
further information on J2EE, please see [3]. In the CMEH framework, handlers for the
various events are created by implementing a set of simple interfaces. The following
interface is representative of the entire set of interfaces that allows the system devel-
oper to handle each of the five exception event types:

```java
public interface MethodExceptionEventHandler {
    public Object handleMethodExceptionEvent(
        Invocation methodInvocation,
        ExceptionEvent event,
        MethodExceptionEventContext c) throws Exception;
}
```

In order to deploy their exception event handling code, the system developer must
modify the XML deployment descriptor of the EJB (the container configuration file)
whose methods they want to monitor. The system developer must add a new tag into
the <assembly-descriptor> portion of the deployment descriptor, as follows:

```xml
<assembly-descriptor>
<exception-handler>
    <method>
        <interface-name>TestEJBRemote</interface-name>
        <method-name>method1</method-name>
    </method>
    <method-called class="com.tufts.cmeh.mcHandler1"/>
    <method-exception class="com.tufts.cmeh.meHandler1"/>
</exception-handler>
```

The <exception-handler> tag is formatted much in the same way as
<method-permission> and <container-transaction> tags. The
(method> tag specifies which method is to be handled by the container-managed
exception handler. Each of *-handler tags specify the fully-qualified Java class to use in handling that event. It is valid to specify the same event handler class for several different component methods, and it is also valid to specify several handlers to handle the same event for the same component method, allowing exception handling code to be further modularized.

The next sections detail the JBoss application server implementation of the CMEH framework. Also covered is the reliance of the framework on several J2EE services, as well as services specific to JBoss.

3.1 The Interceptor Stack

In the JBoss server, services (such as transaction and security) are wrapped around a client’s call via the interceptor stack. The interceptor stack is a chain of stateless components that implement the Interceptor interface. This interface has a single method, invoke, that is passed a wrapped method call. The task of a single interceptor in the stack is to receive the invocation from the previous interceptor, perform any necessary processing, and then either pass the invocation on to the next interceptor, or throw an exception, effectively canceling the client’s method call.

The interceptor stack is located within the component container. The final interceptor in the chain is the container interceptor, which makes the actual call to the EJB method itself. The return value of the component method is then propagates back through interceptor stack, where once again the interceptors have the opportunity to perform operations on the invocation or throw an exception back to the client.

The CMEH framework adds a new interceptor to the chain that is responsible for intercepting method invocations at the appropriate times and dispatching the exception events.

3.2 The JMS-Based Exception Event Model

The exception event model in the container-managed exception handling framework is based on the Java Messaging Service (JMS). This service, which is provided as part of the JBoss J2EE application server, provides a means of dispatching and listening for asynchronous messages. In JMS, a topic is a form of channel on which listeners wait for messages. When the system developer deploys their exception event handlers, the framework automatically registers them to listen on the appropriate JMS topic. When an event is fired by the framework, a new JMS message is created and then dispatched to the correct topic. Event handlers, deployed to handle the type of event that is carried by the JMS message, will receive the event and a new thread is automatically created by JMS in which to handle the event. This allows for the framework to support asynchronous handling of exceptional behavior, which will prove helpful if several handlers are deployed on the same event for the same component method and some of the handling of exceptional behavior can be done concurrently. If several synchronous exception event handlers are deployed to handle the same event on the same component methods, the handlers receive the exception event in the order they were specified in the deployment descriptor. Specifying whether or not a handler is to be used asynchronously is also specified in the XML deployment descriptor.
3.3 The ExceptionHandlerService MBean

The exception handler service, responsible for the deployment of event handlers and dispatching JMS messages, is implemented as a Managed Bean or MBean in the CMEH framework. Other MBeans in JBoss include services for transactions and security. When an EJB wishing to use CMEH (as specified in the deployment descriptor) is deployed into the component container, the ExceptionHandlerService MBean deploys the event and registers them with the appropriate JMS topic so that they can have exception events dispatched to them. If the system developer deploys a new version of the exception handler when the system is up and running, the ExceptionHandlerService's class loader dynamically replaces the exception event listener object so that the new version will receive the events.

3.4 Exception Event Automation

Some exception event handling patterns are useful enough that they have been automated in the CMEH framework. For instance, translation of exceptions can be accomplished with the following addition to the deployment descriptor:

```xml
<method-exception class="com.tufts.cmeh.Translator">
  <translate from="IOException" to="MyException"/>
</method-exception>
```

By adding this to the XML deployment descriptor, the appropriate exception event handlers are created and deployed automatically, allowing the developer to leverage the CMEH framework without writing any event handling code.

4 Performance Overhead

Adding these exception handling mini-components to a software system introduces some performance overhead because there is exception checking code running on method calls that rarely produce any erroneous or exceptional behavior. Since this framework has not yet been deployed with a large-scale software system, empirical results have not been collected, however the added benefits of proper separation of concerns, ease of development and increased robustness should outweigh any costs.

There is a certain amount of overhead is added to the system by the framework itself, even if the exception handlers themselves are very simple. Empirical data has been collected for a software system with only two components. Various tests were run to determine the amount overhead added to the system with varying number of dummy exception handling components. All tests were run on a Linux machine (Redhat 8, 2 GHz), with the JBoss server running on the Sun Java SDK 1.4.1. The times were recorded when the invocation first entered the component and then again right before the methods returned, meaning the data represent only the invocation times, not the time to execute the method bodies. Method body times will typically greatly outweigh the invocation costs for complex components.
There is clearly a slight amount of overhead incurred purely by installing the CMEH, even before any event handlers are deployed. The framework also requires another 4 megabytes of RAM (a 2% increase). Currently, both the synchronous and asynchronous exception handlers are using JMS to provide the exception events. As a result, synchronous handlers incur a larger performance overhead. However, in theory, the synchronous handlers could be based on a much simpler mechanism (such as Java Events), cutting the performance costs.

5 Related Work

The CMEH Framework is directly related to work being done in the field of Aspect-oriented Programming (AOP). AOP for exception handling stresses the detangling of exceptional code, as well as support for multiple configurations, incremental development and dynamic reconfiguration [1]. However, at the time this work began, the AspectJ Java compiler required the source code of any classes that will use aspects, which is not a reasonable constraint when dealing with COTS components.

6 Conclusions and Future Work

Since commercial component developers are generally unaware of the exceptional behavior of the components their code will interact with, current exception handling techniques are insufficient for dealing with component-based systems. They fail to handle most exceptions in a useful way and they don’t allow for handling exception behavior in the context of the application. The CMEH framework provides a means for system developers to handle exceptional behavior in their systems in a simple, modular, well-organized fashion. It provides automation for deploying and maintaining event handlers that are used to handle exceptions in an application-specific way. By utilizing this framework, systems can be made more robust, reducing the number of errors and the mean-time-to-failure of the system in some cases. Before the exceptional behavior of the system is handled properly, accurately predicting the behavior of component assemblies is considerably more difficult.

Currently, the implementation strategy is focusing on both adding new features as well as minimizing the overhead imposed by the system. The synchronous event handlers will be implemented using simple Java events and will be removed from the JMS portion of the framework. Also, in order to make the asynchronous event handling more useful, features are being added to automatically deploy event handlers on

---

**Table 1. Latency Statistics for the CMEH Framework.**

<table>
<thead>
<tr>
<th>Method call</th>
<th>Method return</th>
<th>Method exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard JBoss</td>
<td>2.4 ms</td>
<td>.8 ms</td>
</tr>
<tr>
<td>CMEH, No handlers</td>
<td>5.2 ms</td>
<td>0.2 ms</td>
</tr>
<tr>
<td>1 Sync handler</td>
<td>37.0 ms</td>
<td>36.6 ms</td>
</tr>
<tr>
<td>3 Sync handlers</td>
<td>234.2 ms</td>
<td>141.4 ms</td>
</tr>
<tr>
<td>1 Async handler</td>
<td>37.0 ms</td>
<td>29.2 ms</td>
</tr>
<tr>
<td>3 Async handlers</td>
<td>76.2 ms</td>
<td>87.4 ms</td>
</tr>
</tbody>
</table>
to remote application servers in order to handle exceptional behavior in a truly parallel fashion. This should greatly increase the speed when performing processor-intensive exception recoveries, providing the machines have low network latency. New features for automatic detection and recovery from invalid component states will likely not be added to the framework, as it is outside the scope of this research.

References

A Framework for Constructing Adaptive Component-Based Applications: Concepts and Experiences

Humberto Cervantes and Richard S. Hall

Laboratoire LSR-IMAG, 220 rue de la Chimie, Domaine Universitaire, BP 53, 38041 Grenoble, Cedex 9, France

Abstract. This paper describes the experience of building component-oriented applications with a framework that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. The framework’s approach is to define a service-oriented component model, which is a component model that includes concepts from service orientation and an execution environment that provides automatic adaptation mechanisms. This paper focuses on an example scenario and two real-world examples where this framework has been used.

1 Introduction

This paper describes the experience of building component-oriented applications with a framework, called Gravity [4], that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. Dynamic availability is a situation where functionality provided by components may appear or disappear at any time during application execution and this outside of application control.

In this paper, dynamic availability is conceptually caused by continuous deployment activities that introduce and remove component functionality into the execution environment. Continuous deployment activities represent the installation, update, and removal of components during execution by an actor that may or may not be the application itself. The constituent components of an application are directly impacted by these activities, since they are likely to have dependencies on functionality provided by other components in order to provide their own functionalities. If some required functionality is removed, a component may not be able to provide its own functionality any longer; on the contrary, newly arriving functionality may enable another component to provide its own functionality.

The traditional component-oriented approach is based on the ideas that an application is assembled from reusable building blocks [9] (i.e., components) available at the time of assembly and that during execution, components are not spontaneously added or removed. As a consequence, dynamic availability is not supported explicitly in component models. Of course, it is possible to support dynamically available components through programmatic means, but this results in mixing both application and adaptation logic into the application code. In order to simplify this situation, component orientation should incorporate concepts from other approaches that explicitly support dynamic availability of functionality, such as service orientation.
oriented computing, services provide functionality that can be published and removed from service registries [4] at any time.

This paper discusses the experiences of building applications using the Gravity framework that provides a service-oriented component model and an associated execution environment. In this approach, functionality provided by components and their subsequent compositions are realized using service-orientation concepts. Special attention is paid to one aspect of the Gravity framework, called the Service Binder, that manages adaptation logic at run time. Applications built using this mechanism are capable of assembling and adapting autonomously. The basic approach of the Service Binder was presented in [3]; this paper describes how applications using this mechanism are built and how they adapt.

The remainder of the paper is structured as follows: section 2 discusses management of dynamic availability, section 3 presents an example scenario, section 4 presents two application scenarios where these concepts were applied, and section 5 presents related work followed by section 6 with conclusions and future work.

## 2 Managing Dynamic Availability

In a service-oriented component model, components provide and require functionality, where the functionality is modeled as a service that is published and discovered at run time using a service registry inside the execution environment. Gravity’s execution environment also manages dynamic availability by creating and destroying bindings among component instances using the service-oriented interaction pattern of publish-find-bind [3]. To enable this, a component provides information (dependency properties) about its service dependencies to the execution environment for run-time management.

In this approach, a component instance is either invalid or valid. An invalid instance’s service dependencies are not satisfied and it is unable to execute and to provide its services, while a valid instance’s service dependencies are satisfied and it is able to execute and provide its services. All instances initially start in an invalid state and become valid if their service dependencies are satisfied. At run time, instances may alternate between valid and invalid as required functionality is added to and removed from the execution environment, respectively. As such, the creation of a component instance is intentional, since the execution environment of the framework constantly tries to maintain the validity of the instance until it is explicitly destroyed.

To facilitate management, a component is described with a component descriptor; an example descriptor is depicted in figure 1. Inside the component tag of the descriptor, the class attribute refers to the component implementation, which is a Java class. The service attribute inside the provides and requires tags corresponds to provided and required services, respectively, which are described syntactically as Java interfaces. Additionally, the requires tag declares additional information including:

- **Cardinality.** Expresses both optionality and aggregation. In a component descriptor, the lower end of the cardinality value represents optionality, where a ‘0’ means that dependency is optional and ‘1’ means that it is mandatory. The upper end of the cardinality value represents aggregation, where a ‘1’ means the dependency is singular and ‘n’ means that it is aggregate.
Policy. Determines how dynamic service changes are handled: a *static* policy indicates that bindings cannot change at run time without causing the instance to become invalid, whereas a *dynamic* policy indicates that bindings can change at run time as long as bindings for mandatory dependencies are satisfied.

Filter. Constrains candidate providers of a required service to those matching a query (written in LDAP query syntax); the query is issued over the service properties associated with components using the *property* tag.

Bind/Unbind Methods. These methods allow the execution environment to set/unset references to the required service, i.e., create bindings.

During execution components may be installed or removed and, as a result, component instances are automatically created and destroyed by the execution environment. When an instance is created, it is associated with an *instance manager* that acts as a container responsible for managing instance’s life cycle. The instance manager uses the execution environment’s service registry to resolve and monitor its instance’s service dependencies. When all of the instance’s service dependencies are satisfied, the instance manager activates the instance and publishes its services; at this point the instance is valid. During execution, the service registry notifies instance managers about changes in service availability. An instance manager may need to reconfigure its bindings according to dependency properties in response to service registry notifications. If reconfiguration is not possible, the instance manager invalidates its instance and continuously tries to make it valid again.

3 Assembly and Adaptation of an Application

In a service-oriented component model, an application is constructed as a set of interconnected valid component instances at execution time. Such an application assembles and adapts autonomously as its constituent component instances are connected by their respective instance managers. Autonomous management for dynamic availability is discussed below, along with the main limitation of this approach.

3.1 Autonomous Management for Dynamic Availability

Initial Assembly. An application in this framework is typically built out of a “core” component instance that guides the application’s execution. Other component instances provide the services used by the core component and these instances can
themselves require services provided by other instances. The assembly process of such an application begins as instances are created inside the execution environment of the service-oriented component model. The execution of the application starts the moment the core instance becomes valid. The validation of the core instance occurs in a specific order that obeys the service dependency characteristics of the individual components. Specifically, the order is:

1. Instances with optional or no service dependencies are validated first.
2. Instances with mandatory service dependencies are validated if the services they require become available due to newly validated services.
3. The second step repeats as long as new services are introduced that resolve additional mandatory service dependencies.

Figure 2a depicts a word-processor application example that is built out of a core component instance (main from WordProcessorComponent) that uses services for spell checking and printing purposes. The main component instance, depicted at the left of the figure, has two service dependencies; the first dependency is on a spell checking service and is mandatory, singular, and dynamic, the second dependency is on a printing service and is optional, singular, and dynamic. In the figure, the main instance is bound to two instances that provide the required spell checking and printing services, checker (from SpellCheckComponent) and printer (from PrintComponent), respectively. The checker instance itself requires a dictionary service;
this dependency is mandatory, aggregate, and dynamic. The checker instance is bound to another component instance that provides a dictionary service (frenchdict from FrenchDictComponent).

If these four instances are created simultaneously, frenchdict and printer are validated first in no particular order, after which the frenchdict becomes valid, then the checker becomes valid, and finally main becomes valid. At this point the application begins execution.

**Instance Addition.** Figure 2b shows the word-processor application after a new instance providing a dictionary service (englishdict from EnglishDictComponent) becomes available. The characteristics of the spell checker component’s service dependency (aggregate and dynamic) allow new bindings to be created and, as a result, the englishdict instance is added to the application.

**Instance Removal.** Figure 2c shows the word-processor application after the removal of the printer instance. The destruction of the associated binding does not cause main to become invalid since the required service interface is characterized as optional and dynamic.

**Instance Substitution.** Instance substitution results from the removal of an instance that satisfies a service dependency that is mandatory, singular, and dynamic. In the word-processor application, this is the case for the SpellCheckService of the main instance. The service registry in figure 2c contains an entry for an additional SpellCheckService than the one being used by the application; since the service dependency is singular, only a single binding is created to one of the available spell checker services. In figure 2d, however, the checker instance is removed. In response, the instance manager for the main instance looks for a substitute, which is fulfilled by the other available component instance. The instance manager creates a binding to the new spell checker (checker2 from EnglishSpellCheckComponent) and the reconfiguration succeeds. Notice that checker2 is already bound to the existing English dictionary.

**Application Invalidation and Repair.** A situation that can occur as a result of an instance being destroyed is the triggering of a “chain reaction” that leads to the invalidation of the core component and, thus, the entire application. This situation occurs if the word-processor application is in the state depicted in figure 2a and the frenchdict instance is destroyed. The instance manager for the spell checker will look for a replacement, but the checker instance becomes invalid since no alternatives are available. This invalidation causes the main instance to become invalid, since no other spell checker is available. This situation causes the instance managers to enter new management cycles. As a consequence, as soon as a new instance providing a dictionary service becomes available, the spell checker is re-validated, followed by the main instance, and then the application as a whole.

### 3.2 Approach Limitations

The main limitation to this approach is the unpredictability of service dependency binding creation. Unpredictability results from the ambiguity that arises when there are multiple candidate services available to resolve a given service dependency. This situation occurs, for example, when a service dependency is singular, but at the time
the instance manager tries to resolve the dependency, multiple candidates are present. Unpredictability can be reduced by using filters in required service interfaces, but this is not sufficient. Knowing which of the available candidates is the best choice is difficult, if not impossible. This issue is similar to such research questions as locality, where the desire is to choose services that a physically near or appropriate.

4 Evaluations

This section describes two application scenarios in which the Service Binder\(^1\) mechanism was used; the Service Binder, and the service-oriented component model as a whole, is built on top of the OSGi service platform [8]. The Service Binder simplifies the construction of applications on the OSGi services platform, where dynamic availability is typically handled programmatically.

4.1 Device Monitoring at Schneider Electric

The Service Binder was used at Schneider Electric\(^2\) for a research project oriented around electric device monitoring. In this project, a series of electric devices are connected to a bus that is itself accessible to a gateway running an OSGi platform. A series of monitoring components inside the system are in charge of polling devices and producing notifications when exceptional situations occur. Requirements for this system are that it must run continuously and that it must be possible to add or remove new monitoring components in the running system as new devices are connected or disconnected to or from the bus, respectively.

Figure 3a represents the architecture of the system. Business objects contain monitoring logic and provide a service that allows a scheduler (the core in this application) to activate them periodically. Business objects have service dependencies on services that allow them to create logs and send e-mail notifications.

Support for the run-time addition or removal of business objects is achieved through the scheduler’s service dependency on the BOService, which is optional,

---

\(^1\) Available for download at http://gravity.sf.net/servicebinder

\(^2\) http://www.schneiderelectric.com
aggregate, and dynamic, and by the business objects’ service dependency on the polling service. In this application, business objects and notification mechanisms can be continuously introduced into or removed from the system as a result of continuous deployment activities that are triggered outside the application.

4.2 VersaTest Client

The Service Binder was used at a company called Ascert, which creates a system for testing online transaction processing systems, called VersaTest\(^3\). VersaTest runs as a server and a client for the VersaTest server was built as an extensible environment where different client-side tools are integrated as plug-ins. The different tools communicate with the server and graphically display different types of information that result from the server’s test cases. The VersaTest client is built around a core that provides the main functionality of the application, including a menu, toolbar, and a manager for multiple windows. These services allow multiple tools to work as a cohesive whole.

Figure 3b presents a simplified representation of the architecture of the VersaTest client system. The core component instance provides two services, one for adding entries to the menu and another one for creating windows. In this application, it is not the core component that requires services; instead, it provides services that are required by the tools. Tools not only require core services, but can themselves provide services and require services from other tools.

The VersaTest client application must support multiple configurations, where a configuration consists of the core and a particular set of tools. In this project, autonomous assembly capabilities provided by the Service Binder are leveraged as the VersaTest client is launched by starting an OSGi platform with a set of components corresponding to the different tools used in a configuration. This means that it is not necessary to create explicit compositions for each configuration, it is sufficient to simply include an arbitrary set of tools to compose. The Service Binder also allows tools to be added, removed, and updated during execution, but these capabilities are not currently used in the commercial version of the application.

5 Related Work

Related work includes component models and service platforms, along with techniques to create auto-adaptive applications. Industrial component models include COM [2], and CCM [7]. Service platforms include OSGi, Jini, and web services. Jini [1] is a distributed Java-based service platform that introduces the concept of leases to support distributed garbage collection. Web services target business application interoperability. The OSGi’s Wire Admin service [8] provides a mechanism to compose OSGi services between a producer and a consumer, but bindings are explicit and point-to-point.

Dynamically reconfigurable systems focus on changing the architecture of a system during execution. These systems use explicit architecture models and map changes in these models to the application implementation. Numerous works exist

\(^3\) http://www.ascert.com/versatest.html.
around self-adaptation through dynamic reconfiguration in component-based systems, such as [6] and [5]. This last work presents a framework to create self-organizing distributed systems. In this approach, component instances are managed independently and their connections are modified when component instances are introduced to or removed from the system according to constraints defined as an architectural style written in an ADL; however, the logic resulting from these constraints must be programmed.

6 Conclusions and Future Work

This paper described the experience of building component-oriented applications with a framework that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. In this framework, each component instance is managed independently and its bindings are created and adapted at run time based on information associated with a component’s service dependencies; instances are constantly managed to maintain their validity with respect to their service dependencies.

Applications built from this framework are capable of adding new functionality or releasing/substituting departing functionality. They also exhibit self-repairing characteristics since instance managers strive to maintain component instance validity and, consequently, the validity of the application as a whole. This framework was implemented on top of the OSGi framework and was successfully used in two different application scenarios.

Ongoing work includes studying mechanisms to reduce issues associated with unpredictability and ambiguity. These issues are exacerbated when multiple instances of the same application may exist at the same time, unlike the application scenarios described in this paper where only one application instance exists at any given time. An initial approach for addressing these issues is discussed in [4].

Finally, the authors would like to acknowledge the people at Schneider Electric and Ascert for their support and feedback.

References

Abstract. An object-oriented framework can be a key component in building products for a given application area. An application developer need only provide definitions suited to the needs of the particular application for the hook methods. With appropriate initializations, the calls to the hook methods made by the template methods defined in the framework will be dispatched to the definitions provided by the application developer, thus customizing the behavior of the template methods. Specifying and testing such frameworks present some challenges; we discuss these and develop ways to address them.

1 Introduction

Object-oriented (OO) application frameworks [19,12] can be thought of as large-grained software components. An early example was the MacApp framework [6] that provided many of the functionalities of applications for the Mac, thereby reducing the work involved in building individual applications, and ensuring a uniform look-and-feel among them. But specifying and testing the framework appropriately so that it can serve as a reliable foundation for building these applications presents many challenges, which we consider in this paper.

A framework contains one or more template methods [8] that, in a sense, mediate the interactions between different objects by calling at the right points appropriate hook methods of various classes. To build a new application, the developer provides definitions, suited to the needs of the particular application, for these hook methods. With appropriate initializations of objects of the framework as instances of these derived classes, calls to the hook methods that the template methods make are dispatched at run-time, via the mechanism of OO polymorphism, to their definitions in the derived classes. This results in the template methods exhibiting behavior customized to the needs of the application. Since the hook method call-patterns of the template methods are often the most involved aspect of the application’s total behavior, the framework can considerably reduce the effort required for developing new applications.

At the same time, polymorphism also leads to a reasoning problem, in particular in deducing the richer behavior that template methods exhibit in the application [5,21]. To address this, rather than using standard functional specifications [17], we will use a richer type of specification called an interaction specification, which will contain essential information about the template method’s calls to the hook methods, i.e., its interactions with these methods. We will illustrate the need for such specifications and their details, in the next section.
The second problem we address is how to test against such specifications. The standard approach [4] to specification-based testing is to choose an appropriate initial state and parameter values that satisfy the pre-condition of the method, invoke the method, and check whether the final state and result returned (if any) satisfy the post-condition of the method. Running the test is straightforward when using functional specifications, but testing against an interaction specification poses a problem since it refers to the calls the template method makes to the hook methods during its execution and this information is not recorded in the state. One solution to this would be to insert suitable instructions into the code of the template method bodies to record the needed information. However, such an approach not only changes the code being tested, it is not feasible if we did not have access to the source code. As Weyuker [23] notes, “as the reuse of software components and the use of COTS components become routine, we need testing approaches that are widely applicable regardless of the source code’s availability, because . . . typically only the object code is available.” In Section 3, we present our approach to testing against interaction specifications that requires no modification to, or even the availability of, the source code of the framework, and present results from a prototype implementation. In Section 4, we discuss related work and conclude with a pointer to future work.

final class Bank {
    protected Account a1, a2;
    public final void processTrans(String req) {
        Account acc; String trans; int amt;
        acc = AccNo(req); trans = TransName(req); amt = Amount(req);
        if(acc == 1) {
            if(trans.equals("deposit")) {a1.deposit(amt);}
            if(trans.equals("withdraw")) {a1.withdraw(amt);}
            if(trans.equals("transfer")) {... transfer amt from a1 to a2...} }
        elsif( acc == 2 ){ ... similar ... } }
    }

Fig. 1. Controller Class for Bank Framework.

2 Interaction Specifications

Consider the simple Bank class shown in Fig. 1, the controller class for a simple banking framework. Account is a generic account class (Fig. 2). A Bank object has two Accounts, a1 and a2. The application developer will define derived classes of Account; a1 and a2 will each be an instance of one of these derived classes. processTrans() is the template method of Bank, which processes a single transaction request contained in the String req. Each request consists of an account number, the name (“deposit, withdraw, or transfer”) of the transaction, and the amount involved. processTrans() extracts this information, and invokes the appropriate hook methods (deposit() and/or withdraw()) on the appropriate Account (a1 and/or a2). For transfers, if the given account number is 1, then the specified amount will be transferred from a1 to a2, and if it is 2, from a2 to a1. These calls will be dispatched to the corresponding methods of the appropriate derived classes, based on the actual classes that a1 and a2 are instances of.

The Account class methods increment and decrement balance in the expected manner. These methods are easily specified as in (1). “@pre” in the post-conditions denotes
class Account {
    protected int balance; // current balance
    Account() { balance = 0; }
    public void deposit(int amt) { balance = balance + amt; }
    public void withdraw(int amt) { balance = balance - amt; }
}

Fig. 2. Base Account Class.

[22] the value of the variable in question at the start of the method execution. We should note that behavioral subtyping [15] requires that redefinitions of deposit() and withdraw() in derived classes satisfy these specifications.

Let us now consider the behavior of the template method processTrans().

pre.Bank.processTrans(req) \equiv \text{LegitReq}(req) \quad (2)

\begin{align}
\text{post.Bank.processTrans(req)} & \equiv \\
& \left[ \left( \text{AccNo}(\text{req}) = 1 \right) \land \left( \text{TransName}(\text{req}) = \text{deposit} \right) \right] \Rightarrow \\
& \left\{ \neg a2 \land \left( a1.\text{balance} = a1.\text{balance}@pre + \text{Amount}(\text{req}) \right) \right\} \\
& \land \left[ \left( \text{AccNo}(\text{req}) = 1 \right) \land \left( \text{TransName}(\text{req}) = \text{withdraw} \right) \right] \Rightarrow \\
& \left\{ a2 \land \left( a1.\text{balance} = a1.\text{balance}@pre - \text{Amount}(\text{req}) \right) \right\} \\
& \land \left[ \left( \text{AccNo}(\text{req}) = 1 \right) \land \left( \text{TransName}(\text{req}) = \text{transfer} \right) \right] \Rightarrow \\
& \left\{ a1.\text{balance} = a1.\text{balance}@pre - \text{Amount}(\text{req}) \right\} \\
& \land \left( a2.\text{balance} = a2.\text{balance}@pre + \text{Amount}(\text{req}) \right) \right\} \quad (2.1)
\end{align}

The pre-condition requires that req be a “legitimate request”, the formal definition of which we omit. “!” in the post-condition denotes that the named variable is unchanged from when the method started. Thus the post-condition states that if the account number in req was 1, and the transaction was deposit, then a1.balance will be appropriately updated while a2 remains unchanged; withdraw and transfer are similar.

The specification (2) gives us the functional behavior of Bank.processTrans(). It does not tell us which hook methods this method invokes, nor on what objects. Without this information, in general we cannot predict the behavior processTrans() will exhibit in a particular application. Suppose, for example, as in Fig. 3, we define two derived classes, TCAccount and FeeAccount. TCAccount includes a transaction count of both deposits and withdrawals; FeeAccount imposes a transaction fee on withdrawals. Suppose the application is initialized so that a1 is of type FeeAccount and a2 of type TCAccount.

class TCAccount extends Account {
    protected int tCount; // transaction count (must be initialized to 0)
    public void deposit(int amt) { super.deposit(amt); tCount++; }
    public void withdraw(int amt) { super.withdraw(amt); tCount++; }
}

class FeeAccount extends Account {
    protected int tFee; // transaction fee (initialized to 0)
    public void withdraw(int amt) { super.withdraw(amt); tFee++; }
}

Fig. 3. Application-level Classes.
From (2), we see that a transfer request will appropriately update the balance for a1 and a2. However, (2) does not tell us which hook methods will be invoked on a1 and a2. It is possible that instead of invoking a1.withdraw(amt) and a2.deposit(amt), processTrans() may have directly decremented a2.balance and/or directly incremented a1.balance by amt instead. Depending on which implementation is used in the source code of the framework, the behavior of the application, i.e., the resulting values of a1.tFee and a2.tCount, will be different.

To address this, we need an interaction specification for each template method t() that provides information about the hook method calls that t() makes. In our example, we only need information about which methods were invoked, the number of times each was invoked, and the objects involved. But, in general, this is inadequate; if in TCAccount only transactions involving at least 51 dollars are counted, then we also need the values of the arguments in the calls. Or if in FeeAccount fee is imposed only if the balance is below 5000 at the time of withdrawal, we need information about the object state at the time of the call.

To handle these issues, we associate an auxiliary variable $\tau_t$ with each template method t(). $\tau_t$ is a trace or sequence that records information about t()’s hook method calls. At the start of t(), $\tau_t$ will be empty ($\varepsilon$). Consider a call a1.h(x) to a hook method h(); this will be recorded by appending a single element to $\tau_t$ that includes the identity of the object involved (a1), the identity of the hook method called (h), the arguments (x) passed, the state of the object at the time of the call, the result (if any) returned from the call, and the state at the time of the return. The post-condition of t()’s interaction specification will provide information about the elements in $\tau_t$, i.e., about the hook method calls t() made during execution; this will allow the application developer to predict the effect that the various hook method (re-)definitions in his or her derived classes will have on the behavior of the template methods of the framework.

To handle these issues, we associate an auxiliary variable $\tau_t$ with each template method t(). $\tau_t$ is a trace or sequence that records information about t()’s hook method calls. At the start of t(), $\tau_t$ will be empty ($\varepsilon$). Consider a call a1.h(x) to a hook method h(); this will be recorded by appending a single element to $\tau_t$ that includes the identity of the object involved (a1), the identity of the hook method called (h), the arguments (x) passed, the state of the object at the time of the call, the result (if any) returned from the call, and the state at the time of the return. The post-condition of t()’s interaction specification will provide information about the elements in $\tau_t$, i.e., about the hook method calls t() made during execution; this will allow the application developer to predict the effect that the various hook method (re-)definitions in his or her derived classes will have on the behavior of the template methods of the framework.

Consider (3) which is (part) of the interaction specification of processTrans(). Since we have only a single template method, we use $\tau$ to denote its trace. We label the pre-condition and post-condition in (3) "IS" to denote that this is an interaction, rather than functional specification. The pre-condition asserts that $\tau$ is empty (since at this point no hook methods have been invoked) and, as before, that req is a legitimate request. We are focusing on the case when the req is for transferring money from a1 to a2. The first part of this is the same as (2.1) and is omitted. The two new lines (3.2) tell us that the length of $\tau$, i.e., the number of elements in $\tau$, is 2; the object involved in the in the first element is a1; the method called is withdraw(); and the argument value is the same as Amount(req). The next line gives us similar information about the second element of $\tau$.

1 Although balance is a protected data member, Account and Bank might be in the same package, so Bank.processTrans() could access it directly.
Given this, the application developer who defines the derived classes **TCAccount** and **FeeAccount** as in Fig. 3 can see, given that \( a_1 \) is of type **FeeAccount** and \( a_2 \) of type **TCAccount**, that the effect of processing a transfer transaction from \( a_1 \) to \( a_2 \) would be to also increment \( a_1.tFee \) and to increment \( a_2.tCount \) by 1 each. The proof rules proposed in [20] allow the application developer to arrive at such conclusions by combining interaction specifications such as (3) with the behaviors of the hook methods implemented in the derived classes. Here our interest is in testing the template methods to see if they satisfy their interaction specification, and we turn to this next.

### 3 Testing the Framework

A key problem in testing a template method \( t() \) against its interaction specification is that the trace is not a real variable in its code. One approach around this is to introduce a variable, call it \( \tau \), initialize it to \( \langle \rangle \), and start execution of \( t() \). But when \( t() \) finishes, \( \tau \) will still be empty. What we need to do, in addition, is to introduce code that would append, to \( \tau \), information about every hook method call that \( t() \) makes. We could do this by inserting appropriate instructions immediately before and after every hook method call in the source code of \( t() \). Thus the call “aa.h(\( \overline{x} \))” would be replaced by:

\[
\overline{s} = aa.\overline{s}; \quad aa.h(\overline{x}); \quad \tau.append(aa, h, \overline{x}, s, \ldots);
\]

This appends to \( \tau \) an element recording information about this hook method call.

```java
class TS_Account extends Account {
    public Trace tau; // reference to the global Trace
    TS_Account(Trace t) {super.Account(); tau = t;} // binds tau to global Trace variable
    public void deposit(int amt) {
        traceRec tauel = ...info such as name of method called (deposit), object value
        and identity, parameter value (amt) etc...// tauel represents one element of tau.
        super.deposit(amt); // call original hook method
        tauel = ...add info about current state etc...
        tau.append(tauel); }
    public void withdraw(int amt) {...similar to above...}
}
```

Fig. 4. **TS_Account** class.

While this works, it changes the source code of the framework. What we need to do is intercept the hook method calls without modifying \( t() \). We could do that by changing the run-time system (the JVM in the case of Java) to perform the interception. Can we do better? The answer is yes, and it is based on exploiting polymorphism in the same manner that template and hook methods do. Let us return to our case study. Consider the **TS_Account** class (Fig. 4; “TS” denotes “trace-saving”). When testing `processTrans()`, we use instances of **TS_Account** for the Accounts \( a_1 \) and \( a_2 \); so whenever `deposit()` or `withdraw()` is invoked on \( a_1 \) or \( a_2 \), the call is dispatched to those defined in the trace-saving class. In Fig. 4, \( \tau \) is as usual the trace variable and \( \tau_{au} \) records information about one hook method call; this will be appended to \( \tau \) once the call to the original hook has finished and returned. One point of detail worth noting is that since hook method
calls on either of the account objects have to be recorded on the same trace, the two TS_Account objects have to share tau; we ensure this by having the constructor of this class receive the trace as an argument, and have the main() method of Test_Bank pass the same trace as argument when constructing the two TS_Account objects.

class Test_Bank {
  public static void test_processTrans(String req, Bank tc, Trace tau) {
    if (...tc and req satisfies processTrans's precond...) {
      Bank tc_old = ...save tc's initial state...
      tau.Clear();
      tc.processTrans(req);
      assert(...trace-based postcond of processTrans with approp. subs. ...); }
  }
  public static void main(String[] args) {
    Trace tau = new Trace();
    TS_Acct tsa1 = new TS_Acct(tau); TS_Acct tsa2 = new TS_Acct(tau);
    Bank tc = ...new Bank with tsa1 and tsa2 bound to a1 and a2 ...
    String req = ...a valid transaction request...
    test_processTrans(req, tc, tau); }
}

Fig. 5. The testing class Test_Bank.

Let us now consider the testing class, Test_Bank (Fig. 5). The main method constructs the TS_Account objects tsa1 and tsa2 and “registers” them with the Bank object (tc). Next we create a request req, and pass it to test_processTrans().

In test_processTrans(), we first check the pre-condition of processTrans(). Next we save the current state of tc (since the post-condition may refer to this state). (req does not appear in the post-condition, so we do not save it.) Next, we invoke processTrans() on this object; during this execution of processTrans(), the hook method calls will be recorded on tau. When control finally returns to test_processTrans(), we check if the the (interaction) post-condition is satisfied. If it is, this test “passes”.

The sequence diagram (Fig. 6) illustrates a test run. The first line represents the testing class Test_Bank; the second represents the test case object tc. The next two lines represent the a1 object of Bank. These two lines represent the two different aspects of this object: the base-class (Account) portion and the derived class (TS_Account) portion. We separate these two aspects to emphasize how the code in TS_Account and Account interact. The last two lines similarly represent the a2 object.

Fig. 6. Sequence Diagram for Test_Bank.test_processTrans().
Testing processTransSeq. (contd. from first column)
Method deposit called.
Method deposit called.
Method withdraw called.
Method withdraw called.
Method deposit called.

Postcond. of processTransSeq not met!

\[ \tau = \left( \begin{array}{l}
("deposit", (a1,1462), (a1,3015), 1553, 1553), \\
("deposit", (a2,497), (a2,880), 383, 383), \\
("withdraw", (a1,3015), (a1,2769), 246, 246), \\
("withdraw", (a1,2769), (a1,2297), 472, 472), \\
("deposit", (a2,880), (a2,1352), 472, 472) \end{array} \right) \]

Test number 1 failed!

Fig. 7. Output from sample run.

We have built a prototype\(^2\) based on our approach. The system takes as input the interaction specs for template methods and functional specs for the non-template methods, creates test classes and other classes needed for testing. The methods to be treated as hooks must be explicitly identified. The system creates skeleton calls to the test methods; the user must insert test values by hand. When creating test cases, the necessary trace-saving objects must also be properly registered. To do the actual testing, the generated classes are compiled, and the test class executed. An example of the system’s output is in Fig. 7. The one test shown here is for processTransSeq, a template method that executes a sequence of transaction requests by repeatedly invoking processTrans. The test case here performs two deposits, one withdraw, and then two transfers, the second of which uncovers a bug. We inserted a bug in the code so that transfers from a2 to a1 are performed by directly manipulating the individual Account balances. The elements of the trace indicate the name of the hook method, the incoming and outgoing Account object on which it was invoked, and the incoming and outgoing values of the amt parameter. To enable us to identify the account object involved, a name data member was introduced to the TS_Account class, which was set appropriately.

4 Related Work and Discussion

Several authors [5,10] have argued that dealing with hook and template methods requires additional information beyond functional specifications. Kirani and Tsai [13] also consider the need for specifying sequences of calls made by a method and testing against it; but they are interested in all calls, not just hook method calls. Ammons et al. [2] introduce a tool that infers trace specifications by observing patterns in execution runs, rather than testing against such specs. Other authors have addressed problems related to testing of polymorphic interactions [16,1,18] in OO systems. But in much of this work, the approach is, given the entire system, i.e., all the base and derived classes, test the behavior of each polymorphic method \(t()\) by using objects of different derived classes and check whether the resulting (functional) behavior of \(t()\) is appropriate in each case. These approaches usually also require access to the bodies of the template methods in question. General questions related to testing OO systems have been considered by [9,11,3,7] and others. Harrold et al. [9] proposes an algorithm to decide which tests of a base class can be reused in testing a derived class. Hsia et al. [11] deals also with client code in deciding which tests can be reused. Antoy and Hamlet [3] and Doong and Frankl [7] present systems that test against algebraic specifications. None of these, however, deal with interaction behaviors of template methods.

\(^2\)Available at: http://www.cis.ohio-state.edu/~tyler
We conclude with a pointer for future work. Suppose the framework objects contain references to other objects. In this case, what does it mean to save the state of a given object immediately before a hook method call? Only save the identity of the second object it refers to, or also the state of that object (and any that it refers to, etc.)? This is, of course, a standard problem in dealing with OO systems. JML [14] has notations for making these questions explicit and we intend to explore using JML, extended suitably to allow for traces, to express interaction specifications.

References

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable OO Software. Addison-Wesley, 1995.
Industrial Requirements on Component Technologies for Embedded Systems

Anders Möller\textsuperscript{1,2}, Joakim Fröberg\textsuperscript{1,3}, and Mikael Nolin\textsuperscript{1}

\textsuperscript{1} Mälardalen Real-Time Research Centre (MRTC)  
Mälardalen University, Box 883, SE–721 23, Västerås, Sweden  
\textsuperscript{2} CC Systems  
www.cc-systems.com  
\textsuperscript{3} Volvo Construction Equipment  
\{anders.moller,joakim.froberg,mikael.nolin\}@mdh.se  
www.volvo.com

Abstract. Software component technologies have not yet been generally accepted by embedded-systems industries. In order to better understand why this is the case, we present a set of requirements, based on industrial needs, that are deemed decisive for introducing a component technology. The requirements we present can be used to evaluate existing component technologies before introducing them in an industrial context. They can also be used to guide modifications and/or extensions to component technologies, to make them better suited for industrial deployment. One of our findings is that a major source of requirements is non-technical in its nature. For a component technology to become a viable solution in an industrial context, its impact on the overall development process needs to be addressed. This includes issues like component life-cycle management, and support for the ability to gradually migrate into the new technology.

1 Introduction

During the last decade, Component-Based Software Engineering (CBSE) for embedded systems has received a large amount of attention, especially in the software engineering research community. In the office/Internet area CBSE has had tremendous impact, and today components are downloaded and on the fly integrated into, e.g., word processors and web browsers. In industry however, CBSE is still, to a large extent, envisioned as a promising future technology to meet industry specific demands on improved quality and lowered cost, by facilitating software reuse, efficient software development, and more reliable software systems [1].

CBSE has not yet been generally accepted by embedded-system developers. They are in fact, to a large extent, still using monolithic and platform dependent software development techniques, in spite of the fact that this make software systems difficult to maintain, upgrade, and modify. One of the reasons for this

* This work is supported by Volvo Construction Equipment, CC Systems, and KKS (The Knowledge Foundation), within the project HEAVE.
status quo is that there are significant risks and costs associated with the adoption of a new development technique. These risks must be carefully evaluated and managed before adopting a new development process.

The main contribution of this paper is that it straightens out some of the question-marks regarding actual industrial requirements placed on a component technology. We describe the requirements on a component technology as elicited from two companies in the business segment of heavy vehicles. Many of the requirements are general for the automotive industry, or even larger parts of the embedded systems market (specifically segments that deal with issues about distributed real-time control in safety-critical environments), but there are also some issues that are specific for the business segment of heavy vehicles.

The list of requirements can be used to evaluate existing component technologies before introducing them in an industrial context, therefore minimising the risk when introducing a new development process. Thus, this study can help companies to take the step into tomorrow’s technology today. The list can also be used to guide modifications and/or extensions to component technologies, to make them better suited for industrial deployment within embedded system companies. Our list of requirements also illustrates how industrial requirements on products and product development impact requirements on a component technology.

This paper extends previous work, studying the requirements for component technologies, in that the results are not only based on our experience, or experience from a single company [2,3]. We base most of our results on interviews with senior technical staff at the two companies involved in this paper, but we have also conducted interviews with technical staff at other companies. Furthermore, since the embedded systems market is so diversified, we have limited our study to applications for distributed embedded real-time control in safety-critical environments, specifically studying companies within the heavy vehicles market segment. This gives our results higher validity, for this class of applications, than do more general studies of requirements in the embedded systems market [4].

2 Introducing CBSE in the Vehicular Industry

Component-Based Software Engineering arouses interest and curiosity in industry. This is mainly due to the enhanced development process and the improved ability to reuse software, offered by CBSE. Also, the increased possibility to predict the time needed to complete a software development project, due to the fact that the assignments can be divided into smaller and more easily defined tasks, is seen as a driver for CBSE.

CBSE can be approached from two, conceptually different, points of view; distinguished by whether the components are (1) used as a design philosophy independent from any concern for reusing existing components, or (2) seen as reusable off-the-shelf building blocks used to design and implement a component-based system [5]. When talking to industrial software developers with experience from using a CBSE development process [6], such as Volvo Construction Equip-
ment\(^1\), the first part, (1), is often seen as the most important advantage. Their experience is that the design philosophy of CBSE gives rise to good software architecture and significantly enhanced ability to divide the software in small, clearly-defined, development subprojects. This, in turn, gives predictable development times and shortens the time-to-market. The second part, (2), are by these companies often seen as less important, and the main reason for this is that experience shows that most approaches to large scale software reuse is associated with major risks and high initial costs. Rather few companies are willing to take these initial costs and risks since it is difficult to guarantee that money is saved in the end.

On the other hand, when talking to companies with less, or no, experience from component-based technologies, (2) is seen as the most important motivation to consider CBSE. This discrepancy between companies with and without CBSE experience is striking.

However, changing the software development process to using CBSE does not only have advantages. Especially in the short term perspective, introducing CBSE represents significant costs and risks. For instance, designing software to allow reuse requires (sometimes significantly) higher effort than does designing for a single application [7]. For resource constrained systems, design for reuse is even more challenging, since what are the most critical resources may vary from system to system (e.g. memory or CPU-load). Furthermore, a component designed for reuse may exhibit an overly rich interface and an associated overly complex and resource consuming implementation. Hence, designing for reuse in resource constrained environments requires significant knowledge not only about functional requirements, but also about non-functional requirements. These problems may limit the possibilities of reuse, even when using CBSE.

With any software engineering task, having a clear and complete understanding of the software requirements is paramount. However, practice shows that a major source of software errors comes from erroneous, or incomplete, specifications [7]. Often incomplete specifications are compensated for by engineers having good domain knowledge, hence having knowledge of implicit requirements. However, when using a CBSE approach, one driving idea is that each component should be fully specified and understandable by its interface. Hence, the use of implicit domain knowledge not documented in the interface may hinder reuse of components. Also, division of labour into smaller projects focusing on single components, require good specifications of what interfaces to implement and any constraints on how that implementation is done, further disabling use of implicit domain knowledge. Hence, to fully utilise the benefits of CBSE, a software engineering process that do not rely on engineers’ implicit domain knowledge need to be established.

Also, when introducing reuse of components across multiple products and/or product families, issues about component management arise. In essence, each component has its own product life-cycle that needs to be managed. This includes version and variant management, keeping track of which versions and

\(^1\) Volvo Construction Equipment, Home Page: http://www.volvo.com
variants is used in what products, and how component modifications should be propagated to different version and variants. Components need to be maintained, as other products, during their life cycle. This maintenance needs to be done in a controlled fashion, in order not to interfere aversely with ongoing projects using the components. This can only be achieved using adequate tools and processes for version and variant management.

3 A Component Technology for Heavy Vehicles

Existing component technologies are in general not applicable to embedded computer systems, since they do not consider aspects such as safety, timing, and memory consumption that are crucial for many embedded systems [8,9]. Some attempts have been made to adapt component technologies to embedded systems, like, e.g., MinimumCORBA [10]. However, these adaptations have not been generally accepted in the embedded system segments. The reason for this is mainly due to the diversified nature of the embedded systems domain. Different market segments have different requirements on a component technology, and often, these requirements are not fulfilled simply by stripping down existing component technologies; e.g. MinimumCORBA requires less memory then does CORBA, however, the need to statically predict memory usage is not addressed.

It is important to keep in mind that the embedded systems market is extremely diversified in terms of requirements placed on the software. For instance, it is obvious that software requirements for consumer products, telecom switches, and avionics are quite different. Hence, we will focus on one single market segment: the segment of heavy vehicles, including, e.g., wheel loaders and forest harvesters. It is important to realise that the development and evaluation of a component technology is substantially simplified by focusing on a specific market segment. Within this market segment, the conditions for software development should be similar enough to allow a lightweight and efficient component technology to be established [11].

3.1 The Business Segment of Heavy Vehicles

Developers of heavy vehicles faces a situation of (1) high demands on reliability, (2) requirements on low product cost, and (3) supporting many configurations, variants and suppliers. Computers offer the performance needed for the functions requested in a modern vehicle, but at the same time vehicle reliability must not suffer. Computers and software add new sources of failures and, unfortunately, computer engineering is less mature than many other fields in vehicle development and can cause lessened product reliability. This yields a strong focus on the ability to model, predict, and verify computer functionality.

At the same time, the product cost for volume products must be kept low. Thus, there is a need to include a minimum of hardware resources in a product (only as much resources as the software really needs). The stringent cost requirements also drive vehicle developers to integrate low cost components from
suppliers rather than develop in-house. On top of these demands on reliability and low cost, vehicle manufacturers make frequent use of product variants to satisfy larger groups of customers and thereby increase market share and product volume.

In order to accommodate (1)-(3), as well as an increasing number of features and functions, the electronic system of a modern vehicle is a complex construction which comprise electronic and software components from many vendors and that exists in numerous configurations and variants.

The situation described cause challenges with respect to verification and maintenance of these variants, and integration of components into a system. Using software components, and a CBSE approach, is seen as a promising way to address challenges in product development, including integration, flexible configuration, as well as good reliability predictions, scalability, software reuse, and fast development. Further, the concept of components is widely used in the vehicular industry today. Using components in software would be an extension of the industry’s current procedures, where the products today are associated with the components that constitute the particular vehicle configuration.

What distinguishes the segment of heavy vehicles in the automotive industry is that the product volumes are typically lower than that of, e.g., trucks or passenger cars. Also the customers tend to be more demanding with respect to technical specifications such as engine torque, payload etc, and less demanding with respect to style. This causes a lower emphasis on product cost and optimisation of hardware than in the automotive industry in general. The lower volumes also make the manufacturers more willing to design variants to meet the requests of a small number of customers.

However, the segment of heavy vehicles is not homogeneous with respect to software and electronics development practices. For instance, the industrial partners in this paper face quite different market situations and hence employ different development techniques:

- CC Systems\(^2\) (CCS) is developing and supplying advanced distributed embedded real-time control systems with focus on mobile applications. Examples, including both hardware and software, developed by CCS are forest harvesters, rock drilling equipment and combat vehicles. The systems developed by CCS are built to endure rough environments, and are characterised by safety criticality, high functionality, and the requirements on robustness and availability are high.

CCS works as a distributed software development partner, and cooperates, among others, with Alvis Hägglunds\(^3\), Timberjack\(^4\) and Atlas Copco\(^5\). Experience from these companies are included in this paper, this makes our findings more representative for the business segment of heavy vehicles.

\(^2\) CC Systems, Home page: [http://www.cc-systems.com](http://www.cc-systems.com)
\(^3\) Alvis Hägglunds, Home page: [http://www.alvishagglunds.se](http://www.alvishagglunds.se)
\(^4\) Timerjack, Home page: [http://www.timerjack.com](http://www.timerjack.com)
CCS’ role as subcontractor requires a high degree of flexibility with respect to supported target environments. Often, CCS’ customers have requirements regarding what hardware or operating systems platforms to use, hence CCS cannot settle to support only some predefined set of environments. Nevertheless, to gain competitive advantages, CCS desires to reuse software between different platforms.

- Volvo Construction Equipment (VCE) is one of the world’s major manufacturers of construction equipment, with a product range encompassing wheel loaders, excavators, motor graders, and more. What these products have in common is that they demand high reliability control systems that are maintainable and still cheap to produce. The systems are characterised as distributed embedded real-time systems, which must perform in an environment with limited hardware resources.

VCE develops the vehicle electronics and most software in house. Some larger software parts, such as the operating system, are bought from commercial suppliers. VCE’s role as both system owner and system developer gives them full control over the system’s architecture. This, in turn, has given them the possibility to select a small set of (similar) hardware platforms to support, and select a single operating systems to use. Despite this degree of control over the system, VCE’s experience is that software reuse is still hindered; for instance by non-technical issues like version and variant management, and configuration management.

### 3.2 System Description

In order to describe the context for software components in the vehicular industry, we will first explore some central concepts in vehicle electronic systems. Here, we outline some common and typical solutions and principles used in the design of vehicle electronics. The purpose is to describe commonly used solutions, and outline the de facto context for application development and thereby also requirements for software component technologies.

The system architecture can be described as a set of computer nodes called Electronic Control Units (ECUs). These nodes are distributed throughout the vehicle to reduce cabling, and to provide local control over sensors and actuators. The nodes are interconnected by one or more communication bus forming the network architecture of the vehicle. When several different organisations are developing ECUs, the bus often acts as the interface between nodes, and hence also between the organisations. The communication bus is typically low cost and low bandwidth, such as the Controller Area Network (CAN) [12].

In the example shown in Fig. 1 on the following page, the two communication busses are separated using a gateway. This is an architectural pattern that can be used for several reasons, e.g., separation of criticality, increased total communication bandwidth, fault tolerance, compatibility with standard protocols [13,14,15], etc. Also, safety critical functions may require a high level of verification, which is usually very costly. Thus, non-safety related functions might be
separated to reduce cost and effort of verification. In some systems the network is required to give synchronisation and provide a fault tolerance mechanisms.

The hardware resources are typically scarce due to the requirements on low product cost. Addition of new hardware resources will always be defensive, even if customers are expected to embrace a certain new function. Because of the uncertainty of such expectations, manufacturers have difficulties in estimating the customer value of new functions and thus the general approach is to keep resources at a minimum.

In order to exemplify the settings in which software components are considered, we have studied our industrial partner’s currently used nodes. Below we list the hardware resources of a typical ECU with requirements on sensing and actuating, and with a relatively high computational capacity (this example is from a typical power train ECU):

Processor: 25 MHz 16 bit processor (e.g. Siemens C167)
Flash: 1 MB used for applications
RAM: 128 kB used for the runtime memory usage
EEPROM: 64 kB used for system parameters
Serial interfaces: RS232 or RS485, used for service purpose
Communications: Controller Area Network (CAN) (one or more interfaces)
I/O: There is a number of digital and analogue in and out ports

Also, included in a vehicle’s electronic system can be display computer(s) with varying amounts of resources depending on product requirements. There may also be PC-based ECU’s for non-control applications such as telematics, and information systems. Furthermore, in contrast to these resource intense ECU’s,
there typically exists a number of small and lightweight nodes, such as, intelligent sensors (i.e. processor equipped, bus enabled, sensors).

Figure 2 depicts the typical software architecture of an ECU. Current practice typically builds on top of a reusable “software platform”, which consists of a hardware abstraction layer with device drivers and other platform dependent code, a Real-Time Operating System (RTOS), one or more communication protocols, and possibly a software (component) framework that is typically company (or project) specific. This software platform is accessible to application programmers through an Application Programmers Interface (API). Different nodes, presenting the same API, can have different realisation of the different parts in the software platform (e.g. using different RTOSs).

Today it is common to treat parts of the software platform as components, e.g. the RTOS, device drivers, etc, in the same way as the ECU’s bus connectors and other hardware modules. That is, some form of component management process exists; trying to keep track of which version, variant, and configuration of a component is used within a product. This component-based view of the software platform is however not to be confused with the concept of CBSE since the components does not conform to standard interfaces or component models.

4 Requirements on a Component Technology for Heavy Vehicles

There are many different aspects and methods to consider when looking into questions regarding how to capture the most important requirements on a component technology suited for heavy vehicles. Our approach has been to cooperate with our industrial partners very closely, both by performing interviews and by participating in projects. In doing so, we have extracted the most important requirements on a component-based technique from the developers of heavy vehicles point of view.

The requirements are divided in two main groups, the technical requirements (Sect. 4.1) and the development process related requirements (Sect. 4.2). Also,
in Sect. 4.3 we present some implied (or derived) requirements, i.e. requirements that we have synthesised from the requirements in sections 4.1 and 4.2, but that are not explicit requirements from industry. In Sect. 4.4 we discuss, and draw conclusions from, the listed requirements.

4.1 Technical Requirements

The technical requirements describe the needs and desires that our industrial partners have regarding the technically related aspects and properties of a component technology.

4.1.1 Analysable. Vehicle industry strives for better analyses of computer system behaviour in general. This striving naturally affects requirements placed on a component model. System analysis, with respect to non-functional properties, such as the timing behaviour and the memory consumption, of a system built up from well-tested components is considered highly attractive. In fact, it is one of the single most distinguished requirements defined by our industrial partners.

When analysing a system, built from well-tested, functionally correct, components, the main issues is associated with composability. The composability problem must guarantee non-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system [1].

When considering timing analysability, it is important to be able to verify (1) that each component meet its timing requirements, (2) that each node (which is built up from several components) meet its deadlines (i.e. schedulability analysis), and (3) to be able to analyse the end-to-end timing behaviour of functions in a distributed system.

Because of the fact that the systems are resource constrained (Sect. 3), it is important to be able to analyse the memory consumption. To check the sufficiency of the application memory, as well as the ROM memory, is important. This check should be done pre-runtime to avoid failures during runtime.

In a longer perspective, it is also desirable to be able to analyse properties like reliability and safety. However, these properties are currently deemed too difficult to address on a component level and traditional methods (like testing and reviewing) are considered adequate.

4.1.2 Testable and Debuggable. It is required that there exist tools that support debugging both at component level, e.g. a graphical debugging tool showing the components in- and out-port values, and at the traditional white-box source code debugging level. The test and debug environment needs to be “component aware” in the sense that port-values can be monitored and traced and that breakpoints can be set on component level.

Testing and debugging is by far the most commonly used technique to verify software systems functionality. Testing is a very important complement to
analysis, and it should not be compromised when introducing a component technology.

In fact, the ability to test embedded-system software can be improved when using CBSE. This is possible because the component functionality can be tested in isolation. This is a desired functionality asked for by our industrial partners. This test should be used before the system tests, and this approach can help finding functional errors and source code bugs at the earliest possible opportunity.

4.1.3 Portable. The components, and the infrastructure surrounding them, should be platform independent to the highest degree possible. Here, platform independent means hardware independent, RTOS independent and communication protocol independent.

Components are kept portable by minimising the number of dependencies to the software platform. Such dependencies are off course necessary to construct an executable system, however the dependencies should be kept to a minimum, and whenever possible dependencies should be generated automatically by configuration tools.

Ideally, components should also be independent of the component framework used during run-time. This may seem far fetched, since traditionally a component model has been tightly integrated with its component framework. However, support for migrating components between component frameworks is important for companies cooperating with different customers, using different hardware and operating systems, such as CC Systems.

4.1.4 Resource Constrained. The components should be small and light-weighted and the components infrastructure and framework should be minimised. Ideally, there should no run-time overhead compared to not using a CBSE approach.

Systems are resource constrained to lower the production cost and thereby increase profit. When companies design new ECUs, future profit is the main concern. Therefore the hardware is dimensioned for anticipated use but not more.

Provided that the customers are willing to pay the extra money, to be able to use more complex software functionality in the future, more advanced hardware may be appropriate. This is however seldom the case, usually the customers are very cost sensitive. The developer of the hardware rarely takes the extra cost to extend the hardware resources, since the margin of profit on electronics development usually is low.

One possibility, that can significantly reduce resource consumption of components and the component framework, is to limit the possible run-time dynamics. This means that it is desirable to allow only static, off-line, configured systems. Many existing component technologies have been design to support high runtime dynamics, where components are added, removed and reconfigured at runtime. However, this dynamic behaviour comes at the price of increased resource consumption.
4.1.5 Component Modelling. A component technology should be based on a standard modelling language like UML [16] or UML 2.0 [17]. The main reason for choosing UML is that it is a well known and thoroughly tested modelling technique with tools and formats supported by third-party developers.

The reason for our industrial partners to have specific demands in these details, is that it is believed that the business segment of heavy vehicles does not have the possibility do develop their own standards and practices. Instead they preferably relay on the use of simple and mature techniques supported by a welth of third party suppliers.

4.1.6 Computational Model. Components should preferably be passive, i.e. they should not contain their own threads of execution. A view where components are allocated to threads during component assembly is preferred, since this is believed to enhance reusability, and to limit resource consumption. The computational model should be focused on a pipe-and-filter model [18]. This is partly due to the well known ability to schedule and analyse this model off-line. Also, the pipes-and-filters model is a good conceptual model for control applications.

However, experience from VCE shows that the pipe-and-filter model does not fit all parts of the system, and that force fitting applications to the pipe-and-filter model may lead to overly complex components. Hence, it is desirable to have support for other computational models; unfortunately, however, which models to support is not obvious and is an open question.

4.2 Development Requirements

When discussing requirements for CBSE technologies, the research community often overlooks requirements related to the development process. For software developing companies, however, these requirements are at least as important as the technical requirements. When talking to industry, earning money is the main focus. This cannot be done without having an efficient development processes deployed. To obtain industrial reliance, the development requirements need to be considered and addressed by the component technology and tools associated with the technology.

4.2.1 Introducible. It should be possible for companies to gradually migrate into a new development technology. It is important to make the change in technology as safe and inexpensive as possible.

Revolutionary changes in the development technique used at a company are associated with high risks and costs. Therefore a new technology should be possible to divide into smaller parts, which can be introduced separately. For instance, if the architecture described in Fig. 2 is used, the components can be used for application development only and independently of the real-time operating system. Or, the infrastructure can be developed using components, while the application is still monolithic.
One way of introducing a component technology in industry, is to start focusing on the development process related requirements. When the developers have accepted the CBSE way of thinking, i.e. thinking in terms of reusable software units, it is time to look at available component technologies. This approach should minimise the risk of spending too much money in an initial phase, when switching to a component technology without having the CBSE way of thinking.

### 4.2.2 Reusable.
Components should be reusable, e.g., for use in new applications or environments than those for which they where originally designed [19]. The requirement of reusability can be considered both a technical and a development process related requirement. Development process related since it has to deal with aspects like version and variant management, initial risks and cost when building up a component repository, etc. Technical since it is related to aspects such as, how to design the components with respect to the RTOS and HW communication, etc.

Reusability can more easily be achieved if a loosely coupled component technology is used, i.e. the components are focusing on functionality and do not contain any direct operating system or hardware dependencies. Reusability is simplified further by using input parameters to the components. Parameters that are fixed at compile-time, should allow automatic reduction of run-time overhead and complexity.

A clear, explicit, and well-defined component interface is crucial to enhance the software reusability. To be able to replace one component in the software system, a minimal amount of time should be spent trying to understand the component that should be interchanged.

It is, however, both complex and expensive to build reusable components for use in distributed embedded real-time systems [1]. The reason for this is that the components must work together to meet the temporal requirements, the components must be light-weighted since the systems are resource constrained, the functional errors and bugs must not lead to erroneous outputs that follow the signal flow and propagate to other components and in the end cause unsafe systems. Hence, reuse must be introduced gradually and with grate care.

### 4.2.3 Maintainable.
The components should be easy to change and maintain, meaning that developers that are about to change a component need to understand the full impact of the proposed change. Thus, not only knowledge about component interfaces and their expected behaviour is needed. Also, information about current deployment contexts may be needed in order not to break existing systems where the component is used.

In essence, this requirement is a product of the previous requirement on reusability. The flip-side of reusability is that the ability to reuse and reconfigure the components using parameters leads to an abundance of different configurations used in different vehicles. The same type of vehicle may use different software settings and even different component or software versions. So, by introducing reuse we introduce more administrative work.
Reusing software components lead to a completely new level of software management. The components need to be stored in a repository where different versions and variants need to be managed in a sufficient way. Experiences from trying to reuse software components show that reuse is very hard and initially related with high risks and large overheads [1]. These types of costs are usually not very attractive in industry.

The maintainability requirement also includes sufficient tools supporting the service of the delivered vehicles. These tools need to be component aware and handle error diagnostics from components and support for updating software components.

4.2.4 Understandable. The component technology and the systems constructed using it should be easy to understand. This should also include making the technology easy and intuitive to use in a development project.

The reason for this requirement is to simplify evaluation and verification both on the system level and on the component level. Also, focusing on an understandable model makes the development process faster and it is likely that there will be fewer bugs.

It is desirable to hide as much complexity as possible from system developers. Ideally, complex tasks (such as mapping signals to memory areas or bus messages, or producing schedules or timing analysis) should be performed by tools. It is widely known that many software errors occur in code that deals with synchronisation, buffer management and communications. However, when using component technologies such code can, and should, be automatically generated; leaving application engineers to deal with application functionality.

4.3 Derived Requirements

Here, we present two implied requirements, i.e. requirements that we have synthesised from the requirements in sections 4.1 and 4.2, but that are not explicit requirements from the vehicular industry.

4.3.1 Source Code Components. A component should be source code, i.e., no binaries. The reasons for this include that companies are used to have access to the source code, to find functional errors, and enable support for white box testing (Sect. 4.1.2). Since source code debugging is demanded, even if a component technology is used, black box components is undesirable.

Using black-box components would, regarding to our industrial partners, lead to a feeling of not having control over the system behaviour. However, the possibility to look into the components does not necessary mean that you are allowed to modify them. In that sense, a glass-box component model is sufficient.

Source code components also leaves room for compile-time optimisations of components, e.g., stripping away functionality of a component that is not used in a particular application. Hence, source code components will contribute to lower resource consumption (Sect. 4.1.4).
4.3.2 Static Configuration. For a component model to better support the technical requirements of analysability (Sect. 4.1.1), testability (Sect. 4.1.2), and light-weightiness (Sect. 4.1.4), the component model should be configured pre-runtime, i.e. at compile time. Component technologies for use in the office/Internet domain usually focus on a dynamic behaviour [8,9]. This is of course appropriate in this specific domain, where powerful computers are used. Embedded systems, however, face another reality - with resource constrained ECU's running complex, dependable, control applications. Static configuration should also improve the development process related requirement of understandability (Sect. 4.2.4), since there will be no complex run-time reconfigurations.

Another reason for the static configuration is that a typical control node, e.g. a power train node, does not interact directly with the user at any time. The node is started when the ignition key is turned on, and is running as a self-contained control unit until the vehicle is turned off. Hence, there is no need to reconfigure the system during runtime.

4.4 Discussion
Reusability is perhaps the most obvious reason to introduce a component technology for a company developing embedded real-time control systems. This matter has been the most thoroughly discussed subject during our interviews. However, it has also been the most separating one, since it is related to the question of deciding if money should be invested in building up a repository of reusable components.

Two important requirements that has emerged during the discussions with our industrial partners are safety and reliability. These two are, as we see it, not only associated with the component technology. Instead, the responsibility of designing safe and reliable system rests mainly on the system developer. The technology and the development process should, however, give good support for designing safe and reliable systems.

Another part that has emerged during our study is the need for a quality rating of the components depending on their success when used in target systems. This requirement can, e.g., be satisfied using Execution Time Profiles (ETP’s), discussed in [20]. By using ETPs to represent the timing behaviour of software components, tools for stochastic schedulability analysis can be used to make cost-reliability trade offs by dimensioning the resources in a cost efficient way to achieve the reliability goals. There are also emerging requirements regarding the possibilities to grade the components depending on their software quality, using for example different SIL (Safety Integrity Levels) [21] levels.

5 Conclusions
Using software components and a CBSE approach is, by industry, seen as a promising way to address challenges in product development including integration, flexible configuration, as well as good reliability predictions, scalability,
reliable reuse, and fast development. However, changing the software development process to using CBSE does not only have advantages. Especially in the short term perspective, introducing CBSE represents significant costs and risks.

The main contribution of this paper is that it straightens out some of the question-marks regarding actual industrial requirements placed on a component technology. We describe the requirements on a component technology as elicited from two companies in the business segment of heavy vehicles. The requirements are divided in two main groups, the technical requirements and the development process related requirements. The reason for this division is mainly to clarify that the industrial actors are not only interested in technical solutions, but also in improvements regarding their development process.

The list of requirements can be used to evaluate existing component technologies before introducing them in an industrial context, therefore minimising the risk when introducing a new development process. Thus, this study can help companies to take the step into tomorrow’s technology today. They can also be used to guide modifications and/or extensions to component technologies, to make them better suited for industrial deployment within embedded system companies.

We will continue our work by evaluating existing software component technologies with respect to these requirements. Our initial findings from this evaluation can be found in [22]. Using that evaluation we will (1) study to what extent existing technologies can be adapted in order to fulfil the requirements of this paper, (2) investigate if selected parts of standard technologies like tools, middleware, and message-formats can be reused, (3) make a specification of a component technology suitable for heavy vehicles, and (4) build a test bed implementation based on the specification.

Acknowledgements

A special thanks to Nils-Erik Bänkestad and Robert Larsson, at Volvo Construction Equipment, for fruitful discussions and for their helpfulness during our stay. We would also like to thank Jörgen Hansson at CC Systems for interesting discussions, new ideas, and for making this research project possible.

References

Prediction of Run-Time Resource Consumption in Multi-task Component-Based Software Systems

Johan Muskens and Michel Chaudron

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513
5600 MB Eindhoven, The Netherlands
{J.Muskens,M.R.V.Chaudron}@tue.nl

Abstract. Embedded systems must be cost-effective. This imposes strict requirements on the resource consumption of their applications. It is therefore desirable to be able to determine the resource consumption of applications as early as possible in its development. Only then, a designer is able to guarantee that an application will fit on a target device.

In this paper we will present a method for predicting run-time resource consumption in multi-task component based systems based on a design of an application. In [5] we describe a scenario based resource prediction technique and show that it can be applied to non-pre-emptive non-processing resources, like memory. In this paper we extend this technique, which enables us to handle pre-emptive processing resources and their scheduling policies. Examples of these class of resources are CPU and network.

For component based software engineering the challenge is to express resource consumption characteristics per component, and to combine them to do predictions over compositions of components. To this end, we propose a model and tools, for combining individual resource estimations of components. These composed resource estimations are then used in scenarios (which model run-time behavior) to predict resource consumption.

1 Introduction

Our research was carried out in the context of the Robocop project\textsuperscript{1}. The aim of Robocop is to define an open, component-based framework for the middleware layer in high-volume embedded appliances. The framework enables robust and reliable operation, upgrading, extension, and component trading. The appliances targeted by Robocop are consumer devices such as mobile phones, set-top boxes, dvd-players, and network gateways. These devices are resource (CPU, memory,

---

\textsuperscript{1} The project is funded in part by the European ITEA program. This is a joint project of various European companies, together with private and public research institutes.
bandwidth, power, etc.) constrained and the applications they support, typically have real-time requirements. The component model used within Robocop will be introduced in section 2.

Resources in embedded systems are relatively expensive and (usually) cannot be extended during the lifetime of the system. Consequently, an economical pressure exists to limit the amount of resources, and applications have strong constraints on resource consumption. In contrast to the fixed nature of available resources, software is subject to change. For example, new applications / components can be added, or existing ones can be replaced or removed.

To make sure that a combination of applications fits on a particular target system, knowledge about their resource consumption is required. Access to this information at design-time can help to prevent resource conflicts at run-time. This helps to decrease development time and, consequently, leads to a reduction of development costs.

However, the current techniques for addressing non-functional aspects of a system during the early phases of software development are laborious. In this paper we describe a method for estimating resource properties for component-based systems, using models of the behavior and the resource consumption of individual components. Key features of our approach are that the prediction method can be used in the early phases of application development, is intuitive and requires little effort.

We consider applications that may consist of multiple tasks. Tasks may run across multiple components. We associate a behavior model with each component. A behavior model of a task is obtained by composing the behavior models of individual components. We associate a resource model with operations of components. Using these behavior models for the tasks, the resource model for operations and a scheduling function we are able to predict the resource consumption for the system.

We propose a scenario-based prediction method in order to avoid the combinatorial complexity of full state-space analysis of systems, as is encountered by model-checking approaches [2]. In our approach resource estimations are made for a set of scenario’s that represent plausible/critical usages/executions of the system. Similar to the use of scenario’s in the evaluation of software architectures [12], the validity of the results of our analysis method depends on the selection of the scenario’s that are used.

The paper is structured as follows. In Section 2 we discuss the Robocop component model. In Section 3 we make a classification of different resource types. In Section 4, we present the theory for scenario based resource prediction. In Section 5 we show how to apply our method in practice, by showing an example. Section 6 discusses related work and draws some conclusions.

2 The Robocop Component Model

The Robocop component model is inspired by COM, CORBA and Koala [13]. A Robocop component is a set of models. Each model provides a particular type
of information about the component. Models may be in human-readable form (e.g. as documentation) or in binary form. One of the models is the ‘executable model’, which contains the executable component. Other examples of models are: the functional model, the non-functional model (modeling timeliness, reliability, memory use, etc.), the behavior model, and the simulation model. The Robocup component model is open in the sense that new types of models can be added.

A component offers functionality through a set of ‘services’. Services are static entities that are the Robocup equivalent of public classes in Object-Oriented programming languages. Services are instantiated at run-time. The resulting entity is called a ‘service instance’, which is the Robocup equivalent of an object in OO programming languages.

A Robocup component may define several interfaces. We distinguish ‘provides’ interfaces and ‘requires’ interfaces. The first defines the operations that are implemented by a service. The latter defines the operations that a service needs from other services.

In Robocup, as well as in other component models, service interfaces and service implementations are separated to support ‘plug-compatibility’. This allows different services, implementing the same interfaces, to be replaced. As a consequence, the actual implementations to which a service is bound do not need to be known at the time of designing a service. This implies that resource consumption cannot be completely determined for an operation, until an application defines a specific binding of the requires interfaces of service instances to provides interfaces.

Within the Robocup component model tasks are not confined to services. Tasks may, but do not need to, cross service boundaries. An example of a system built up out of multiple services with tasks that run across service boundaries is given in figure 4.

3 Resource Categories

We model different types of resources in different manners. First of all we distinguish processing resources and non-processing resources. Processing resources are resources that process elements; for instance statements (CPU), packets (Network) and samples (Sound card). Other resources are classified as non-processing resources. An example of a non-processing resource is memory. Secondly we distinguish pre-emptive resources and non-pre-emptive resources. Non-pre-emptive resources are claimed by a consumer and can only be used again when they are not needed any more by the specific consumer. The use of a pre-emptible resource may be interrupted after the necessary budget has been allocated to the consumer. Figure 1 illustrates that pre-emptive resources can be used by multiple consumers at the same time and non-pre-emptive resources can only be used by one consumer at any specific moment in time. Furthermore figure 1 shows that non-processing resources are requested and released by a consumer and processing resources are requested and used until processing is completed. Different types of resources require different type of specifications mechanisms
for the usage of the resource. Non pre-emptive resources have a claim mechanism. Pre-emptive resources have a mechanism to indicate the desire to use them (hopefully the resource manager will grant the wish). Non-processing resources have start-using:stop-using mechanism, processing resources only have a start:using mechanism (it is free again when it has processed all the data), there is no stop mechanism. However sometimes there is a notion of a deadline. Table 1 describes how we specify the resource use for different resource types. In this specification we assume resources are countable. The variables $X$ and $Y$ represent a amount of resources.

\[\text{Table 1. Resource Specification.}\]

<table>
<thead>
<tr>
<th></th>
<th>Processing</th>
<th>Non-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-emptive</td>
<td>Require: X</td>
<td>Require: X, Release: Y</td>
</tr>
<tr>
<td>Non-pre-emptive</td>
<td>Claim: X</td>
<td>Claim: X, Release: Y</td>
</tr>
</tbody>
</table>

4 Resource Prediction Method

We want to determine the resource usage of component-based systems during their execution. This run-time resource use depends on the composition of services that form a system and the operations executed. In this section we describe an approach for predicting the resources that a system uses for a specific scenario built out of multiple tasks. Tool support has been developed for this approach in order to automate prediction of resource consumption.

This section is structured as follows. In subsection 4.1 we discuss the specification of a service. In subsection 4.2 we show how to determine the set of services needed by an application. Subsection 4.3 shows how we obtain the resource consumption of a specific operation of a service. In subsection 4.4 we discuss how to model resource consumptions for call sequences. Finally we consider the resource measuring per task in subsection 4.5 and per scenario in subsection 4.6.
4.1 Service Specifications

Our approach considers composition of services. Each service has one or more provides interfaces and one or more requires interfaces. A provides interface lists the operations that the service offers to other services. A requires interface lists the operations that a service needs in order to function. For the purpose of resource prediction, the resources that are claimed and released are specified per operation.

We propose service specifications to capture such information about services. A service specification contains the following information:

- `provides interface`: A list of interface names;
- `requires interface`: A list of interface names;
- `Resource consumption`: For each implemented operation, the resource usage is specified. The specification of the resource consumption depends on the type of resource (See table 1);
- `Behavior`: For each implemented operation, the sequence of operations it uses.

Figure 2 contains an example service specification. It is a specification for service `s1` which uses the interfaces `I2` and `I3` and provides the interface `I1`. Service `s1` implements the operation `f` which uses operations `g` and `h` from interface `I2` and `I3`, respectively. Operation `f` requires 5000 cpu cycles. Furthermore we can see that operation `f` claims 100 bytes of memory and on return, releases 100 bytes.

The behavior section of the service specification defines that operation `g` from interface `I2` is called zero or more times before operation `h` from interface `I3`, and that it is called a varying number of times. How such a sequence of calls will be instantiated, in order to predict memory consumption, will be discussed shortly. All resource consumption in our approach, is specified per operation. Therefore, resources consumed during service instantiation and resources released at service destruction are specified as part of constructor and destructor operations. The

```plaintext
service s1
requires I2
requires I3
provides I1 {
    operation f
    resource cpu
        require 5000
    resource memory
        claim 100
        release
    behavior:
        I2.g*;
        I3.h}
```

**Fig. 2.** An example service specification.
behavior of an operation can be depicted as a message sequence chart (MSC) [11]. Observe that these are partial message sequence charts, because indirect operation calls (for instance operations that are called by $I_3.h$ in Figure 2) are not specified in the service specification.

The service specifications can be defined at design time, or can partly be generated from an implementation. In the latter case, call graph extraction can be used to obtain the behavior and to determine the requires interface. In case a service specification was defined at design-time, service specification generation can serve to validate an implementation (i.e., to check whether the service specification, extracted from the implementation, corresponds to the one defined at design-time).

## 4.2 Composition of Services

As illustrated in Figure 2, services can only use operations from interfaces, not directly from services. In other words, in our component model, a service cannot define a dependency upon a specific service. At development- or run-time the decision is made which services to bind to each required interface.

Two services that have the same provides interface may have different requires interfaces. As a consequence, the behavior of an application is not known before all requires interfaces of its services are bound. Hence, this behavior is needed for making estimations of the resource use.

To construct the behavior model of an application, we have to determine and bind the required interfaces. Finding the required services works by transitive replacing required interfaces by services. To describe this process of composition formally, we define a function $C$. Given a set of required interfaces and service specifications, functions $C$ yields a composition of needed services. The input domain of $C$ is a set of interfaces. In this paper we take the output of $C$ a set of service names. In general the output of $C$ may contain more information about the bindings between the services. Function $C$ is defined inductively next. First we introduce some auxiliary notation:

\[ s.provides = \text{set of interfaces provided by } s \]
\[ s.requires = \text{set of interfaces required by } s \]
\[ f.calls = \text{ordered set of functions called by function } f \]

\[ C(I_1, \ldots, I_n) = C(I_1) \cup \ldots \cup C(I_n) \quad (1) \]
\[ C(I) = s \cup C(I') \text{ for a service } s \quad (2) \]

where $I \in s.provides$ and $I' = s.requires$

If interfaces are implemented by multiple services, the set of services might be ambiguous. In this case there are (at least) the following options:
The composition function $C$ needs assistance to choose between appropriate candidate services. Assistance can be in the form of user intervention (i.e., a person chooses among candidates), in the form of composition constraints, or composition requirements (for instance, a decision is made based on maximal performance, or minimal memory consumption requirements).

The system generates all possible configurations. This can be used to compare their performance in the next phase.

### 4.3 Per-operation Resource Measurement

The set of services established by $C$, is the basis from which we will be able to deduce the behavior of specific scenarios can be constructed. Using this set we will determine the resource use of called operations. To that end, we define a weight function $w$ using ‘resource combinators’. This function calculates consumption of a resource $r$ of an operation $f$ over a composition of services, by accumulating the resources of $f$ itself, and of the resource of all the operations that are called by $f$. This means $w$ will compose the behavior triggered by operation $f$ (see figure 3).

\[ w(f(s), r) = (a) \text{ where } a = s.f.r.require \text{ and } r = \text{pre-emptive and processing} \]  \hspace{1cm} (3)

\[ w(f(s), r) = (a, b) \text{ where } a = s.f.r.require \text{ and } b = s.f.r.release \text{ and } r = \text{pre-emptive and } \neg \text{processing} \]  \hspace{1cm} (4)

\[ w(f(s), r) = (a) \text{ where } a = s.f.r.claim \text{ and } r = \neg \text{pre-emptive and } \text{processing} \]  \hspace{1cm} (5)

\[ w(f(s), r) = (a, b) \text{ where } a = s.f.r.claim \text{ and } b = s.f.r.release \text{ and } r = \neg \text{pre-emptive and } \neg \text{processing} \]  \hspace{1cm} (6)

Fig. 3. Composition of behavior.

The definition of this function depends on the type of resource (see figure 1). Function $w$ is defined as:

\[ w(f(s), r) = (a) \text{ where } a = s.f.r.require \text{ and } r = \text{pre-emptive and } \text{processing} \]  \hspace{1cm} (3)

\[ w(f(s), r) = (a, b) \text{ where } a = s.f.r.require \text{ and } b = s.f.r.release \text{ and } r = \text{pre-emptive and } \neg \text{processing} \]  \hspace{1cm} (4)

\[ w(f(s), r) = (a) \text{ where } a = s.f.r.claim \text{ and } r = \neg \text{pre-emptive and } \text{processing} \]  \hspace{1cm} (5)

\[ w(f(s), r) = (a, b) \text{ where } a = s.f.r.claim \text{ and } b = s.f.r.release \text{ and } r = \neg \text{pre-emptive and } \neg \text{processing} \]  \hspace{1cm} (6)
\[ w(f(s_1, \ldots, s_n), r) = w(f(s_i), r) \odot \left( w(f_1(s_1, \ldots, s_n\backslash s_i), r) \oplus \ldots \oplus w(f_k(s_1, \ldots, s_n\backslash s_i), r) \right) \]

where \( s_i \) provides \( f \) and
\[
(f_1, \ldots, f_k) = s_i.f.calls
\]

The following example illustrates how the functions \( C \) and \( w \) compose the behavior triggered by operation \( f \), following the example depicted in figure 3. The first step is establishing the set of services that form the implementation for operation \( I1.f \).

\[
w(I1.f(C(I1)), r) = w(I1.f(s_1, s_2, s_3), r)
\]

Using this set we are able to compose the behavior of \( I1.f \) in 2 steps:

\[
w(I1.f(s_1, s_2, s_3), r) = w(I1.f(s_1), r) \odot \left( w(I2.g(s_2, s_3), r) \oplus w(I3.h(s_2, s_3), r) \right)
\]

\[
w(I2.g(s_2, s_3), r) = w(I2.g(s_2), r) \odot \left( w(I3.i(s_3), r) \right)
\]

Considering resource usage of the execution of operations \( o_1 \) and \( o_2 \), we distinguish the following cases:

- \( o_1 \) and \( o_2 \) are executed sequentially: This means that first operation \( o_1 \) is executed and after \( o_1 \) terminates, then \( o_2 \) is executed. An example of two operations that are executed sequentially are \( I2.g \) and \( I3.h \) in the implementation of interface \( I1 \) by service \( s_1 \) (see figure 3).
- \( o_2 \) is executed as part of the execution of \( o_1 \): This means that first operation \( o_1 \) is executed. Before \( o_1 \) terminates, as part of the behavior of \( o_1 \), \( o_2 \) is executed. An example of an operation that is executed as part of another operation is \( I3.i \), which is executed as part of the the behavior of operation \( I2.g \) implemented by service \( s_2 \) (see figure 3).

In these different cases we want to combine the resource usage of the operation executions in different ways. In order to be able to combine the resource usage of operation executions appropriately for each of the cases mentioned before, we introduce a resource combinator for each case. The resource combinator \( \oplus \) is used to model resource usage of two sequentially executed operations. The
combinator \( \odot \) is used to model resource usage as part of an operation. The definitions of these combiners may be varied, in order to deal with different types of resources or to measure resource usage in different ways. We will now present two straightforward example definitions for the resource combiners. In [5] we considered memory usage and defined these operators as follows:

\[
(a, b) \oplus (c, d) = (a - b + c, d) \quad \text{if } c \geq b \\
(a, b + d - c) \quad \text{if } c < b \\
(a, b) \odot (c, d) = (a + c, b + d)
\]

In section 5 we describe an example concerning CPU usage. In that example we will use the following definitions for the resource combiners:

\[
\begin{align*}
    a \oplus b &= a + b \\
    a \odot b &= a + b
\end{align*}
\]

Now we combine the composition function \( C \) and weight function \( w \), to form the weight function \( W_{op} \), that calculates resource consumption that results from executing an operation of an interface. It is defined as:

\[
W_{op}(f(I), r) = w(f(C(I)), r)
\]

for interface \( I \) and operation \( f \) (8)

This function estimates the resource consumption of resource \( r \) of an operation call \( f \) from interface \( I \). The function first determines a set of services that (together) form an implementation of \( f \). Then, the weight function \( w \) is used to predict resource consumption of \( f \) and all operations that are transitively invoked by \( f \).

4.4 Modeling Resource Consumption for Call Sequences

Each operation call that is specified in a service behavior specification may be accompanied with an iterator (‘%’) to indicate that the operation maybe called multiple times. For the purpose of resource estimation, we represent an iterated operation call as a lambda expression:

\[
f\% = \lambda l \rightarrow l \times f
\]

This lambda expression states that operation \( f \) is called \( l \) times, where \( l \) is variable. The weight function for an iterated operator call can now also be defined as a lambda expression:

\[
w(\lambda l \rightarrow l \times f(s)) = \lambda l \rightarrow l \times w(f(s)) = w(f(s))_1 \oplus \ldots \oplus w(f(s))_l
\]

(10)

Such lambda expressions are instantiated with concrete numbers, when resource consumption for a complete scenario is predicted.
4.5 Per Task Resource Measurement

As part of the resource prediction for a scenario we need to be able to predict resource consumption for a sequence of operation calls. We introduce the notion of a task, which is a sequence of operation calls. A task can be continuous or terminating, a continuous task corresponds to a repeating sequence of operation calls and a terminating task corresponds to a finite sequence of operation calls.

Using function $W_{op}$ we are able to estimate resource consumption for individual operation calls, we need to predict resource consumption of a task. To that end, we need to know what plausible sequences of operation calls are. Such sequences (which we call task definitions), are determined together with implementors of services. If iterated operations are called in a task definition, the task definition becomes a lambda expression. Defining a task therefore consists of the following two steps:

1. Defining a plausible sequence of operation calls;
2. Instantiating concrete numbers for the lambda expressions, if any.

Instantiating a lambda expression fixes the number of operation calls. Thus, instantiating a lambda expression $\lambda l \to l \times f$ with, say, 2 yields a sequence of two calls to $f$.

We can now define the weight function $W_t$ for a task, consisting of sequential operation calls. This function computes for each task, e.g. a sequence of operations, the required resources.

$$W_t(f, r) = W_{op}(f, r)$$  \hspace{1cm} (11)

$$W_t(f; g, r) = W_t(f, r) \oplus W_t(g, r)$$  \hspace{1cm} (12)

4.6 Scenario-Based Resource Measurement

In the Robocop component model, multiple components can work on different tasks that are executed in parallel. A scenario is defined as a set of tasks. This set contains the tasks that are executed in parallel, and for which the resource estimation needs to be made.

We can now define the weight function $W$ for a scenario $S$ and resource $r$, consisting of multiple tasks. This function computes for each task the estimated resource consumption:

$$W_s(\{t\}, r) = W_t(t, r)$$  \hspace{1cm} (13)

$$W_s(\{t_1, \ldots, t_n\}, r) = W_s(\{t_1\}, r) \otimes \ldots \otimes W_s(\{t_n\}, r)$$  \hspace{1cm} (14)

$$W(S, r) = TS(W_s(S, r))$$  \hspace{1cm} (15)

The function $W$ introduces the functions $W_s$ and $TS$ and a new resource combinator $\otimes$. This resource combinator is used to combine resources that are consumed in parallel. The resources can be combined only after we know what type of scheduling is used. Therefore the introduced resource combinator $\otimes$ is abstract. It is the task of the function $TS$ to do the scheduling and transform the expression in an expression which only contains concrete operators.
In the next section we will demonstrate the use of the weight functions \( w, W_{op}, W_t, W_s, \) and \( W \) in an example.

5 Example

In this section we will discuss a small example to illustrate how the method works. We will present a design of a media player build out of 9 components, a FileReader, a Decoder, a AudioDecoder, a VideoDecoder, a AudioRenderer, a VideoRenderer and 3 Buffer components.

The media player application has 4 tasks. A task \((Tr)\) to do the reading from file, one \((Td)\) to decode the video and audio data, one \((Ta)\) to render audio and one \((Tv)\) to render video. The system described above is illustrated by figure 4. The service definitions can be found in figure 5.

In our example we consider a mediaplayer running multiple continuous tasks. We consider the scenario illustrated in figure 4 and are interested in the resource cpu.

\[
S = \{Tr, Td, Ta, Tv\}
\]
\[
r = \text{cpu}
\]

We want to predict the cpu usage of scenario \( S \).

\[
W(S, \text{cpu}) = TS(W_s(S, \text{cpu}))
\]
\[
= TS(W_s(\{Tr, Td, Ta, Tv\}, \text{cpu}))
\]
\[
= TS(W_t(Tr, \text{cpu}) \otimes W_t(Td, \text{cpu}) \otimes W_t(Ta, \text{cpu}) \otimes W_t(Tv, \text{cpu}))
\]

Now we are going to use the task definitions and the scheduling function. We consider Round Robin scheduling. This means that if a task \( A \) requires 100 cycles for a loop and there are \( n \) tasks, \( A \) will have performed a loop each \( n \times 100 \) cycles. The tasks and the scheduling function are defined as follows:

\[
TS(A_1 \otimes \ldots \otimes A_n) = (B_1, \ldots, B_n)
\]
where \( B_i = A_i \times n \)
These definitions enable us to evaluate $W(S, r)$ further. We now have the following intermediate result:
\[ W(S, cpu) = (W_{op}(I\text{FileReader}.Read, cpu) \times 4, \]
\[ W_{op}(I\text{Decode}.Decode, cpu) \times 4, \]
\[ W_{op}(I\text{AudioRenderer}.Render, cpu) \times 4, \]
\[ W_{op}(I\text{VideoRenderer}.Render, cpu) \times 4) \]

We will now show how \( W_{op}(I\text{Decode}.Decode, cpu) \) is evaluated. This means a set of services that implement \( I\text{Decode}.Decode \) will be generated, the behavior of all operations called for the implementation of \( I\text{Decode}.Decode \) will be generated. We will use the following abbreviations \( D = \text{Decoder}, Vd = \text{VideoDecoder}, Ad = \text{AudioDecoder} \) and \( B = \text{Buffer} \).

\[ W_{op}(I\text{Decode}.Decode, cpu) = w(I\text{Decode}.Decode(C(I\text{Decode}.Decode)), cpu) \]
\[ = w(I\text{Decode}.Decode(D, Vd, Ad, B), cpu) \]
\[ = w(I\text{Decode}.Decode(D), cpu) \odot \left( w(I\text{Buffer}.Get(Vd, Ad, B), cpu) \oplus w(I\text{AudioDecoder}.Decode(Vd, Ad, B), cpu) \oplus w(I\text{VideoDecoder}.Decode(Vd, Ad, B), cpu) \right) \]

To continue the example we show how \( w(I\text{Buffer}.Get(Vd, Ad, B), cpu) \) evaluates in the next step.

\[ w(I\text{Buffer}.Get(Vd, Ad, B), cpu) = w(I\text{Buffer}.Get(B), cpu) \]
\[ = 50 \]

After we evaluate the other \( w \) functions we get the following result.

\[ W_{op}(I\text{Decode}.Decode, cpu) = 100 \odot \left( 50 \oplus (5000 \odot (\lambda l \rightarrow l \times 50)) \oplus (20000 \odot (\lambda k \rightarrow k \times 50)) \right) \]

Before we can continue evaluating this result we need to instantiate the two lambda expressions. We assume decoding one data packet results in 10 packets for the audio buffer and 5 for the video buffer. Therefore we instantiate the variable \( l \) representing the number of Puts in the audiobuffer with 10 and the variable \( k \) representing the number of Puts in the videobuffer with 5.

The last thing we need for the evaluation of \( W_{op}(I\text{Decode}.Decode, cpu) \) is the definitions of the resource combinators. During our calculations we use the following resource consumption operator definitions:

\[ a \oplus b = a + b \]
\[ a \odot b = a + b \]
At this time we are able to evaluate $W_{op}(I\text{Decode} \cdot \text{Decode}, \text{cpu})$ completely.

$$W_{op}(I\text{Decode} \cdot \text{Decode}, \text{cpu}) = 25900$$

After evaluating the other $W_{op}$ functions we get the required cpu cycles required for the execution of one loop for the individual tasks in scenario $S$:

$$W(S, \text{cpu}) = (1050 \times 4, 25900 \times 4, 250 \times 4, 1250 \times 4)$$

6 Concluding Remarks

6.1 Discussion

Until now, we applied our prediction method only to a small application. However, within the Robocop project, we have access to real-world applications. Therefore, we are at present applying our method to industrial software systems of some of our partners. The experiments that we have done thus far are very promising.

Our prediction technique has the following advantages:

- The method fits very well with current design approaches for reactive systems such as UML [8] or [9] where the dynamics of systems are modelled using scenarios. Hence, the effort needed for modelling the resource use in addition to an existing specification is limited. Furthermore, the modelling of scenarios typically happens early in the development process. Hence, our approach can be used to obtain feedback during the early stages of design.
- The method is compositional: results about the resource of a system can be computed based on specifications that are provided for its constituent components.
- The method is general. It can be applied to different types of resources and for different types of estimations. This is achieved by allowing different definitions of resource combinators.

The approach also has some limitations that need further research:

- We assume that consumption is constant per operation, whereas it typically depends on parameters passed to operations and previous actions. Extending the method in this respect is future work;
- Our scenario-based approach depends on the ability to define realistic runs. This makes our approach dependent on the quality of an architect;
- The model presented does not deal with synchronization between tasks.

6.2 Related Work

The development of methods and techniques for the prediction of properties of component based systems is a notoriously hard problem and has been the subject
of recent OOPSLA and ICSE workshops. A guideline for research projects in predictable assembly is proposed in [4].

An interesting approach based on queueing networks is presented in [1]. This approach combines performance modelling using queueing networks with software modelling using UML. A notable difference with our approach is the way in which a model of the overall system behaviour is obtained. In our approach, this is assembled from partial behaviour specifications, instead of taking the overall system behaviour as input for the prediction process.

An example approach that addresses the prediction of end-to-end latency is presented in [10]. Based on our experience, we can subscribe the statement of Hissam et al. that it is necessary that prediction and analysis technology are part of a component technology in order to enable reliable assembly of component-based systems. However, there are many alternative ways in which the two may be technically related across the different phases of the component life cycle.

Prior work of a predictable assembly approach for estimating static resource use (using a Koala-like component model) is described in [7]. This builds on earlier work on prediction of static memory consumption described in [6].

The area of software architecture evaluation is dominated by scenario-based approaches [3,12]. The disclaimer of these approaches is that the quality of their results depends on the representativeness of the scenario’s chosen. The same disclaimer applies to the approach we propose in this paper.

The main differences of our approach are that we use a formal model: our method addresses dynamic instead of static resource consumption, supports resource consumption analysis in the early phases of software development and prediction requires little effort.

### 6.3 Contributions

In a previous paper [5] we presented a first approach for predicting resource use of component-based systems. This paper extends the approach such that it is able to deal with concurrent tasks that share system resources. To support this, we extended our approach with a means of mapping logical scenario’s to tasks. Subsequently, these tasks are subject to system-wide resource scheduling policies. We have shown how these global scheduling policies can be taken into account into the component-based prediction technique. We conclude that for doing predictions for multitask systems, it does not suffice to have only component-level information, but that also system-wide resource sharing policies need to be taken into account.

In the previous paper we focussed on memory-use. In this paper we focussed on CPU-use. This led to the need to classify resources into different categories (pre-emptive vs. non-pre-emptive x processing vs non-processing). For each category, different formula are needed for combining different types of resources.

We have illustrated our approach to a case study that is closely inspired by an actual industrial system.
References

Design Accompanying Analysis of Component-Based Embedded Software

Walter Maydl
University of Passau, Innstr. 33, 94032 Passau, Germany
maydl@fmi.uni-passau.de
http://www.fmi.uni-passau.de/~maydl/

Abstract. Design accompanying analysis techniques for component-based embedded systems based on the dataflow paradigm are presented. The underlying signal model covers not only the value range and the time domain but also attributes of the signal data transport. Components are modelled as functions on streams of signal data. This allows to describe the behavior of dataflow components precisely by constraints. Static constraints, e.g., equality of sampling periods, may be as complex as multivariate polynomials and are enforced by a new interface type system. Dynamic constraints, e.g., describing communication protocols, are checked using a novel model checking technique based on fifo automata. The objective of these mathematically well-founded analysis techniques is to detect as many program errors as possible during design. Moreover, the component model is compositional resulting in well-defined hierarchical abstraction. Alltogether, this results in a more reliable development of complex applications in a shorter design time.

1 Introduction

There is currently a trend for research towards design and analysis of increasingly complex component software for embedded systems [1,2]. One goal is early error detection avoiding costly redesigns. To achieve this objective, design accompanying analysis techniques are proposed to minimize the amount of errors reported to the user. These techniques are used incrementally, if possible, reusing previous results to reduce the computation time needed. Automatic error correction (e.g., automatic lossless type conversion) minimizes further the number of unnecessary error messages. Another goal is well-defined hierarchical abstraction which is an important design principle for constructing large-scale component systems. Therefore, a compositional component model is presented which prevents errors like artificial deadlocks. This derivation of properties of a component system from given properties of single components and rules for their interaction is one of the main problems of component-based software development [3].

These new analysis techniques are based on a novel and very general signal model allowing the efficient description of infinite physical signals consisting of (1) equidistantly sampled signal values, (2) segments (composed of equidistantly sampled relevant signal values) and “pauses” in-between (dead times), e.g., signals measured at an assembly line, acoustic signals like speech, neural signals,
etc., or (3) non equidistantly sampled signal values (events), e.g., key strokes. Another aspect of this signal model is efficient signal data transport which is important for the distributed execution of the dataflow component system [4].

This new signal model leads to the definition of a novel model for dataflow components and new dataflow paradigms. This new component model fits well into Szyperski’s definition of a component (see below). Static constraints (e.g., equality of sampling periods) and dynamic constraints (e.g., communication protocols) which have to be satisfied for the component and the resulting component system to be well-defined are derived from that component model.

To address these constraints, a novel type resolution algorithm and a new model checking technique are invoked during each design step to guarantee early feedback. Interface types and the static type constraints are used as input to the type resolution algorithm. Problems detected by the interface type system are, e.g., memory overflow, premature program termination, or faulty calculations (possibly dangerous to the environment). Moreover, the interface type system is very flexible by allowing polymorphism, overloading, automatic type conversion, etc. The model checker constructs a model of the dataflow graph using fifo automata. This model is used to check dynamic constraints regarding protocol compatibility. Deadlock detection and avoidance, the calculation of memory usage and cyclic schedules are important aspects of this analysis. Furthermore, protocol adaptation is possible.

Section 2 sets out related work. In Section 3, the new signal model is introduced. Section 4 is focussed on the new component model and the extension of “classical” dataflow paradigms. Section 5 introduces the new type system and model checking mechanisms. In Section 6, runtime tests of these analysis techniques are depicted. Section 7 summarizes the presented work.

2 Related Work

This section highlights on related work regarding the signal model, dataflow paradigms, type systems, and model checking.

Signal Model. In [5], a signal model for different models of computation is introduced. A signal is described as a collection of events (value-tag pairs). In current dataflow paradigms, the tags only indicate the order in which events are consumed and produced by a dataflow component. These tags are represented indirectly by the order of the events in the fifos. No physical time information is assigned to these events. That is, the input data is represented as an unstructured stream of data values encapsulated by tokens (data containers) [6,7].

Dataflow Paradigms. For this kind of streams, a hierarchy of dataflow paradigms is defined [6,7]: SYNCHRONOUS DATAFLOW (SDF) represents the basic paradigm which consists of atomic components and edges connecting them (cf. Fig. 1). To each input and output edge of a component a fixed integer marking (weight) is assigned. A component having as many tokens at its input edges as the assigned weights state is enabled. Invoked by a scheduler, it consumes and produces as many tokens as the marking demands. By adding the components
SWITCH and SELECT (cf. Fig. 2 a and 2 b), SDF is extended to the BOOLEAN CONTROLLED DATAFLOW (BDF) paradigm. These components have conditional input or output edges. The control token causes the select component, for instance, to pass the data token of the corresponding input edge to the output edge. Switch is defined in a similar way. DYNAMIC DATAFLOW (DDF) adds the MEREGE component (cf. Fig. 2 c) to BDF. If a token arrives at any of its input edges, merge passes it on to its output edge. If there are tokens at both of its input edges, they are passed on randomly.

SDF is computationally weaker than the Turing machine. Thus, many interesting questions like presence of deadlocks, the calculation of a cyclic schedule, or the amount of memory needed can be determined for a pure SDF graph [6,7]. BDF and DDF are Turing equivalent [6]. That is, the above stated questions are not decidable any more [6,8,9].

The operational semantics of these dataflow paradigms is not well-defined. For instance, a hierarchical SDF component is expected to behave like an atomic SDF component. This may lead to ARTIFICIAL DEADLOCKS [10,11]. These deadlocks are artificial because without hierarchy no deadlock occurs. The analysis carried out is not INCREMENTAL as previous detected results are not reused [7]. As the data streams are just a sequence of values, several errors, e.g., regarding the time domain, cannot be detected due to the lack of information [4].

**Type System.** Type systems of functional programming languages (like Haskell and ML) and applications of type system concepts to embedded system design like in Ptolemy II [12] represent the starting point of this new interface type system. Parametric polymorphism or polymorphism defined on type classes (overloading) [13,14], higher-order functions, and static type checking [15] at the end of program design are important aspects in functional programming. In Ptolemy II, a type system for embedded systems is specified [16]. A lattice is used to model the lossless type conversion relation among types and type constraints are defined by inequalities. Polymorphic typing of dataflow components and automatic lossless type conversion (e.g., a cast from int to double) at runtime are supported. Ptolemy II represents the current state of the art of type system design for this application area.

**Model Checking.** The component model for the SDF paradigm presented here was derived from analysis and design methods for computer protocols based on
COMMUNICATING FINITE STATE MACHINES (CFSMs), e.g., used in PROMELA [17,18]. Important aspects of protocol design are formal validation, protocol synthesis, and conformance testing. However, due to the use of unbounded fifos, many questions become undecidable. In Ptolemy II and CAL, INTERFACE AUTOMATA [19] are used to model the behavior of software components and the interaction between different models of computation [16,20]. As there are no fifos included in that formalism, no ASYNCHRONOUS communication can be described. REGULAR STATE MACHINES [21] are another approach to model state transition systems like SDF graphs. This approach does not incorporate colored tokens. Moreover, none of the mentioned techniques is design accompanying. There are also several model checking tools defined for the verification of properties of traditional programming languages [22,23].

3 A New Signal Model

An embedded system is usually designed by using different models of computation (cf. Fig. 3) [1,12,24]. Dataflow graphs are responsible for signal processing and finite state machines control the dataflow graph or deal with user interaction, for instance. The dataflow graph is a combination of dataflow components encapsulating parameterized algorithms (such as digital filters, FFT, PID-controllers, etc.) connected by edges (fifo channels) to solve a complex application problem. The embedded system may consist of realtime parts (e.g., responsible for controlling a plant) and non-realtime parts (e.g., gathering of statistics) [25]. Tasks associated with these parts may be periodic, aperiodic, or sporadic. The
periodicity of a task depends on the arrival times of the blocks (see below) but not directly on the sampling period. The execution of the embedded system is asynchronous and distributed.

**Analog and Digital Signals.** Usually, input signals of an embedded system are analog (cf. Fig. 4 a) and have to be sampled and quantized. Thus, a signal

---

**Fig. 4.** Steps Towards a New Signal Model.
\[
\begin{array}{ll}
\text{point set } P & \rightarrow \text{block set } B \\
\quad f_P : P \times P \rightarrow P \cup \{\bot\} & \quad f_B : B \times B \rightarrow B \cup \{\bot\} \\
\quad f_P((t, x), (t', x')) = & \quad f_B(b, b') = \\
\begin{cases} 
(t, x + x') & \text{if } t = t' \\
\bot & \text{otherwise}
\end{cases} & \begin{cases} 
f_B((t, x) \bullet b, (t', x') \bullet b') = 
\quad f_P((t, x), (t', x')) \bullet f_B(b, b') \\
\text{otherwise: } f_B(b, b') = \bot
\end{cases}
\end{array}
\]

Fig. 5. Denotational Semantics of a Dataflow Component (Pointwise Combination).

\( f : T \rightarrow X \) may be depicted as an infinite sequence of time-value pairs (cf. Fig. 4 b). The time domain \( T \) is an infinite set of equidistant points in time. The value range is described by a value set \( X \) and a physical unit \( pu \) such as Ampère, Newton, Watt, etc. \( X \) may be of any basic type such as \( bool \) or \( double \) or of any aggregation type such as array or record.

**Signals as Sequences of Segments.** In many applications, only those signal values are relevant that are, e.g., greater than some threshold \( x_{th} \). The dead times in-between are discarded. Therefore, the original signal is split up (e.g., by some plug-in card) into several segments containing only signal values \( x_i > x_{th} \) (cf. Fig. 4 c). This new signal model is able to represent all three kinds of infinite signals depicted in Fig. 3. Signals of kind (3) are equivalent to an infinite sequence of finite segments consisting of just one time-value pair.

**Signals as Sequences of Blocks.** Segments are further subdivided in equally sized blocks (see Fig 4 d). There are several reasons for that:

- First of all, this subdivision results in the optimal amount of signal data for transportation in a distributed embedded realtime system. The transfer of single signal values leads to computational overhead and the transfer of complete segments is unfavorable as the reaction time of the embedded system is too long and as the length of segments is often not predictable.
- Secondly, several dataflow components (e.g., encapsulating an FFT) need a block of signal data with a certain length for execution.
- Thirdly, the measurement hardware may produce blocks anyway.

**Signals as Streams of Colored Tokens.** A block marking (color) \( bm \) is assigned to each block (cf. Fig. 4 e) to distinguish different segments of a signal. The values of \( bm \) are: \( s \): starting block, \( m \): middle block, \( e \): ending block, \( o \): the only block of a single-block segment. If we abstract from the block contents, the combination of block and block marking is called (colored) token.

### 4 A Novel Component Model

**Szyperski’s Definition.** Szyperski defines a component as follows [3]: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be
deployed independently and is subject to composition by third parties.” As a corollary of this definition, he states that “software components are ‘binary’ units that are composed without modification.” Compared with various other definitions [2,3,26], this one is quite stringent. A dataflow component (cf. Fig 3) fits perfectly into Szyperski’s definition. Dataflow components are provided as precompiled units in form of shared libraries. Software engineers implement components and application engineers are responsible for component composition. The application engineers can purchase a subset of components necessary to fulfill their assignments.

**Denotational Semantics of a Dataflow Component.** The denotational semantics of a dataflow component is defined based on the different levels of abstractions of the signal model (time-value pair, block, and stream; see above). A dataflow component is a function $f_S$ on infinite streams $S$ of colored tokens. $f_S$ is based on the firing function (i.e., encapsulated algorithm) $f_B$ defined on the block set $B$. $f_B$ again utilizes the function $f_P$ defined on the point set $B$. As an example, the pointwise combination of blocks is defined as set out in Fig. 5. The overall semantics of a dataflow graph (component system) is the fixed point of the composition of the component functions applied to the input data [8,27].

**Well-Definedness of a Dataflow Component.** Constraints are derived from this functional description and are enforced during design to guarantee that dataflow components and component systems (cf. Fig. 3) are well-defined. Static constraints are concerned with the characteristics of tokens consumed at a certain point in time. If the dependencies between the different tokens on the incoming and outgoing interfaces are satisfied, the component is well-defined regarding these static constraints. Dynamic constraints, e.g., communication protocols, are concerned with the characteristics of token sequences.

**Novel Dataflow Paradigms.** The dataflow paradigms presented in Section 2 have to be extended to deal with colored token streams [25,28].

*Colored SDF.* In addition to the firing conditions of SDF, the colors of the incoming tokens have to fit the constraints of a protocol. Examples for constraints are that the number of block tokens of a signal segment has to be fixed, that it has to be a multiple of some integer value, or that the colors occur in a certain order (cf. Fig. 4 d). A component may be able to deal with several protocols.

*Colored BDF.* Colored switch and select are guiding all tokens of a signal segment in the same direction. In order to not destroy the segment structure, all blocks of a segment are transferred before blocks of another segment are considered. Colored BDF components may, e.g., collect all blocks of a signal segment, separate a signal segment consisting of one block in several smaller blocks, fill the shorter of two segments with dummy blocks, or cut the longer of two segments to adjust their overall length [25,28].

*Colored DDF.* Colored Merge does also preserve the segment structure. Merge may introduce nondeterminism into the dataflow graph [9].
5 Design Accompanying Analysis

A novel interface type system addressing static constraints and new model checking mechanisms for dynamic constraints are presented. The theoretical background is explained in [4,25,29].

5.1 Interface Type System

Definitions

1. A port of a dataflow component, e.g., \( x \) in Fig. 6, is called an INTERFACE [16].
2. The allowed values of an interface are determined by an INTERFACE TYPE. The set of all possible interface types is represented by the TYPE DOMAIN:

\[
ST \times SP \times VS \times PU \times BL \times BM.
\] (1)

3. The different parts of the cross product are called TYPE TRAITS which represent all signal characteristics mentioned in Section 3: starting times \( ST \), sampling periods \( SP \), value sets \( VS \), physical units \( PU \), block lengths \( BL \), and block markings \( BM \).

Fig. 6 depicts a dataflow component. The interface type \( \text{type}(x) \) of interface \( x \) — the analogon of a datatype — is an element of the type domain.

POLYMORPHISM is an important feature of this interface type system. The elements of the value set (\( \text{number}, [7; 14] \)) are, e.g., arrays of numbers of any integral or floating point type with array sizes between 7 and 14.

Static Type Constraints. A dataflow component consists of interfaces with associated interface types and a firing function (cf. Fig. 5 and 6). The firing function may impose type constraints on the different type traits of the incoming and outgoing streams: equality constraints (defined on all type traits), multiplication and division constraints (defined on physical units), multivariate polynomials (defined on sampling periods, array sizes, and block lengths). Each edge implies equality between the types of the connected interfaces.

Type Resolution

Type Traits as Lattices. All the type traits form lattices. A partially ordered set is a lattice if infimum and supremum exist for all pairs of its elements [30]. If an interface type \( x \) is less than another interface type \( y \) according to this order, it is said to be “less defined.”
**Type Resolution Algorithm.** Each change in the dataflow graph invokes the type resolution algorithm (cf. Fig. 7). The insertion of new dataflow components or edges results in the addition of new type constraints to the constraint set. The type resolution algorithm applies constraint propagation rules (see below) implied by these type constraints to the current interface types. As the type constraints connect several interface types, the update of an interface type may result in the update of a dependent type. That is, the type resolution algorithm traverses several other interfaces (possibly several times) until no interface types are updated any more. When new dataflow components or edges are added, the previously calculated interface types are reused. Deletion of components or edges imposes the restart of the calculation from the initial interface types.

**Constraint Propagation Rules.** The constraint propagation rule for equality of two interface types is equivalent to the calculation of their supremum. The constraint propagation rule for the other constraints are more challenging [4]. Multivariate polynomials are evaluated by applying the constraint propagation rules consecutively to two interface type variables at a time.

**Mathematical Description.** The type resolution algorithm is a functional \( \Phi_C : ( \text{ID} \rightarrow \text{DOM} ) \rightarrow ( \text{ID} \rightarrow \text{DOM} ) \) : \( \text{current types} \rightarrow \text{updated types} \) (2)

\( \Phi_C \) is applied repeatedly to the current interface types until no more changes of the interface types occur. Thus, \( \Phi_C \) delivers the least fixed point.

**Problems.** The termination problem results from the application of the type resolution algorithm to infinite lattices (e.g., necessary for the definition of aggregation types like nested records or arrays). This problem is solved by setting a limit for the number \( n \) of nestings. Although not satisfactory in theory, practically this solution is appropriate. Another problem is the weak constraints problem. The fixed point found by the type resolution algorithm satisfies the proposed type constraints. The constraints, however, are not strong enough. In case, unspecified interface types remain at the end of the design phase, either...
Fig. 8. Fifomata and Protocol Constraints.

the user has to set some parameters or an algorithm has to be started which tests the different remaining interface types whether they are valid solutions. The type resolution algorithm is a compromise between calculation time needed and exactness. Mostly, the interface types can be resolved. In special cases, the user has to intervene.

Type Conversion. Detected type conflicts may be resolved by type conversion. If no information will be lost (e.g., casting from int to double), automatic type conversion is possible. In the remaining cases, conversion suggestions are proposed to the user. These methods are also based on lattices (e.g., determining the new starting point \( s_3 = \max\{s_1, s_2\} \)). Type conversion on arrays is done by copying to avoid the well-known subtyping problem [16].

5.2 Model Checking

“Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite [31].”

Definitions. The firing function of a dataflow component is modelled as a fifomaton (fifo automaton) consuming colored tokens from and producing colored tokens to fifo channels (cf. Fig. 8).

1. A FIFO (MESSAGE QUEUE, CHANNEL) is a triple \( c = (M_c, n_c, w_c) \) where \( M_c \) is a finite set called the message alphabet. \( n_c \in \mathbb{N} \cup \{\infty\} \) is the fifo capacity. \( w_c \in \times_{c \in C} M_c^\ast \) is a tuple of words representing the fifo contents.

2. A FIFOMATON is defined as follows:

\[
\begin{align*}
F &= ((Q_m, Q_t), q_0, (C_i, C_o, C_h), T) \\
Q_m, Q_t &= \text{sets of MAIN and TRANSIENT STATES} \\
Q &= Q_m \cup Q_t, \text{ set of states} \\
q_0 \in Q_m &= \text{initial state} \\
C_i, C_o, C_h &= \text{sets of input, output, and hidden fifos} \\
C &= C_i \cup C_o \cup C_h, \text{ set of fifos} \\
M &= \bigcup_{c \in C} M_c, \text{ message alphabet} \\
T &= Q \times ((\{?\} \times (\times_{c \in C_i} M_c^\ast)) \cup (\{!\} \times (\times_{c \in C_o} M_c^\ast))) \times Q, \text{ set of transitions}
\end{align*}
\]

A fifomaton is called CANONICAL if \( C_h = \emptyset \).
3. **TRANSITIONS**: Relation \( T \) delivers for a current state \( q \) and an action \( a \) the successor state \( \hat{q} \). An action \( a \) may either be an input action \( c?m \) or an output action \( c!m \) where \( c \) denotes the fifo and \( m \) the message. “!” or “?” indicate a write or read operation, respectively.

4. An action \( a \) is **EXECUTABLE** if \( a \) is the empty action, \( a \) is an input action and \( m \) is a prefix of the corresponding fifo contents, or \( a \) is an output action and the updated fifo content does not exceed the capacity of the corresponding fifo \((n_c \leq \text{size}(w_c) + \text{size}(m))\).

5. During **EXECUTION**, all fifomata and all fifos are first set to their initial state or initial content, respectively. Then as long as possible, an arbitrary fifomaton \( i \) with an arbitrary executable transition rule is executed. Otherwise, the execution terminates.

6. Each fifomaton defines sequences of input and output actions representing a **COMMUNICATION PROTOCOL**. For the protocol to be suited for endless execution, the corresponding fifomaton may not contain dead ends.

As read and write operations have to be executed atomically, the fifomaton modeling the component behavior separates them using a transient state. This is necessary for modeling **FEEDBACK CONNECTIONS** in the dataflow graph.

**Dynamic Protocol Constraints.** The firing function of a dataflow component represented as a fifomaton may impose dynamic protocol constraints on the incoming and outgoing colored token streams. Examples for constraints are given in Section 4 (cf. Fig 8). Each edge connects two fifos and enforces the compatibility of the producing and consuming fifomaton.

**Model Checker**

**Model Checking Algorithm.** The model checking algorithm (cf. Fig. 9) is invoked each time changes in the dataflow graph occur. The insertion of new components or edges results in the amendment of a new fifomaton or the connection of two fifos, respectively. The model checking algorithm applies composition rules to determine a canonical fifomaton which is the basis of the analysis (see below). When new dataflow components are added, the previously calculated fifomata are reused. If fifomata or edges are deleted, the calculation has to start all over.
Composition Rules. The behavior of a COMPOSITION OF DATAFLOW COMPONENTS is given by the canonical fifo maton resulting from the composition of the single components. First, the PRODUCT FIFOMATON is computed. Then, fifos are CONNECTED. Finally, the resulting fifo maton is SIMPLIFIED by totally eliminating bounded fifos, pruning dead ends, and removing empty transitions [29].

Analysis. Using a canonical fifo maton, the following questions may be analyzed:

1. Protocol Compatibility/Deadlock detection: If colored tokens are transferred in the dataflow graph, complex communication protocols may be defined. If the composition of two fifo matata is not empty, their protocols are compatible. In case the composition is empty, a deadlock was detected.
2. Cyclic Schedules: The canonical fifo maton of a dataflow graph defines its cyclic schedule. By imposing additional constraints on the calculation of the transitions of the composed fifo matata, various optimizations may be obtained.
3. Memory Usage: The memory usage of a cyclic schedule can be derived from the canonical fifo maton explicitly modeling all possible contents of hidden fifos as part of the states.

Problems. The THEORY OF COMPUTABILITY [32] limits what can be decided by an algorithm. The colored dataflow paradigms introduced have an increasing computational complexity. The colored SDF paradigm is not Turing equivalent. That is, the bounds of the fifo capacities are determined by

\[ n_c = \text{least\_common\_multiple}(t_{source}, t_{dest}) + |w_c| \]  

(3)

The canonical fifo maton equivalent to all cycles found in the reachability tree can be constructed and analyzed. However, colored BDF and DDF are Turing equivalent. The construction of the canonical fifo maton (reachability tree) is not guaranteed to terminate due to possibly unbounded fifos. So, deadlock detection, calculation of memory usage etc. are not decidable/computable. To address this problem, the above mentioned dataflow paradigms form a hierarchy in which each new paradigm adds additional modeling capabilities and increases the computational complexity. The different dataflow components are each assigned to one dataflow paradigm suited to its computational complexity. This allows the designer to decide between analyzability and modeling power. Moreover, not all BDF or DDF graphs are Turing equivalent (e.g., due to a while-loop) and can still be analyzed. Otherwise, SDF subgraphs may be checked.

A closely related problem is the STATE EXPLOSION [33]. Even if the search space is finite, the number of states of the canonical fifo maton (reachability tree) may be huge. In case the time needed for model checking is too long, it may either be executed in the background, offline, or on demand. Another approach to shorten the computation time is the calculation of just one CYCLE of the reachability tree. Only one possible execution of the dataflow graph will be determined. This can be used to, e.g., detect COUNTER EXAMPLES [33]. If no cycle exists, the protocols are not compatible. Moreover, the cycle can be used to derive a cyclic schedule containing no deadlock. The memory usage of that schedule can be determined easily.
Protocol Adaptation and Deadlock Avoidance. During composition of different fifomata, only the compatible parts of them are integrated in the composed fifomata. By removing the incompatible parts from the original fifomata, debugging (regarding protocol violations) during runtime is possible.

If the composition of two fifomata is empty and one of the original fifomata could not complete its first aggregated operation, initialization tokens (cf. Fig. 1) are inserted. Deadlocks due to lack of initializing data are broken up this way.

Hierarchy. The canonical fifomaton of the underlying dataflow subgraph defines the behavior of a HIERARCHICAL COMPONENT. This definition is compositional meaning that the replacement of the hierarchical component by its defining dataflow subgraph does not change the semantics of the overall dataflow graph. This is a significant improvement as the usual assumption that a hierarchical component consisting of dataflow components has to behave like an atomic dataflow component may result in artificial deadlocks (cf. Section 2).

6 Results

These analysis techniques for embedded systems (cf. Fig. 3) are implemented using C++ and Qt 3.04. The tests were carried out on a computer with 1 GB of memory and a Intel Pentium 4 (2.8 GHz). Mean values of 10 runs are reported.

Type Resolution Algorithm

Systematic Tests. Dataflow graphs containing up to 800 edges were investigated. The constraint complexity was either simple (no type constraints), medium (equality constraints), or difficult (multivariate polynomials). The edges were inserted layer by layer beginning at the sources (forward), beginning at the sinks (backward), or by first constructing subgraphs which were finally connected (parts). Fig. 10 (a) depicts the mean insertion time depending on constraint complexity and graph size. All times measured are too short to disturb the designer.

Real World Example. In this test, a dataflow graph designed for the production control of beverage cans [34] was examined. The graph consists of 204 dataflow components and 336 edges with constraints ranging from simple to dif-

![Fig. 10. Insertion Times for Systematic Tests (a) and Real World Example (b).](image-url)
Fig. 11. Model Checking Algorithm (Computation Time (a) and Memory Usage (b)).

Difficult. Edges were inserted randomly. Fig. 10 shows the insertion times for each edge which confirm the results of the systematic tests.

Model Checking

Dataflow components with three different kinds of communication protocols depicted as regular expressions were used: (sme)*, (sm*e)*, and ((sm*e)|o)*. In the protocol (sme)*, the sequence ‘sme’ is read and/or written repeatedly in three atomic steps, for instance. The components were inserted layer by layer. Dataflow graphs containing up to 5000 component instances were examined. Fig. 11 (a) shows the overall calculation times for detecting one cycle for these component systems. Due to the state explosion problem, the calculation time increases exponentially. However, the analysis time is small enough for design accompanying use. Fig. 11 (b) shows the number of nodes of the reachability tree depending on the number of component instances which represent a measure for the memory needed by the model checker.

7 Conclusion

The starting point for this novel concept of design and analysis of component-based embedded systems (cf. Fig. 3) was the observation that the consideration of additional signal characteristics would help to avoid many runtime errors. Type system and model checking techniques are introduced which assist the designer during the construction of large embedded systems consisting of up to several hundred dataflow components by detecting as many errors as possible.

The underlying idea of this new approach is to provide correctness by construction (and not by testing). It can be generally stated that not only in the area of embedded systems software development processes that do not depend on exhaustive testing gain more and more importance.

The presented analysis techniques support the exploitation of all the significant advantages of a component-based approach:

- improved quality of the applications due to reuse at the component level,
- support of rapid prototyping and development (shorter time to market),
– easy adaptation to changing requirements and improved configurability (particularly important for embedded systems with changing environments),
– enhanced maintainability (due to software updates at the component level),
– separation of tasks of application engineers (component assembly) from tasks of software engineers (component construction), and
– development of protected intellectual property (IP) libraries containing novel algorithms.

These advantages lead to a multiplication of financial investment and an accelerated increase of innovation. Therefore, the presented analysis techniques are a valuable contribution to efficient design of component-based embedded systems.

References

32. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
Introducing a Component Technology for Safety Critical Embedded Real-Time Systems

Kristian Sandström, Johan Fredriksson, and Mikael Åkerholm

Mälardalen Real-Time Research Centre
Department of Computer Science and Engineering
Mälardalen University, Box 883, Västerås, Sweden
http://www.mrtc.mdh.se
kristian.sandstrom@mdh.se

Abstract. Safety critical embedded real-time systems represent a class of systems that has attracted relatively little attention in research addressing component based software engineering. Hence, the most widely spread component technologies are not used for resource constrained safety critical real-time systems. They are simply to resource demanding, to complex and to unpredictable. In this paper we show how to use component based software engineering for low footprint systems with very high demands on safe and reliable behaviour. The key concept is to provide expressive design time models and yet resource effective run-time models by statically resolve resource usage and timing by powerful compile time techniques. This results in a component technology for resource effective and temporally verified mapping of a component model to a commercial real-time operating system.

1 Introduction

The vehicle domain represents a class of embedded real-time systems where the requirements on safety, reliability, resource usage, and cost leaven all through development. Historically, the development of such systems has been done using only low level programming languages, to guarantee full control over the system behaviour. As the complexity and the amount of functionality implemented by software increase, so does the cost for software development. Therefore it is important to introduce software development paradigms that increase software development productivity. Furthermore, since product lines are common within the domain, issues of commonality and reuse is central for reducing cost as well as increasing reliability.

Component based software engineering is a promising approach for efficient software development, enabling well defined software architectures as well as reuse. Although component technologies have been developed addressing different demands and domains, there are few component technologies targeting the specific demands of safety critical embedded real-time systems. Critical for the safe and reliable operation of these systems is the real-time behaviour, where the timeliness of computer activities is essential. To be able to guarantee these
Properties it is necessary to apply real-time systems theory. Thus, a component technology to be used within this domain has to address specification, analysis, and implementation of real-time behaviour.

A typical real-time constraint is a deadline on a transaction of co-operating activities. A transaction in these systems would typically sample information about the environment, perform calculations based on that information and accordingly apply a response to the environment, all within a limited time frame. Also important is the ability to constrain the variation in periodicity of an activity (jitter). The reason for this is that variations in periodicity of observations of the environment and responses to the same, will affect the control performance. Hence, a component technology for this domain should have the ability to clearly express and efficiently realize these constraints [1,2,3,4].

The work described in this paper present a component technology for safety critical embedded real-time systems that is based on experience from our previous work with introducing state-of-the-art real-time technology in the vehicle industry. The benefits in development have been discussed in [5] and have also been proven by long industrial use. That real-time technology has been incorporated in the Rubus development suite and has been further developed [6]. Experience from the industrial application of the research reveals that a proper component model is not enough; success requires an unbroken chain of models, methods, and tools from early design to implementation and run-time environment.

The contribution of the work presented in this paper includes a component technology for resource effective and temporally verified mapping of a component model to a resource structure such as a commercial Real-Time Operating System (RTOS). This is made possible by introduction of a component model that support specification of high level real-time constraints, by presenting a mapping to a real-time model permitting use of standard real-time theory. Moreover, it supports synthesis of run-time mechanisms for predictable execution according to the temporal specification in the component model. Furthermore, in this work some limitations in previous work with respect to specification and synthesis of real-time behaviour are removed. These limitations are partially discussed in [5] and is mainly related to jitter and execution behaviour.

Many common component technologies are not used for resource constrained systems, nor safety critical, neither real-time systems. They are simply to resource demanding, to complex and unpredictable. The research community has paid attention to the problem, and recent research has resulted in development of more suitable technologies for these classes of systems. Philips use Koala [7], designed for resource constrained systems, but without support for real-time verification. Pecos [8] is a collaboration project between ABB and University partners with focus on a component technology for field devices. The project considers different aspects related to real-time and resource constrained systems, during composition they are using components without code introspection possibilities that might be a problem for safety critical applications. Rubus OS [6] is shipped with a component technology with support for prediction of real-
time behaviour, though not directly on transactions and jitter constraints and not on sporadic activities. Stewart, Volpe, and Khosla suggest a combination of object oriented design and port automaton theory called Port Based Objects [9]. The port automaton theory gives prediction possibilities for control applications, although not for transactions and jitter constraints discussed in this paper. Schmidt and Reussner propose to use transition functions to model and predict reliability in [10]; they are not addressing real-time behaviour. Wallnau et al. suggest to restrict the usage of component technologies, to enable prediction of desired run-time attributes in [11], the work is general and not focused on particular theories and methods like the work presented in this paper.

The outline of the rest of this paper is as follows; section 2 gives an overview of the component technology. In section 3 the component model is described and its transformation to a real-time model is explained in section 4. Section 5 presents the steps for synthesis of real-time attributes and discusses run-time support. Finally, in section 6, future work is discussed and the paper is concluded.

2 Component Technology

In this section we will give an overview of the component technology facilitating component based software development for safety-critical embedded real-time systems. We will hereafter refer to this component technology as the AutoComp technology. A key concept in AutoComp is that it allows engineers to practise Component Based Software Engineering (CBSE) without involving heavy run-time mechanisms; it relies on powerful design and compile-time mechanisms and simple and predictable run-time mechanisms. AutoComp is separated into three different parts; component model, real-time model and run-time system model. The component model is used during design time for describing an application. The model is then transformed into a real-time model providing theories for synthesis of the high level temporal constraints into attributes of the run-time system model. An overview of the technology can be seen in Fig. 1. The different steps in the figure is divided into design time, compile time, and run-time to display at which point in time during development they are addressed or used.

During design time, developers are only concerned with the component model and can practise CBSE fully utilizing its advantages. Moreover, high level temporal constraints in form of end-to-end deadlines and jitter are supported. Meaning that developers are not burdened with the task of setting artificial requirements on task level, which is essential [12], [5]. It is often natural to express timing constraints in the application requirements as end-to-end constraints.

The compile time steps, illustrated in Fig. 1, incorporate a transition from the component based design, to a real-time model enabling existing real-time analysis and mapping to a RTOS. During this step the components are replaced by real-time tasks. Main concerns in this phase are allocation of components to tasks, assignment of task attributes, and real-time analysis. During attribute assignment, run-time attributes that are used by the underlying operating system are assigned to the tasks. The attributes are determined so that the high level
constraints specified by the developer during the design step are met. Finally, when meeting the constraints of the system, a synthesis step is executed. It is within this step the binary representation of the system is created, often the operating system and run-time system are also included with the application code in a single bundle.

The run-time system is assumed to be a traditional RTOS with Fixed Priority Scheduling (FPS) of tasks. Most commercial RTOS can be classified into this category; furthermore they are simple, resource efficient and many real-time analysis techniques exist. In some cases a layer providing run-time support for the tasks has to be implemented in order to fully support FPS models used in real-time theory.

3 Component Model

Vehicles present a heterogeneous environment where the interaction between the computer system and the vehicle take different forms. Some vehicle functionality requires periodic execution of software, e.g., feedback control, whereas other functionality has a sporadic nature, e.g., alarms. Although vehicle control plays a central role, there is also an abundance of other functionality in vehicles that is less critical and has other characteristics, e.g., requires more flexibility. Although less critical, many of these functions will still interact with other more critical parts of the control system, consider for example diagnostics. We present a model that in a seamless way allows the integration of different functionality, by supporting early specification of the high level temporal constraints that a
given functionality has to meet. Moreover, the computational model is based on a data flow style that results in simple application descriptions and system implementations that are relatively straightforward to analyse and verify. The data flow style is commonly used within the embedded systems domain, e.g., in IEC 61131 used for automation [13] and in Simulink used for control modelling [14].

The definition of the AutoComp component model is divided into components, component interfaces, composition, the components invocation cycle, transactions and system representation. In Fig. 2 the component model is illustrated using UML2, which could be a possible graphical representation during design.

![UML Diagram](image)

**Fig. 2.** In the upper left part of the figure there is a UML 2 component diagram for modelling of a component. The lower part of the figure is a composition diagram showing a composition of two components. Finally the upper right part of the figure is a sequence diagram with a timing constraint that is used to express the end-to-end deadline for a transaction.

The components are defined as *glass box*, meaning that a developer can see the code of a component for introspection purposes. It does not mean that a developer has to look into a component during normal composition, and not that it is allowed to modify a component. The introspection possibility is a requirement during verification of safety critical applications in order to gain complete knowledge about components behaviour. Furthermore, the components can only exchange data with each others through data ports. A component can
be a composite containing a complete subsystem, or a basic component with an entry function. Composite components can be treated as any other component during composition, but it is also possible to enter a composite and change timing requirements and other properties. The entry function provided by non-composite components can be compared to the entry function for a computer program, meaning that the contained number of functions of the component can be arbitrary.

The interfaces offered by a component can be grouped into the two classes data and control interfaces. The data interfaces are used to specify the data flow between components, and consist of data ports. Data ports have a specified type and can be either provided or required. Provided ports are the ports provided by components for input, i.e., the ports a component reads data from. Required ports are the ports a component writes data to. A component also has a control interface with a mandatory control sink, and an optional control source. The control interface is used for specifying the control flow in the application, i.e., when or as a response to what component should be triggered. The control sink is used for triggering the functionality inside the component, while the control source is used for triggering other components.

During composition the developer has three main techniques to work with. The data flow is specified through connection of provided and required data ports. The rules are as follows; required ports must be wired to provided ports with a compatible type. It is possible to make abstractions through definition of composite components. Composite components can be powerful abstractions for visualizing and understanding a complex system, as well as they provide larger units of reuse. The control flow is specified through binding the control sinks to period times for periodic invocation, to external events for event invocation, or to control sources of other components for invocation upon completion of the other components.

A components invocation cycle can be explained as in the following sentences. Upon stimuli on the control sink, in form of an event from a timer, an external source or another component; the component is invoked. The execution begins with reading the provided ports. Then the component executes the contained code. During the execution, the component can use data from the provided ports and write to the required ports as desired, but the writes will only have local effect. In the last phase written data become visible on the required ports, and if the control source in the control interface is present and wired to the control sink of another component stimulus is generated.

Transactions allow developers to define and set end-to-end timing constraints on activities involving several components. A transaction in AutoComp can be defined as:

A transaction $Tr_i$ is defined by a tuple $< C, D, J_s, J_c >$ where:

- $C$ - represent an ordered sequence of components;
- $D$ - represent the end-to-end deadline of the transaction;
- $J_s$ - represent the constraint on start jitter of the transaction;
- $J_c$ - represent the constraint on completion jitter of the transaction.
Fig. 3. UML class diagram showing the static view of the component model.

The end-to-end deadline is the latest point in time when the transaction must be completed, relative to its activation. Jitter requirements are optional and can be specified for transactions involving time triggered components. Start jitter is a constraint of the periodicity of the transactions starting point, while completion jitter is a constraint on the periodicity of a transactions completion point. Both types of jitter are expressed as a maximum allowed deviation from the nominal period time. A restriction, necessary for real-time analysis, is that components directly triggered by an external event can only be part of a transaction as the first component.

A system can be described with the UML class diagram in Fig. 3. A system is composed of one or several components, each with a data interface, a control interface and a realization as a subsystem or an entry function. A system also has zero or more data couplings, describing a connected pair of required and provided data ports. Furthermore, systems have zero or more control couplings which describe a connected pair of control sink and source. Finally, the last part of a system is zero or more transactions with the included components, an end-to-end deadline and the possibility to specify jitter requirements.

4 Model Transformation

Model transformation involves the steps necessary in order to transit from the component model allowing an efficient and powerful design phase, to a run-time model enabling verification of temporal constraints and usage of efficient and deterministic execution environments. As previous stated in section 2 we assume a FPS run-time model. The FPS model defines a system as a set of
tasks with the attributes period time, priority, offset, and WCET. Hence, it is necessary to translate the component model with its temporal constraints in to tasks holding these attributes. The translation is performed in two separate steps; the first step is to make a transformation between components and task (task allocation), the second step is to assign attributes to the tasks (attribute assignment). To assign the FPS model attributes in such a way that the high level temporal constraints on transactions are met is non-trivial and has been addressed in research by e.g., [1], [3].

4.1 Task Allocation

The easiest approach for task allocation is a one to one relationship between components and tasks, but that is not necessarily optimal. In fact the task allocation step has a lot of different tradeoffs. Such tradeoffs can be found between reliability and run time overhead; few tasks reduce run time overhead at the cost of memory protection (usually at task level) between components. Testability and schedulability are examples of other properties that are affected by the allocation scheme.

In this paper we introduce a task allocation strategy that strives to reduce the number of tasks considering schedulability and reliability. Components are not allocated to the same task if schedulability is obviously negatively affected and structurally unrelated components are not allocated to the same task in order to cater for memory protection and flexibility.

The first step in the allocation process is to convert all composite components to a flat structure of the contained basic components. Secondly the following rules are applied:

1. All instances of components are allocated to separate tasks, Worst Case Execution Time (WCET) is directly inherited from a component to the corresponding task
2. The start jitterJs corresponding to a transaction with jitter requirements is set as a requirement on the task allocated for the first component in the ordered sequence C, while the completion jitter Jc is set to the task allocated for the last component in the sequence
3. Tasks allocated for components with connected pairs of control sink and sources, where the task with the source do not have any jitter requirements, and both tasks are participating in the same and only that transaction are merged. The resulting WCET is an addition from all integrated tasks WCET
4. Tasks allocated for time triggered components that have the same period time, not have any jitter constraints and are in a sequence in the same and only that transaction are merged. The resulting WCET is an addition from all integrated tasks WCET

The situation after application of the allocation rules is a set of real-time tasks. The high level timing requirements are still expressed in transactions, but instead of containing an ordered set of components a transaction now contain
**Table 1.** A component set.

<table>
<thead>
<tr>
<th>Sink Bound To</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>A T = 100</td>
<td>5</td>
</tr>
<tr>
<td>B A.Source</td>
<td>10</td>
</tr>
<tr>
<td>C T = 60</td>
<td>5</td>
</tr>
<tr>
<td>D T = 40</td>
<td>5</td>
</tr>
<tr>
<td>E T = 40</td>
<td>6</td>
</tr>
<tr>
<td>F T = 40</td>
<td>9</td>
</tr>
</tbody>
</table>

an ordered set of tasks. The rest of the attributes, those that cannot be mapped directly from the component model to the real-time model are taken care of in the following attribute assignment step. In Fig. 4, given the two transactions $Tr_1 = \langle C, D, J_s, J_c \rangle = \langle A, B, C, 60, -, 25 \rangle$ and $Tr_2 = \langle C, D, J_s, J_c \rangle = \langle D, E, F, 40, 5, - \rangle$ the task allocation step for the components in Table 1 is shown. The resulting task set is in Table 2.

**Table 2.** The resulting task set.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Jitter</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>T = 100</td>
<td>15</td>
</tr>
<tr>
<td>Task 2</td>
<td>T = 60</td>
<td>5</td>
</tr>
<tr>
<td>Task 3</td>
<td>T = 40</td>
<td>5</td>
</tr>
<tr>
<td>Task 4</td>
<td>T = 40</td>
<td>15</td>
</tr>
</tbody>
</table>

### 4.2 Attribute Assignment

After the components have been assigned to tasks, the tasks must be assigned attributes so that the high level temporal requirements on transactions are met. Attributes that are assigned during task allocation are WCET for all tasks, a period time for periodic tasks and a Minimum Interarrival Time (MINT) for event triggered tasks.
The scheduling model that is used throughout this paper is FPS, where tasks have their priorities and offsets assigned using an arbitrary task attribute assignment methodology. Examples of existing methods that can be used for priority assignment are Bate and Burns [1], Sandström and Norström [3] or by combination of Verraballi [15] or Cheng and Agrawala [16] with Dobrin, Fohler and Puschner [17]. In this paper it is assumed that task attributes are assigned using the algorithm proposed by Bate and Burns [1], and it is showed that the component model described in this paper is applicable to their analysis model. Weather the tasks are time triggered or event triggered is not considered in the Bate and Burns analysis but is required during the mapping to the FPS model, where periodic and event triggered (sporadic) tasks are separated. The attributes that are relevant, considering this work, in the Bate and Burns approach are listed below.

For tasks:

**T (Period)** - All periodic tasks have a period time that is assigned during the task allocation. Sporadic tasks have a MINT that analytically can be seen as a period time;

**J (Jitter)** - The jitter constraints for a task is the allowed variation of task completion from precise periodicity. This type of jitter constraint is known as completion jitter. Jitter constraints can be set on the first and last task in a transaction;

**R (Worst Case Response Time)** - The initial Worst Case Response time for a task is the WCET for the task, i.e., the longest time for a task to finish execution from its starting point in time.

For transactions:

**T (Period)** - The period of a transaction is the least common multiple of the period times of the participating tasks of the transaction;

**End-to-End Deadline** - Transactions have a requirement that all tasks have finished their execution within a certain time from the transactions point of start in time.

In Bate and Burns approach additional attributes, such as deadline and separation for tasks and jitter requirements for transactions are considered. In this paper those attributes are disregarded since there are no such requirements in the previously described component model. It is trivial to see that from the component model, the period and jitter constraints match the model proposed by Bate and Burns. The initial worst case response time R is assigned the WCET value in the component model. For the transaction the end-to-end deadline requirements match the transaction deadline of the Bate and Burns model. The period time of the transaction is derived from the least common multiple of the period of the tasks participating in the transaction.

The next step is naturally to assign the FPS model with run-time and analysis attributes. The new attributes priority and offsets will be derived through existing analysis methods [1]. The new parameters for the FPS model are described below.
P (Priority) - The priority is an attribute that indicates the importance of the task relative to other tasks in the system. In a FPS system tasks are scheduled according to their priority, the task with the highest priority is always executed first. All tasks in the system are assigned a priority;

O (Offset) - The offset is an attribute that periodic tasks with jitter constraints are assigned. The earliest start time is derived by adding the offset to the period time.

In Table 3 it is summarized what attributes belonging to time triggered and event triggered tasks in the FPS model.

**Table 3.** Attributes associated with time and event triggered tasks.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Time triggered</th>
<th>Event triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MINT</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Priority</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Offset</td>
<td>X (Upon Jitter Constraints)</td>
<td>X</td>
</tr>
<tr>
<td>WCET</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Applying the Bate and Burns algorithm determines task attributes from the tasks and transactions described in Table 2. The resulting run-time attributes priority, offset period and WCET are shown in Table 4. The attributes offset and priority are determined with the Bate and Burns analysis, whilst the period and WCET are determined in the task allocation.

**Table 4.** Assigned task attributes.

<table>
<thead>
<tr>
<th></th>
<th>Priority</th>
<th>Offset</th>
<th>Period</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>15</td>
</tr>
<tr>
<td>Task 2</td>
<td>1</td>
<td>35</td>
<td>60</td>
<td>5</td>
</tr>
<tr>
<td>Task 3</td>
<td>4</td>
<td>0</td>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>Task 4</td>
<td>3</td>
<td>0</td>
<td>40</td>
<td>15</td>
</tr>
</tbody>
</table>

In Fig. 5 a run-time trace for an FPS system is shown and the transactions $Tr_1$ and $Tr_2$ are indicated.

**Fig. 5.** Trace of an FPS schedule.
When the FPS model has been assigned its attributes it has to be verified. The verification of the model is performed by applying real-time scheduling analysis to confirm that the model is schedulable with the assigned parameters. This is necessary since attribute assignment does not necessarily guarantee schedulability, but only assigns attributes considering the relation between the tasks.

4.3 Real-Time Analysis

To show that the FPS tasks will meet their stipulated timing constraints, schedulability analysis must be performed. Much research has been done with respect to analysis of different properties of FPS systems, and all those results are available for use, once a FPS model has been established. The temporal analysis of an FPS system with offsets, sporadic tasks and synchronization has been covered in research by e.g., Palencia et al. [18], [19] and Redell [20].

The output from the analysis is whether the system is feasible or not in the worst case. If the analysis shows that the system is infeasible, the parts that can not keep its requirements are either changed and reanalysed or emphasised for the developer to make changes.

5 Synthesis

The next step after the model transformation and real-time analysis is to synthesise code for the run-time system. This includes mapping the tasks to operating system specific task entities, mapping data connections to an OS specific communication, modifying the middleware, generating glue code, compiling, linking and bundling the program code (see Fig. 6).

The synthesis is divided into two major parts. Given a task set and necessary information about the run-time system, the synthesis generates code considering communication, synchronization.
The first part in synthesis is to resolve the communication within and between tasks. Two communicating components that are assigned to different tasks will form an Inter Task Communication (ITC) while communication between components assigned to the same task are realized with shared data spaces within the task. The ITC is later mapped to operating system specific communication directives.

The other part in the synthesis is to resolve the control couplings, i.e., the sink and source. If a tasks starting point is dependent on the former tasks finishing point the tasks have to be synchronized. The synchronization is solved through scheduling. The synthesis will generate code for scheduling periodic tasks, handle the control flow between tasks and consider offsets. The code generated for the periodic scheduling and offsets is dependent on the middleware and can be realized as a configuration file or actual code in each task. Invocations of sporadic tasks are mapped to event handlers in the middleware or the operating system.

![Diagaram](image)

**Fig. 7.** A component model with adjustments for different operating systems to promote platform independence.

It is assumed that a middleware is present as shown in Fig. 7, for each platform and that it provides functionality that the component model needs but the operating system does not provide. The more functionality the operating system provides, the smaller the middleware has to be. The middleware encapsulates core communication and concurrency services to eliminate many non-portable aspects of developing and is hence platform specific in favour of a platform independent component model. Typical functionality that is not provided by most commercial RTOS is periodicity and support for offsets. The middleware also need to support sink and source couplings since task coupled with its source need to be able to invoke the corresponding task. The run-time system conforms to FPS and hence the run-time task model is similar to the previously described FPS model with some exceptions. The worst case execution time is merely an analysis attribute and is not needed in the run-time model. The MINT is usually a requirement on the environment rather than a task attribute, and is thus also analytical and unnecessary. Hence the run-time task model is for periodic tasks Period time, Priority, Offset and for sporadic tasks Priority.
6 Conclusions and Future Work

In this paper we show how to use component based software engineering for low footprint systems with very high demands on safe and reliable behaviour. The key concept is to provide expressive design time models and yet resource effective run-time models by statically resolve resource usage and timing by powerful compile time techniques.

The work presented in this paper introduces a component technology for resource effective and temporally verified mapping of a component model to a resource structure such as a commercial RTOS. This is made possible by introduction of a component model that support specification of high level real-time constraints, by presenting a mapping to a real-time model, permitting use of standard real-time theory, and by synthesis of run-time mechanisms for predictable execution according to the temporal specification in the component model.

Although the basic concept has been validated by successful industrial application of previous work [5], it is necessary to further validate the component technology presented here. In order to facilitate this, a prototype implementation of the component technology is under development where the core part has been completed. The prototype will enable evaluation of different technology realisations with respect to performance. Moreover, parts of the model transformation need additional attention, foremost the strategies for allocation of components to tasks. Furthermore, we will make efforts in extending the component model making it more expressive and flexible while still keeping the ability for real-time analysis. Interesting is also to investigate trade-offs between run-time footprint and flexibility with respect to e.g., adding functionality post production. Finally, the component technology will be evaluated in a larger, preferably industrial, case.

References


Mathworks: (Mathworks homepage : http://www.mathworks.com)


A Hierarchical Framework
for Component-Based Real-Time Systems

Giuseppe Lipari*, Paolo Gai, Michael Trimarchi,
Giacomo Guidi, and Paolo Ancilotti

Scuola Superiore S. Anna Pisa (Italy)
{lipari,pj,paolo}@sssup.it, {trimarchi,guidi}@gandalf.sssup.it

Abstract. In this paper, we describe a methodology for the design and
development of component-based real-time systems. In our model,
a component consists of a set of concurrent real-time threads that can
communicate by means of synchronized operations. In addition, each
component can specify its own local scheduling algorithm. We also discuss
the support that must be provided at the operating system level, and
present an implementation in the SHaRK operating system.

1 Introduction

Component-based design and development techniques are now being applied
to real-time embedded systems. Until now, little work has been done on the
characterization of the quality of service of a component from a temporal point
of view. This characterization is especially useful in the real-time domain, where
components consist of concurrent cyclic tasks with temporal constraints (e.g.
deadlines). In fact, when we integrate all the components in the final system,
we must be able to analyse the schedulability of the system (i.e. to check if the
temporal constraints are respected).

Lipari, Bini and Fohler [1] presented a model for real-time concurrent com-
ponents. A component consists of one or more concurrent real-time threads and it
is characterized by a demand function that describes its temporal requirements.
The methodology was later extended by Lipari and Bini [2]. In this paper, we
refine the model of a real-time concurrent component by considering blocking
primitives, like synchronized methods. We also present an implementation of
these techniques in the real-time operating system SHaRK.

The paper is organized as follows. In Section 2 we describe the model of the
system and motivate our work. In Section 4 we present our model of a component
and we list the system requirements. Section 5 describes briefly the mechanisms
that a operating system must support. Section 6 describes the implementation
in the SHaRK operating system. Finally, Section 7 presents the conclusions and
describes some future work.

* This work has been partially supported by the Italian Ministry of University and
Research within the COFIN 2001 project “Quack: a platform for the quality of new
generation integrated embedded systems”, and by the European Community within
the IST project 34140 FIRST (Flexible Integrated Real-Time System Technology).

© Springer-Verlag Berlin Heidelberg 2004
2 System Model

The thread model of concurrent programming is very popular and it is supported by most operating systems. In this model, concurrency is supported at two levels: processes and threads. Different threads belonging to the same process share address space, file descriptors, and other resources; the communication between them is often realized by means of shared data structures protected by mutexes. The thread model is supported by all general purpose operating systems because it has a lot of advantages with respect to pure process models: the designer of a concurrent application can design the application as a set of cooperating threads, simplifying the communication and reducing the overhead of the implementation. When designing a concurrent application, in which tasks have to cooperate tightly and efficiently, the thread model is the most suited.

Classical hard real-time systems usually consist of periodic or sporadic tasks that tightly cooperate to fulfill the system goal. In this paper we assume that real-time tasks are implemented as threads, and a classical real-time application as one single multi-thread process. In the remainder of this chapter, we will use the terms thread and task as synonyms. The same for the terms application and process. A user that wants to execute (soft) real-time applications in a general-purpose operating system would like to have the following nice features:

- It should be possible to assign each real-time application a fraction of the system resources, so that it executes as if it were executing alone in a slower virtual processor;
- Each application should receive execution in a timely manner, depending on its real-time characteristics (e.g., the tasks’ deadlines);
- A non real-time application should not be able to disrupt the allocation guaranteed to real-time applications.

Therefore, in this model, we distinguish two levels of scheduling. The global scheduler selects which application is assigned the processor, and the local scheduler selects which task has to execute on the processor. This two-level scheduling has two obvious advantages:

- each application could use the scheduler that best fits its needs;
- legacy applications, designed for a particular scheduler, could be re-used by simply re-compiling, or at most, with some simple modification.
- it adds support for component-based design of concurrent real-time applications. Each component can be seen as a multi-threaded process with its own custom scheduler.

If we are forced to use a single scheduling paradigm for the whole system, we must reduce all the activities into periodic tasks and schedule them by a single paradigm. This solution requires some extra programming effort and is not optimal in terms of resource usage. The best choice would be to program each sets of activities as a distinct component handled by its own scheduler, and then integrate all components in the same system.

If we want to integrate the various components in a new system and we cannot go back to the design phase (for example for cost reasons), we need a method for combining and analyzing the two components together, without changing the scheduling algorithms.
3 Related Work

Kopetz [3] proposes a methodology based on the Time Triggered paradigm to develop systems according to a component-based approach. The approach is very interesting, but it is only applicable in the context of static off-line scheduled systems. Isovic, Lindgren and Crnkovic [4] presented a similar idea in the context of the slot shifting scheduler [5]. However, a component can consist of one single thread. Nielsen and Agha [6] propose to further constraint a component in order to separate the functional specification from the timing constraints. For example, a component is not allowed to specify the scheduling policy, nor priorities or deadlines. In contrast, in our work paper we explicitly allow components to specify their own scheduling policy. Stankovic [7] proposes a tool set called VEST. Again, a component is not allowed to specify its own scheduling algorithm. Moreover, a failing component can influence the behaviors of all other components in the systems, since there is no temporal isolation between components. A general methodology for temporal protection in real-time system is the resource reservation framework [8,9,10]. The basic idea is that each task is assigned a server that is reserved a fraction of the processor available bandwidth: if the task tries to use more than it has been assigned, it is slowed down. Recently, many techniques have been proposed for extending the resource reservation framework to hierarchical scheduling. Baruah and Lipari in [11] propose the H-CBS algorithm. A similar work has been done by Saewong et al. [12] in the context of the resource kernels. Lipari, Bini and Fohler [1,2] developed a methodology for analysing hierarchical schedulers based on a server algorithm like the CBS.

4 Component-Based Real-Time Systems

In our model, a real-time component $C$ is defined by a $t$-uple $\{T, S, R, P, DBF\}$, where: $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is the set of concurrent threads; $S$ is the local scheduling algorithm for the component's threads (see Section 2); $R = \{\omega_1, \ldots, \omega_m\}$ is the set of required synchronized operations; $P = \{\pi_1, \ldots, \pi_l\}$ is the set of provided synchronized operations; $DBF(t)$ is the demand bound function as defined in [1,2].

A thread $\tau_i = \{C_i, D_i, T_i\}$ can be periodic or sporadic. It is characterized by a worst-case execution time $C_i$, a relative deadline $D_i$ and a period (or a minimum interarrival time) $T_i$. Such threads must be scheduled by the local scheduling algorithm $S$. In general, any scheduling algorithm can be used as local scheduling algorithm.

The component may offer some synchronized operation to be used by other components. These are called provided synchronized operations. Every provided operation $\pi_j$ is characterized by a mutex $\mu$ and a worst case execution time $d(\pi_j)$. Of course, the provided operation can be also accessed by the threads belonging to the same component.

The component may need to execute synchronized operations provided by other components. These are called required synchronized operations. The thread that performs a call to a required synchronized operation $\omega_i$ can be blocked be-
cause the corresponding mutex is currently used by another thread. This blocking time has to be taken into account for checking the performance of the component.

The demand bound function $DBF(\Delta t)$ of a component is the maximum amount of time that the component requires in any interval of length $\delta t$ otherwise some deadline could be missed. Therefore, in designing our system, we must ensure that the component is allocated at least $DBF(\Delta t)$ units of time in every interval of time $\Delta t$. The demand bound function can be computed from the thread characteristics and from the local scheduler, as described in [1,2].

For the component to be schedulable, the following condition must hold: $\forall \Delta t > 0, DBF(\Delta t) \leq Z(\Delta t)$, where $Z(\Delta t)$ is the minimum amount of execution time that the component can receive in any interval of length $\Delta t$.

5 Operating System Support

In this section we discuss the basic mechanisms to be provided by the operating system to support our methodology. These mechanisms have been implemented in the SHaRK OS [13], an open source real-time kernel.

Temporal Isolation. Our system allows the integration of different components. Some component may be very critical (each thread in the component must complete its job before the deadline), other components can be less critical (if some deadline is missed the quality of service provided may decrease), other may not possess temporal constraints (non real-time components).

To separate concerns, the operating system must support the “temporal isolation property”: the temporal behavior of one component (i.e. its ability to meet its deadlines) should only depend on the amount of bandwidth assigned to it and not on the presence of other components in the system.

Temporal isolation can be provided by using the resource reservation framework [8]. In this framework, each component $C_i$ is assigned a fraction of the processor bandwidth $U_i$. The net effect is that each component executes as it were executing alone on a dedicated processor of speed $U_i$. In the SHaRK OS, we use the Constant Bandwidth Server (CBS) [14], an algorithm of the class of the resource reservation algorithms, and the GRUB algorithm [15], an extension of the CBS.

In these algorithms, each “component” is assigned a “server”. A server is an abstraction (i.e. a structure internal to the operating system) that is used by the global scheduler to store the scheduling parameters. Each server $S_i$ is characterized by a budget $Q_i$ and a period $P_i$. The scheduler guarantees that each server $S_i$ (and in turn its associated component) will receive an execution time of $Q$ units of time every period $P_i$. In practice, these scheduling algorithms are similar to a Round Robin Scheduler, but each server is assigned a different quantum $Q_i$. The only constraint is that the total bandwidth assigned to the servers cannot exceed 1: $\sum_{i=1}^{n} \frac{Q_i}{P_i} \leq 1$.

Hierarchical Scheduling. Each component is assigned a local scheduler $S$. The global scheduler is the CBS algorithm, which is able to provide temporal isolation. The CBS algorithm selects the “server” to be executed. In turn, the local
scheduler selects which one of the component’s threads must be executed. The thread is allowed to execute until the thread is preempted by another thread of the same component, or the budget of the component is exhausted.

By using a hierarchical scheduling strategy together with the resource reservation framework, we guarantee that each component behaves exactly as it were executing alone on a dedicated slower processor. In this way we allow independence among components and temporal protection.

**Synchronized Operations.** When we consider synchronized operations, components can actually interfere among each other. For example, if a component $C_1$ wants to invoke a synchronized operation $\pi_{21}$ on another component $C_2$, and the operation is currently locked, the first component experiences a blocking time delay. This delay depends on how long the operation remains locked by another component. If it remains locked for too long, some deadline in component $C_1$ could be missed.

There is no way to solve this problem. Therefore, we need to take this blocking time into account during the integration phase.

Another problem is how to reduce the blocking time. In fact, a particular problem, called “Priority Inversion” may arise [16]. The work in [16] has been recently extended to resource reservation algorithms by Lamastra, Lipari and Abeni [17] that proposed the Bandwidth Inheritance Algorithm (BWI). In this work, every server is characterized by an interference time $I_i$ due to the presence of synchronized operations. This interference must be taken into account during the integration phase to check if all components will respect their temporal constraints.

**System Integration.** Our system is composed of many components interacting through synchronized operations. After each component has been developed individually, we can analyze the temporal behavior of the entire system by using the following steps:

1. First, we analyze each component in isolation.
2. The unknown variables $d(\omega_j)$ and $I_i$ are all initialized to 0.
3. We now apply the methodology described in [2] to compute the server budget $Q_i$ and the server period $P_i$ for each component.
4. Then, we try to integrate all components in the final system. In doing this, we can compute the duration of the synchronized required operations $d(\omega_i)$ as the length of the corresponding synchronized provided operations $d(\pi_i)$. Moreover, by applying the algorithm described in [17], we can compute the interference time $I_i$ for each component. Of course, the interference time may be greater than the interference time previously assigned. If this is the case, then we go back to step 3 and recompute the budgets and the periods. The iteration stops when the value of the interference time is less than or equal to the value computed at the previous step.

### 6 Implementation

SHaRK (Soft and Hard Real-time Kernel) is an open source real-time operating system [13] purposely designed to allow the compositability of new approaches
in CPU/Network Scheduling. SHaRK allows the programmer to define its own algorithm. The applications can be developed independently from a particular system configuration, so that new modules can be added or replaced to evaluate the effects of specific scheduling policies in terms of influence on the performance of the overall system. The module interface is the result of a joint work with the University of Cantabria and it was implemented with the financial support of the FIRST (Flexible Integrated Real-Time System Technology) project (IST-2001-34140).

**Kernel Architecture.** SHaRK provides an interface to the user to define a module. The aim of each module is basically to export a common interface so that the desired behavior can be specified independently of the other components. In order to realize independence between applications and scheduling algorithms (and between the schedulers and the kernel), SHaRK is based on a Generic Kernel, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the scheduling modules. In a similar fashion, the access to shared resources is coordinated by resource modules.

A typical problem that arises when developing a real-time component is that the various threads composing the component should be written without thinking at the scheduling algorithm, and only at the end the designer should map its requirements to the real scheduling algorithm it uses. This orthogonal way of thinking is really useful when composing together different components, and it cannot be supported by most of the commercial kernels, that only export a limited set of scheduling algorithms. Looking at these problems, one of the goals of SHaRK is to allow the user to easily specify the QoS of a thread in a way independent from the underlying scheduling algorithm that will be used in the final application. The user can then change the scheduling algorithm and the mutex access protocol without modifying the application.

**Scheduling Modules and Shared Resource Access Protocols.** Scheduling Modules are components used by the Generic Kernel to schedule threads. The implementation of an scheduling algorithms may rely on the presence of another scheduling module, called the Host Module. Such a design choice reflects a feature needed to implement hierarchical scheduling, where the local scheduling algorithms have to choose a thread that is then inserted directly into the scheduling queues of the global scheduling algorithm. When the Generic Kernel has to perform a scheduling decision, it asks the modules for the thread to schedule. Due to lack of space, we cannot describe in detail the functions exported by a scheduling module. The interested reader can refer to the documentation on the SHaRK website, which explains how to create, initialize and use a scheduling module.

As for scheduling, SHaRK achieves modularity also in the implementation of shared resource access protocols. Resource modules are used to implement an interface similar to the POSIX mutexes, and to make resource protocols modular and almost independent from the scheduling policy and from the others resource protocols. To achieve complete modularity, the SHaRK Generic Kernel supports a generic priority inheritance mechanism independent from the scheduling modules. Such a mechanism is based on the concept of shadow threads. A shadow thread is a thread that is scheduled in place on another thread chosen by the
scheduler. When a thread is blocked by the protocol, it is kept in the ready queue, and a shadow thread is binded to it; when the blocked thread becomes the first thread in the ready queue, its binded shadow thread is scheduled instead. In this way, the shadow thread “inherits” the priority of the blocked thread. Using this approach a large number of shared resources protocols can be implemented in a way independent from the scheduling implementation. This independence is very important, since it allows to choose the mutual exclusion protocol and to change it in a second phase when integrating all the components that compose the final application.

Hierarchical Scheduling Implementation. The structure of the SHaRK modules allows the coexistence of different scheduling algorithms in the same system. This characteristic has been extended to support hierarchical scheduling as described in Section 5. In this configuration, one module implements the global scheduling mechanisms (CBSSTAR). Then, there is one module for each component. These modules implement the local scheduling strategy. It communicates with the CBSSTAR scheduling module to select which thread has to be scheduled when a certain component is selected to execute. Finally, the BWI algorithm [17] has been implemented to allow components to access synchronized operations. The basic idea of the implementation is the following: when a thread blocks on a shared resource, the shadow pointer of the blocked thread is set to the blocked thread. When the blocked thread is scheduled, the blocking thread is scheduled instead. The Generic Kernel records in its data structures who is the thread selected by the scheduler (in that case, the blocked thread) and who is the thread that is really executed (that is, the blocking thread): based on these informations, the CBSSTAR is able to do capacity accounting for the blocked thread in the right way. Due to lack of space, we can not describe the events related to the local scheduler and the CBSSTAR/BWI implementation (for a detailed description, please refer to [18]).

7 Conclusions and Future Work

In this paper, we describe a methodology for the design and the development of component-based real-time systems. Unlike previous proposals, we propose a model of a component consisting of a set of concurrent real-time threads plus their scheduling algorithm. Therefore, we can compose components with different temporal constraints and schedulers. After describing the model of the component, we describe the support needed at the operating system level. Moreover, we present the implementation of this framework on the SHaRK operating system. As a future work, we would like to further extend the model to consider other kinds of interactions, like the use of rendez-vous operations and synchronous message passing. Finally, we would like to acknowledge Prof. Giorgio Buttazzo, who created and coordinated the SHaRK project giving us the background knowledge needed to do this work. The SHaRK Kernel is distributed under the GPL license and can be downloaded at the URL http://shark.sssup.it.
References

Abstract. According to Szyperski, “a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only”. But it is well known that these contractually specified interfaces should go well beyond mere syntactic aspects: they should also involve functional, synchronization and Quality of Service (QoS) aspects. In large, mission-critical component based systems, it is also particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces with the QoS contracts obtained from required interfaces. In this paper we propose a language called QoSCL (defined as an add-on to the UML2.0 component model) to let the designer explicitly describe and manipulate these higher level contracts and their dependencies. We show how the very same QoSCL contracts can then be exploited for (1) validation of individual components, by automatically weaving contract monitoring code into the components; and (2) validation of a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation to a Constraint Logic Programming language. We illustrate our approach with the example of a GPS (Global Positioning System) software component, from its functional and contractual specifications to its implementation in a .Net framework.

1 Introduction

In Szyperski's vision [1], “a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-party”. In this vision, any composite application is viewed as a particular configuration of components, selected at build-time and (re-)configured at run-time.

This point of view is now widely adopted in components middleware technologies such as Corba Component Model (CCM) [2] or Microsoft .Net/Com for instance. In these various middleware, a software component is a binary executable code deployed in an environment which manages it (EJBContainer for EJB or component home for CCM). A software component only exhibits its provided or required interfaces, hence defining basic contracts between components allowing one to properly wire them. But it is well known that these contractually specified interfaces should go well beyond
mere syntactic aspects: they should also involve functional, synchronization and Quality of Service (QoS) aspects. In large, mission-critical component based systems, it is also particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces with the QoS contracts obtained from required interfaces.

The aim of this article is to present a QoS contract model (called QoSCL for QoS Constraint Language), allowing such QoS contracts and their dependencies to be specified at design-time in a UML2.0 [3] modeling environment. We then show how the very same QoSCL contracts can be exploited for (1) validation of individual components, by automatically weaving contract monitoring code into the components; and (2) validation of a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation to a Constraint Logic Programming.

The rest of the paper is organized as follows. Using the example of a GPS (Global Positioning System) software component, Section 2 introduces the interest of modelling components, their contracts and their dependencies, and describes the QoS Constraint Language (QoSCL). Section 3 discusses the problem of validating individual components against their contracts, and proposes a solution based on automatically weaving reusable contract monitoring code into the components. Section 4 discusses the problem of validating a component assembly, including getting end-to-end QoS information inferred from individual component contracts by automatic translation to a Constraint Logic Programming. This is applied to the GPS system example, and experimental results are presented. Finally, Section 5 presents related works.

2 The QoS Contracts Language

2.1 Modeling Component-Based Systems

In modelling techniques such as UML2.0 for example, a component is a behavioural abstraction of a concrete physical piece of code, called artifacts. A component has required and provided ports, which are typed by interfaces. These interfaces represent the required and provided services implemented by the modelled artifact. The relationship between these required and provided services must be explicitly stated by the component. The knowledge of this relationship is of utmost importance to the component-based application designer. In the rest of this section, we address this relationship using the example of a GPS device.

A GPS device computes its current location from satellite signals. Each signal contains data which specifies the identity of the emitting satellite, the time when it is sent, the orbital position of the satellite and so on. Every fifteen seconds, each satellite emits the current information as a new data stream.

In order to compute its current location, the GPS device needs at least three signals. The number of received signals is unknown \textit{a priori}, because obstacles might block the signal propagation.
Our GPS device is modeled as a component which provides a `getLocation()` service, and requires a `getSignal()` service from Satellites components. The GPS component is made up of four components:

- the decoder which contains twelve satellite receivers (only three are shown on Fig. 1). This element receives the satellite streams, and demultiplexes it in order to extract the data for each satellite. The number of effective data obtained via the `getData()` service depends not only on the number of powered receivers, but also on the number of received signals. Indeed, this number may change at any time.
- The computer which computes the current location (`getLocation()`) from the data (`getData()`) and the current time (`getTime()`).
- The battery which provides the power (`getPower()`) to the computer and the decoder.
- The clock component which provides the current time (`getTime()`).

```
<table>
<thead>
<tr>
<th>Location &amp; Signal Recovery Component</th>
<th>Decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td></td>
</tr>
<tr>
<td>Satellite</td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td></td>
</tr>
<tr>
<td>Battery</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td></td>
</tr>
</tbody>
</table>
```

Fig. 1. The GPS component-based model.

### 2.2 Contract Aware Components

In component-based models, the services are usually specified at a syntactic level. This level of specification is not precise enough. Indeed, a service can be unavailable according to the state of the environment and, reciprocally, the environment can be modified by the execution of a service.

Following [4] component contracts can be classified into four levels. The first level is the type compatibility. The second level adds pre/post-conditions: the operation’s behavior is specified by using Boolean assertions for each service offered, called pre and post-conditions, as well as class invariants [5]. The third level adds synchronization constraints and the fourth level provides extra-functional constraints. To be more precise, we can build on the well-known idea of design-by-contract [6] provides negotiable contracts for components, and ensures that a service will perform correctly.

The contract concept implicitly includes the dependency between the required and provided extra-functional properties. This fact is clearly illustrated in our example.

The GPS application contains several time out constraints that bound the response time of services. For instance, the provided `getLocation()` service must ensure that it is completed in a delay less than 30s, whereas the `getData()` service must be completed in less than 25s for example.
However, it is obvious that the time spent to acquire data from the decoder, denoted $\theta_p$, has a direct impact on the global cost in time of the \textit{getLocation()} service, denoted $\theta_c$. Not only $\theta_c$ depends on $\theta_p$, but also on the number of active receivers, denoted $\text{nbr}$, because of the interpolation algorithm implemented in the Computer component. $\theta_p$ and $\text{nbr}$ are two extra-functional properties associated to the \textit{getData()} service provided by the Decoder component. The relation which binds these three quantities is:

$$\theta_c = \theta_p + \text{nbr} \times \log(\text{nbr}) . \quad (1)$$

Each receiver demultiplexes a signal, in order to extract the data. This operation has a fixed cost in time: nearly 2 seconds. In addition, the demultiplexed signals must be transformed into a single data vector. This operation takes 3s. If $\theta_r$ (resp. $\theta_s$) denotes the time spent by the receiver to complete the \textit{getDatal()} service (resp. the satellite to complete its \textit{getSignal()} service), then we have the two following formulae:

$$\theta_r = \theta_s + 2 , \quad (2)$$

$$\theta_p = \max(\theta_r) + 3 . \quad (3)$$

There exist many QoS contracts languages, such as QML [9], which allows the designer to specify the extra-functional properties and their constraints on the provided interfaces only (see §5). However, at the component level, it is not possible to specify explicit dependency between them. Moreover, in a component-based model, the services are provided as well as required, and so the QoS contracts must be defined on each side of an assembly in order to verify the contractual conformance, between a required contract and a provided one. To overcome these limitations we introduce the QoS Constraint Language (QoSCL).

### 2.3 Extra-Functional Dependencies with QoSCL

Our own contract model for extra-functional contracts extends the UML2.0 components metamodel. We designed the QoSCL notation with the following objectives in mind:

1. since the extra-functional contracts are constraints on continuous values within multidimensional spaces, we wanted to keep the QML definitions of dimensions and contract spaces.

2. Since our extra-functional contracts would be used on software components with explicit dependency specification, we needed means to express a provided contract in terms of required contracts.

3. Since we targeted platform independent designs, we wanted to use the UML notation and its extension facilities.

We thus designed our extra-functional contract notation as an extension of the UML2.0 metamodel. The overall design principles are as follows with the corresponding new metaclasses into brackets (Fig. 2):
- an extra-functional contract type \([\text{ContractType}]\) is represented as a special case of interface. Therefore a contract type provides a set of operations as means of contract description and configuration.

- As in QML [7], a contract type is a multidimensional space of dimensions \([\text{Dimension}]\). Each dimension is described as an operation. Calling this operation returns the current value of a contract of the corresponding contract type. Since a dimension is represented by an operation, it may have formal parameters in its definition. These parameters are needed to explicit the dependency of a dimension’s value on a provided service contract with respect to a required service contract.

- A contract \([\text{ContractQoSCL}]\) is an implementation of a contract type. At execution time a contract is reified; it can be queried and configured using the interface provided by a contract type.

```
Component « implements » ComponentQoSCL

Port « implements » PortQoSCL

Interface « implements » ContractType « implements » ContractQoSCL

Operation « implements » Dimension
```

**Fig. 2.** QoSCL metamodel.

With the QoSCL metamodel, it is possible to specify contracts such as the TimeOut contract useful for our GPS:

```
+timeOut():bool
+start():bool
+onTimeEvent(source:object, e:ElapsedEventArgs)
```

```
 délai:double
日夜x():double
日夜():bool
:start():bool
+isvalid():bool
+onTimeEvent(source:object, e:ElapsedEventArgs)
```

**Fig. 3.** The TimeOut contract with QoSCL.

The QoSCL metamodel handles three specific aspects of contracts: dependency, composition, and adaptative behaviour. The dependency is the core of this work, and our main contribution to enhance existing extra-functional contracts specification languages, such as QML. A Dimension of a ContractType is an Operation, which can depend on another ContractTypes and their Dimensions.
QoSCL makes it also possible to refine a contract via generalization association. At last, like any abstract functional model, it is possible to implement different behaviors for the same Operation, such as a Dimension. Thus, the renegotiation of a contract can be implemented according to its environment. It is possible to specify in UML2.0 a behaviour thanks to sequence diagrams, activity diagrams or state machine.

3 Implementing Contract-Aware Components

QoSCL allows the expression of functional and extra-functional properties in a software component. The declared properties are useful to the software designer because this gives predictability to a component’s behaviour. However, this predictability is valid only if the component implementation really has the behaviour declared by the component. This implementation validity is classical software validation problem. Standard software validation techniques deal with pre/post-condition contract types [6]. Protocol validation extends this to the “level 3” contract types (behaviour) [4][8]. The rest of this section discusses issues of testing extra-functional property conformance.

3.1 Testing Extra Functional Behaviour

Level 3 contracts \(i.e.\) contracts that include protocols) are more difficult to test because of non-deterministic behaviours of parallel and distributed implementations. One of the most difficult problems is the consistent capture of data on the behaviour of the system's elements. Level 4 contracts \(i.e.\) extra-functional properties) are also difficult to test for quite similar reasons. Our approach for testing level 4 contracts relies on the following features:

- existence of probes and extra-functional data collection mechanisms (monitors);
- test cases;
- oracles on extra-functional properties.

In order to be testable, a component must provide probe points where basic extra-functional data must be available. There are several techniques to implement such probe points and make performance data available to the test environment.

1. The component runtime may include facilities to record performance data on various kind of resources or events (e.g. disk operations, RPC calls, etc). Modern operating systems and component frameworks now provide performance counters that can be "tuned" to monitor runtime activity and therefore deduce performance data on the component's service.

2. The implementation of the component may perform extra computation to monitor its own performance. This kind of “self monitoring” is often found in components that are designed as level 4 component from scratch (\(e.g.\) components providing multimedia services).

3. A component can be augmented with monitoring facilities by weaving a specific monitor piece of model or of code. Aspect-oriented design (AOD) or aspect-oriented programming can help in automating this extension.
We have chosen this latter approach as our main technique for designing monitors. This choice was motivated mainly by the existence of “legacy” components from industrial partners [9]. From a software design process point of view, we consider that designing monitors is a specialist’s task. Monitors rely on low level mechanisms and/or on mechanisms that are highly platform dependant. By using aspect-oriented design (AOD), we separate the component implementation model into two main models: the service part that provides the component’s functional services under extra-functional contracts, and the monitor part that supervises performance issues. A designer in charge of the “service design model” does not need to master monitor design. A specific tool¹ (a model transformer) [10] is used to merge the monitor part of the component with its service part.

More precisely, a contract monitor designer provides component designers with a reusable implementation of a monitor. This implementation contains two items: a monitor design model and a script for the model transformer tool (a weaver). The goal of this aspect weaver is to modify a platform specific component model by integrating new QoSCL classes and modifying existing class and their relationships.

3.2 A Practical Example of Weaving

As shown in section §2.3, the QoSCL allows us to model in the same UML diagram structural, behavioral and contractual features. The QoS aspect weaver is a mechanism integrated into Kase, which:

- modifies the UML diagram (add new classes and associations)
- modifies the behavior of the targeted service

Thanks to QoSCL, it is possible to specify into Kase the contract types and their implementation such as TimeOut and TimeOutC (Fig. 4). According to our vision, detailed in the QoSCL section (§2.3), the TimeOut contract is an interface, which has a special operation denoting the “delay” dimension. The TimeOutC is a .Net class that implements the TimeOut interface. The value of the “delay” dimension is implemented like a private attribute (-delay:double) and its related access/evaluation method (delay():double).

A QoS aspect not only specifies how the structural diagram will be modified, but also how the monitored part and the monitor cooperate: when does the timer start, when does it stop, who handles timeout, etc... This part of the aspect is specified by using the Hierarchical Message Sequence Charts (HMSC) of UML2.0. In the Fig. 5 shows the behavior of a contractual service, called op(), as a HSMC diagram. The op() operation is the service which must verify a TimeOut contract. The op_c() operation is a new operation, which realizes the op() service and evaluates the TimeOut contract. above (Fig. 5). This service has two possible behaviors, according to if the op() service finishes before or after the timer.

¹ The Kase tool is developed by TU-Berlin with the support of the European Project “Quality Control of Component-based Software” (QCCS) [9].
In addition of its structural (Fig. 4) and behavioral (Fig. 5) parts, a contractual QoS aspect has pre-conditions. For example, a :Cl class can verify a TimeOut contract under the condition that it implements the op() service of course. The pre-conditions specify the conditions where the aspect can be weaved. In the implemented tool, the aspect is concretely weaved in the UML diagram by a Python script, which:

- checks the aspect pre-conditions;
- weaves the aspect if they are checked successfully (add new classes, modify constructors and operations, etc...)
The QoS aspect weaver implemented in the Kase tool allows us to specify a QoS aspect and to implement an evaluation of this aspect for a targeted service. According to the QoSCL point of view, contracts can be specified at design time as specialized interfaces. So, at binding time, it is easy to connect two components through their respectively compliant required and provided interfaces (PortQoSCL). At run time, the QoS aspect weaver implemented in Kase allows to implement any contract type in C#. In case of failure, on extra-functional contract can be renegotiated. For instance, a time out contract that fails too often obviously needs to be adjusted (or else the service bound to it has to be shut down).

3.3 Limitations of Extra-Functional Property Testing

The QoSCL notation and the monitor integration technique helps the component designer to define and check extra-functional properties. However, application designers rely on component assemblies to build applications. These designers need to estimate at design time the overall extra-functional properties of a given assembly. Using the techniques presented above, they can perform a kind of integration testing. The tests aim at validating the extra-functional behavior of the assembly with respect to the global specification of the application. However, the application designers often have trouble to select and configure the components, make the assembly and match the global application behavior. Conversely, some applications are built with preconfigured components and the application designer needs to build a reasonable specification of the overall extra-functional behavior of the application.

4 Towards Better Extra-Functional Property Dependencies

4.1 QoSCL at Design Time

QoSCL is a metamodel dedicated to specify contracts whose extra-functional properties have explicit dependencies. Models can be used by aspect weavers in order to integrate the contractual evaluation and renegotiation into the components. However, at design time, it is possible to predict the global quality of the composite software.

Predicting a behaviour is difficult. In the best cases, very limited and generally unrealistic, the behaviour can be proved. Otherwise, the behaviour is predicted with uncertainty. Since we want to predict the quality of a composite, i.e. the value of a set of extra-functional properties, this uncertainty will be translated into a numerical interval or an enumerated set of values, called validity domains.

The dependencies defined in QoSCL, which bind the properties, are generally expressed either as formulae or as rules. The quality of a service is defined as the extra-functional property’s membership of a specific validity domain. Predicting the global quality of a composite is equivalent to the propagation of the extra-functional validity domains through the dependencies.
For instance, we have defined in section 2.2 a set of extra-functional properties that qualifies different services in our GPS component-based model. In addition, we have specified the dependencies between the extra-functional properties as formulae. This knowledge can be specified in QoSCL. The Fig. 6 below represents the GPS component-based UML2.0 diagram (Fig. 1) refined with contractual properties and their dependencies:

![Diagram of GPS component-based UML2.0 diagram](image)

**Fig. 6.** GPS extra-functional properties and their dependencies.

A such diagram makes it possible to answer at designer what is the impact of his specific requirement on a service on the other services. For instance, we would like to know what is the response time of the `getLocation()` service if the satellites emits their signals in the [15;30] interval. Conversely, the designer can also require that the response time for the `getLocation()` service must be less than 25s with a high precision. He wants to know if his requirement is compatible with the satellites specifications.

Our aim is to exploit at design time all the knowledge that is available with QoSCL about the extra-functional properties, and hence give the designer answers to his questions.

### 4.2 The Four Degrees of Conformance

A PortQoSCL is typed by its functional Interface and its ContractType. As we underlined in section §2.3, the assembly of two components via their respective provided and required PortQoSCL is valid if and only if the functional Interface compliance and the extra-functional ContractType compliance are checked.

That is the two first degrees of the conformance that we can check at the connection point. However, the contractual dependency between the Dimensions introduces two other degrees of conformance. Indeed, a Dimension is defined at the Interface level. The main feature of a Dimension is its value. However, at design time, a such knowledge is not yet available. But information about the intervals, to which the values belong, are available. This knowledge, in addition to extra-functional dependencies, implies four degrees of conformance for component-based diagram refined with extra-functional properties:
- 1st degree: the functional conformance. That is the well-known conformance between provided and required typed interfaces.
- 2nd degree: the extra-functional conformance. The extra-functional properties $\Lambda = \{\lambda_i\}$ are defined at the interface level. The whole of the required properties $\Lambda_R$ must be provided ($\Lambda_R \subseteq \Lambda_P$, where $\Lambda_P$ is the set of provided extra-functional properties). This conformance is very close of level 1 (see QoSCL).
- 3rd degree: each extra-functional property is a valuable quantity which belongs to an enumeration or a numerical interval, called validity domains and denoted $\Omega(\lambda)$. For each required property $\lambda_i$, the intersection of its required validity domain $\Omega_R(\lambda_i)$ and its provided validity domain $\Omega_P(\lambda_i)$ must be not empty.
- 4th degree: there exists a system of equations $S(\lambda_1, \lambda_2, \ldots)$ which binds together the required and provided extra-functional properties of a component. The system must be verified for a sub-set of the validity domains where the properties are defined ($\Omega(\lambda_1) \times \Omega(\lambda_2) \times \ldots$).

### 4.3 Solving Conformance with Constraint Logic Programming

As we have shown previously, the extra-functional properties are valuable quantities, stressed by numerical constraints at the interface level, and bonded between them at the component level. These features are common with the Constraint Logic Programming (CLP) [11].

The CLP is an evolution of the logic programming, that integrates the interval arithmetic. The logic programming is an efficient tool to induce or deduce properties from rules and facts. However, the inference engines are unable to prove properties on the numerical domains where are defined numerical variables. The interval arithmetic fills this gap. This concept was originally implemented by J.G. Cleary [12].

A CLP engine integrated in a design tool such as Kase allows the designer to bind the extra-functional properties and to propagate the numerical information through all the components diagram. The designer will get a precious a priori knowledge about the quality of its component. Moreover, this information will be still enriched on the fly by new connections or new contracts added by the designer. He can visualize directly the influence of its actions on the global quality of its component assembly.

The contractual specifications are written in QoSCL. Although this language has all the features of a CLP compliant language, it was not originally dedicated to this specific use. Implementing a dedicated CLP solver engine, with its interval arithmetic solver, is not an easy task. Because of the common features, it is more judicious to transform QoSCL into an existing CLP language, using a MOF compliant model transformation such as MTL [13]. The result of a transformation of the GPS component-based model with QoSCL into a PrologIV program for example is:

```
00- %% CONTRACTUAL DEPENDENCIES
01- => receiver( ThetaR, ThetaS) :-
02- ThetaR ~ ThetaS + 2.
03- => decoder( ThetaD, ThetaS) :-
```
The three first predicates (receiver/2, decoder/2 and computer/5) reify the dependencies of the extra-functional properties defined on each component. The rule/3 predicate is used by the computer/5 predicate in order to bind the Eps, P and Nbr variables. The lines 25 to 27 is a request, which is made up of two parts:

1. line 25: the ThetaS (resp. Nbr, P, Eps) variable belongs to the [15;30] (resp. [3,5,12], [0;15], {high}). These numerical constraints are either induced by the specifications and the environment, or by a designer who wants to know the global extra-functional impact of a specific stress applied on one or more properties.
2. line 26-27: represents the connection between the components, extracted from the component-based model. The extra-functional properties can be shared at this level between two components, like the ThetaD variable is shared by the decoder and the computer. It is also a request that the designer can answer. The answer computed by the inference engine is:

4.4 Inference and Result

In CLP, it is important to underline the fact that the use of numerical predicates (+/3, -/3, ...) on a variable implies that this variable belongs to the real R or the integer N sets
by construction. However, this kind of information induces few new knowledge by propagation and reduction of the intervals. That is the reason why, in line 25, we stress some properties in order to reduce their validity domains. These constraints are either provided by the specifications of the device \((\theta_s \in [15;30], \text{nbr} \in \{3;5;12\})\) or by designer’s requirements \((P \leq 15, \text{eps}=\text{high})\).

The computation of the current location has a power cost \(P\), which depends on the number of active receivers \(\text{nbr}\) according the simple formula:

\[
P = \text{nbr} \times 3.
\]  

(4)

According to (4) and the requirement that the power must be less or equal than 15, we obtain that the number of active receivers must be equal to 3 or 5:

\[
\text{nbr} \in \{3;5\}.
\]

(5)

The lines 15 to 22 express a rule that binds the precision \(\text{eps}\), the response time \(\theta_c\) and the power \(P\) of the \text{getLocation()} service. According to this rule, the constraints (4) and (5), and the fact that \(\text{eps}\) is required to be high, it is obvious that:

\[
\text{nbr} = 3, \quad P = 15.
\]

(6) (7)

The relationship that binds the response time \(\theta_c\) to the response time \(\theta_b\) and the number of active receivers \(\text{nbr}\) is:

\[
\theta_c = \theta_b + \text{nbr} \times \log(\text{nbr}).
\]

(8)

The second term \((\text{nbr} \times \log(\text{nbr}))\) is the time spend by the computer to interpolates the position from the data. Since the validity domain of \(\theta_b\) and the value of \(\text{nbr}\) are known, we have:

\[
\theta_c \in [23.49; 24].
\]

(9)

This interval can be propagated again through the response time chain of dependencies (Fig. 6), and finally we obtain the result on \(\theta_b\) and \(\theta_s\):

\[
\theta_b \in [20; 20.5],
\]

(10)

\[
\theta_s \in [15; 15.5].
\]

(11)

That is the result obtain by the CLP inference engine (l. 29-34).

5 Related Works

QoS and extra-functional properties are not a new concept in the component-based software engineering [1][8]. Many authors have developed languages dedicated to extra-functional specifications: SMIL [14], TINA-ODL [15], QIDL [16], QDL [17] and so on. An interesting analysis of these languages and their common features is done by Aagedal [18]. He concludes that the QoS Modeling Language (QML) [7] is
the most general language for QoS specification. Indeed, the concepts defined in QML are representative of specifications languages dedicated to the extra-functional aspects. These concepts have been integrated into QoSCL.

QML has three main abstraction mechanisms for the QoS specification: contract type, contract and profile. A contract type represents a specific QoS aspect, such as reliability for example. It defines valuable dimensions, used to characterize a particular QoS aspect. A contract is an instance of a contract type and represents a particular QoS specification. Finally, QoS profiles associate contracts with interfaces.

With QML, it is possible to write a contract which specifies that an operation must be completed with a delay less than 30 seconds:

```
type TimeOut = contract {
    delay : decreasing numeric s;
}
TimeOutImpl = TimeOut contract {
    delay < 30;
}
TimeOutProfile for ComputerI = profile {
    from getSignal require TimeOutImpl
}
```

In spite of its generality, QML, as the other languages mentioned above, does not have explicit dependencies between extra-functional properties. The properties are considered as independent quantities, evaluated at run time by monitors. But that does not match reality: we have shown that extra-functional properties have dependencies in the same way as a provided service depends on a set of required services. QoSCL makes these dependencies explicit.

The QoS specifications can be used either to implement contracts evaluation and renegotiation at run time, or to evaluate a priori the global quality of the assembly at design time.

G. Blair has proposed a specific architecture in order to manage component composition, based on a reflective component model [19]. In fact, the component reflection is dedicated to access at properties value and structural information of the composite. A set of rules, called StyleRule, manages the adaptation of the composite to its environment. In fact, these rules can be considered as model transformations used to reconfigure the composite: properties values, connections between components and graph actions.

In the various models shown, the extra-functional properties are not explicitly defined, neither their dependencies a fortiori. Consequently, it is not possible to predict at design time the global quality of the assembly.

Moreover, according to us, the use of rules in order to dynamically configure an assembly at run time is dangerous. Indeed, the authors of [20], which have implemented a declarative approach for adaptative components, underline the limit of such approach: the set of rules which governs the adaptation must respect completeness and uniqueness.

Completeness guarantees that for every possible situation there exists at least one adaptation action, and uniqueness ensures that this action is unique. The authors indicate that the two following properties can be enforced by use of the CLP. However, not only these two properties are not compatible with non-linear constraints, but also the extra-functional dependencies can be non-linear (see formula #8).
That is the reasons why we have chosen to implement the contracts evaluation and renegotiation in imperative language with a weaving aspect technology. Like Genßler and Zeidler [21], the use of CLP is kept till the design time, to check the validity of an assembly according to a set of rules (consistency rules, contractual rules, etc.). However, none of them exhibits a component model with explicit extra-functional dependencies.

At last, researchers of the Software Engineering Institute at Carnegie Mellon University (USA) have underlined the importance to integrate more analysis technology in components-based software [22]. We think that this integration must be considered at the highest level: the component model. QoSCL (specification) and its dedicated tools (validation and renegotiation) presented in this paper are a contribution in this sense.

6 Conclusion and Future Work

In mission-critical component based systems, it is particularly important to be able to explicitly relate the QoS contracts attached to provided interfaces of components with the QoS contracts obtained from their required interfaces. In this paper we have introduced a language called QoSCL (defined as an add-on to the UML2.0 component model) to let the designer explicitly describe and manipulate these higher level contracts and their dependencies. We have shown how the very same QoSCL contracts can then be exploited for validation of (1) individual components, by automatically weaving contract monitoring code into the components and (2) a component assembly, including getting end-to-end QoS information inferred from individual component contracts, by automatic translation into a CLP language.

Both validation activities builds on the model transformation framework developed at INRIA (cf. http://modelware.inria.fr). Preliminary implementations of these ideas have been prototyped in the context of the QCCS project [9] for the weaving of contract monitoring code into components part, and on the Artist project (http://www.systemes-critiques.org/ARTIST) for the validation of a component assembly part. Both parts still need to be better integrated with UML2.0 modelling environments, which is work in progress.

References

CB-SPE Tool: Putting Component-Based Performance Engineering into Practice

Antonia Bertolino¹ and Raffaela Mirandola²

¹ Istituto di Scienza e Tecnologie dell'Informazione “A. Faedo”, CNR
56100 Pisa, Italy
² Dip. Informatica, Sistemi e Produzione, Università di Roma “Tor Vergata”
00100 Roma, Italy

Abstract. A crucial issue in the design of Component-Based (CB) applications is the ability to early guarantee that the system under development will satisfy its Quality of Service requirements. In particular, we need rigorous and easy-to-use techniques for predicting and analyzing the performance of the assembly based on the properties of the constituent components. To this purpose, we propose the CB-SPE framework: a compositional methodology for CB Software Performance Engineering (SPE) and its supporting tool. CB-SPE is based on, and adapts to a CB paradigm, the concepts and steps of the well-known SPE technology, using for input modeling the standard RT-UML PA profile. The methodology is compositional: it is first applied by the component developer at the component layer, achieving a parametric performance evaluation of the components in isolation; then, at the application layer, the system assembler is provided with a step-wise procedure for predicting the performance of the assembled components on the actual platform. We have developed the CB-SPE tool reusing as much as possible existing free tools. In this paper we present the realized framework, together with a simple application example.

1 Introduction

Component Based Software Engineering (CBSE) is today the emerging paradigm for the development of large complex systems. By maximizing the re-use of separately developed generic components, it promises to yield cheaper and higher quality assembled systems [19].

The basic understood principle (or actually aspiration) is that the individual components are released once and for all with documented properties (as, e.g., in [13] or [16]) and that the properties then resulting for an assembled system can be obtained from these in compositional way. While this principle/aspiration has been actively pursued for the system functional properties since the advent of CBSE, it is only recently that equal emphasis is being devoted to the as important non-functional aspects or Quality of Service (QoS), such as reliability, security and performance (see papers in [5] and [6,10,17,22,26]). Indeed, in the context of CB production, among multiple component implementations accomplishing a similar functional specification, preference will be given to those that provide better QoS properties.

In this paper we focus on the evaluation of performance properties (like response time, throughput, etc.) and propose a framework for the automated compositional...
performance prediction and analysis of CB systems, called CB-SPE, which stands for Component-Based Software Performance Engineering.

CB-SPE is a generalization of the Software Performance Engineering (SPE) [18], which is a systematic, quantitative approach for the careful and methodical assessment of performance attributes throughout the lifecycle, from requirements and specification to implementation and maintenance. The original contribution of our CB-SPE approach over the existing SPE is two-fold: on one side, we equipped the CB-SPE of an input modeling notation that is conformant to the standard RT-UML PA sub-profile [23]; this makes the approach more easily usable, also from those designers who are not expert of the specialized (and awkward) performance models but are familiar with the widespread UML language. On the other side, CB-SPE has been conceived for the analysis of component-based systems, so that the performance of the assembled system can be compositionally derived based on the system architecture (the glue) and on the documented performance parameters of its components.

We have earlier illustrated in a different application context how the RT-UML PA sub-profile could be adopted to provide an easy-to-use and standard interface to SPE [3]. With regard to the generalization of SPE to a CB paradigm, in [4] we anticipated the inspiring ideas. We have now refined the methodology and completed the development of the supporting tool. In this paper we present the detailed steps of the approach and the architecture of the CB-SPE tool, through its application to a simple case study. In the realization of the CB-SPE tool the underlying philosophy has been to re-use, as much as possible, existing tools. Thus, the UML editing is carried out through the well-known Argo-UML [28], while the obtained performance (queueing network) model is solved by means of RAQS [29].

The paper is organized as follows: in the next section we overview the CB-SPE framework; in Section 3 some details of the underlying methodology are given. In Section 4 we describe the CB-SPE tool structure and functioning, and finally in Section 5 we outline an example of application. Related work is dealt with in Section 6 and Conclusions are drawn in Section 7.

2 Overview of the CB-SPE Framework

As its name implies, CB-SPE is proposed as a generalization of the well known SPE approach [18] to CB systems. We generically consider component-based applications, built up from software components glued together by means of some integration mechanism. In this context, the components provide the application-specific functionalities (and are considered as black boxes), and the glue defines the workflow that integrates these functionalities to deliver the services required from the CB application. Correspondingly, CB-SPE consists of two layers: the component layer, pertaining to the component developer, and the application layer, pertaining to the system assembler (SA). Figure 1 below provides an overview of the intended context and usage for the framework. On the left, we represent the component developer’s side. This is expected to fill the “Component Repository” with components whose interfaces explicitly declare the component predicted performance properties. Roughly, this implies that the component developers must introduce and validate the performance requirements of the component considered in isolation. In the context of CB-SPE, the goal is to have an estimation of the component performance properties that is
platform independent (so that we can re-use them on the yet unknown platform of the CB application). So in CB-SPE we need to define component models which are parameterized with respect to platform parameters. We provide further explanation of how we intend to do this in Section 3.1. In [8], we have started defining an abstract notation suitable for component modeling, and a related methodology for eliciting the core information required for our compositional analysis.

On the right side of Figure 1 we show the scenario relative to the SA usage of the framework. First, the SA should identify the different types of application users and the relative Use Cases. In general, there will be several types of users and a number of Use Cases, whereby each Use Case $z$ must be weighted by a factor $P(z)$ representing its usage frequency, as illustrated in [7]. The collection of all the Use Case probabilities ($\forall z, P(z)$) represents an usage profile. For the performance goals, the SA, according to the whole application and to its QoS requirements, can decide to assign (as done for instance in [12]) different importance levels (weights) $w_k$ to the various individual performance metrics $k$ (e.g., response time, utilization, throughput). $w_k$ is a relative importance weight and all $w_k$s sum up to 1.

Then, the SA chooses among the components available in the repository those that better fulfil the settled performance requirements. In fact, at this stage, he/she knows the characteristics of the environment in which the components will be deployed and thus can instantiate the component performance properties given by the component developer in parametric form. A possible metric proposed to support the selection is described in Step 1 of Section 3.2.

Once a set of suitable components is selected, the SA can proceed with the system modelling and annotation following the RT-UML profile, as further detailed in step 2 of Section 3.2.

---

1 A more precise usage profile definition can be performed following the approach proposed in [9].
As also evidenced by the figure, the CB-SPE tool plays a central role towards putting into practice such a scenario. The various component performance properties are in fact combined according to the application architecture by use of the CB-SPE tool. In Section 4 we describe the main functions performed by the tool, and also illustrate its architecture.

Finally, based on the analysis results provided by the tool, the SA can reach a more informed decision about the system design. If the performance requirements are fulfilled, he/she can proceed with the acquisition of the pre-selected components and their assembly; otherwise he/she has to iterate the process by repeating the steps described, or lastly admit the unfeasibility of the performance requirements with the acquisition of off-the-shelf components.

3 The Underlying Methodology

In this section we provide further details for some specific parts of the implementation of the CB-SPE methodology above outlined.

3.1 Component Layer (Component Developer’s Side)

Let us suppose that a given component \( C_i \) offers \( h \geq 1 \) services \( S_j \) (\( j = 1 \ldots h \)). Each offered service could be carried out either locally (i.e., exploiting only the resources of the component under exam) or externally (i.e., exploiting also the resources of other components).

The obtained performance analysis results for each service \( S_j \) of component \( C_i \) are exported at the component interfaces, in parametric form, as:

\[
\text{Perf}_{C_i}(S_j|\text{env-par}^*)
\]

where \( \text{Perf} \) denotes the performance index we are interested in (e.g., demand of service, response time and communication delay) and \( |\text{env-par}^* \) is a list of environment parameters (e.g., bandwidth, CPU time, memory buffer). In particular, when the service is local, \( |\text{env-par}^* \) represents the classical platform parameters, such as CPU demand or communication delay, of the node where the component is deployed; while in case of external services, \( |\text{env-par}^* \) parameters include also demands of either different software components or resources deployed on nodes that could be different from the one hosting the considered component.

As said, the modeling notation we adopt for CB-SPE is based on the standard RT-UML PA profile. Considering in particular the component layer, starting from the RT-UML diagrams describing the component, one of the existing approaches [24] can be used to (automatically) derive its performance properties. These results should then be exported at the component interface following the RT-UML syntax, as schematically illustrated in the left side of Figure 1.

At the component level we would thus obtain a model that explicitly takes into account the kind of expected (compatible) environment, yet is still independent of any specific technology (similarly to the ideas introduced in [15]).
3.2 Application Layer (System Assembler’s Side)

The goal of the application layer is to obtain CB applications with the expected performance properties, by the assembly of components whose parametric properties are known. The first phase of determination of usage profile and performance goals has already been discussed in the previous section. In the following, we provide details of the steps the SA should perform to “package” a useful input to the CB-SPE tool and to correctly interpret/utilize the tool results.

**Step 1. Component Pre-selection.** The system assembler chooses among the components that offer similar services, those that provide the best performance. In fact, he/she can instantiate the generic \( \text{Perf}_c(S_j|\text{env-par}|^*) \) given in the component interfaces, with the characteristics of the adopted (or hypothesized) environment, so obtaining a set of values among which the best ones can be selected.

To carry out this choice we propose a metric derived from [12] and reshaped according to this context; we call it perf-err. This metric aggregates several performance metrics in a way that is independent of the units of measure of the individual metrics. It increases as the value of an individual metric improves with respect to its bound, while decreases as the value of an individual metric deteriorates with respect to its bound. It is computed as:

\[
\text{perf-err} = \sum_{k=1}^{n} w_k \cdot f_k(\Delta_k)
\]

where \( n \) is the number of metrics being aggregated, \( w_k \) is the relative importance weight for metric \( k \), \( \Delta_k \) is a relative deviation of the performance metric \( k \) defined in a way that the relative deviation is positive when the performance metric satisfies its goal and negative otherwise, and \( f_k(\ldots) \) is an increasing function of \( \Delta_k \).

An example expression for the relative deviation of CPU demand \( D \) is:

\[
\Delta D = \frac{(D_{\text{required}} - D_{\text{offered}})}{D_{\text{required}}}
\]

\( \Delta D \) is positive when the goal is satisfied, namely when the CPU demand \( D \) related to a component is less than (or equal) to the target \( D \).

**Step 2. Modeling and Annotation.** In this step, the SA should describe, by one or more sequence diagrams (SD), the application workflow (the glue). In a CB framework the SD objects represent the involved components, and the SD messages represent the requests of execution of a component service or correspond to information/data exchanged between the components. Moreover the SA should construct a Deployment Diagram (DD) modeling the available resources and their characteristics. In this case the nodes of the DD can be associated to classical resources (device, processor, database) and communication means. Note that the same diagrams can be re-used for similar applications, by only updating the associated parameters.

The SA should express, by using a comment-based annotation, the attributes associated with events and actions of the diagram. In particular, the PA attributes of the

---

2 Alternatively, the “glue” could also be modelled with annotated Activity Diagrams.
(closed) workload will be associated with the first action of the SD; those of the steps will be linked to each of the subsequent message activations; those of the resources will be related to each of the nodes of the DD. For example, considering the SD, classical examples of PA attributes are: the population, that represents the number of jobs in the scenario, and the response time, that represents the application completion time and is one of the expected results. Considering the DD nodes, examples of PA attributes concern the resource scheduling policy (i.e., the strategy by which the resource handles the different jobs), the resource utilization and the resource throughput (the amount of work provided per unit of time by a resource belonging to a certain node).

For the adopted PA annotations, we have tried to adhere to the RT-UML standard as much as possible; however, there are some aspects, such as message size or use case probability, that are essential for the generation of the performance model, but are not covered by the standard (as also noticed in [3]). For these aspects, that are few ones anyway, we have followed the policy of including “minimal” lightweight extensions. These are explained in the example application.

At this point the input for the CB-SPE tool has been completed. Two kinds of results can be obtained from the tool: best-worst case and contention based results, as described in the next section. Thus, the remaining SA responsibility is to perform the design choices in the final step.

Step 3. Analysis of Results. The results provided by the CB-SPE tool are analyzed by the SA and, if different from those expected (or desired), he/she can go back to step 1 (or 2), modify the settled parameters, and repeat the process until the desired results are obtained or after a while declare the unfeasibility of the performance requirements.

4 The CB-SPE Tool

In this section we describe the main functionalities of the CB-SPE tool components, while the details concerning its application to a simple case study will be given in the next section. The tool architecture is illustrated in Figure 2, where the blocks represent the constituent modules of the tool, while the “rounded rectangles” exemplify the XMI exchanged information between them.

We aim at the rapid development of an inexpensive proof-of-concept tool implementation to validate the CB-SPE methodology. Therefore our approach is to re-use as much as possible existing free tools. For this reason, to make the SPE calculations, we choose not to implement from scratch a solver, but to transform the RT-UML model into a QN model [11], to be then processed directly by existing solvers [11,18,24,29].

For UML editing we use the well-known Argo-UML [28] tool. Argo (specifically augmented to process also the PA annotations) generates an XML/XMI file that constitutes the input to the other modules.

---

3 Generally, a workload can be either closed, in which a fixed number of requests cycles while the scenario is executed, or open, in which the requests arrive at given (predetermined) rate. In this paper, for simplicity, we consider only closed workload.
The first encountered block is the “Best-Worst case Analyzer”. This analytically provides an optimistic and pessimistic bound on the expected performance for each component choice, and can help the system assembler in pre-identifying a subset of components deserving further investigation.

The best-case analysis corresponds to the special case of a stand-alone application, i.e., where the application under study is the only one in the execution environment (therefore there is no resource contention), and the application dynamics is taken into account only in terms of component usage frequency. To compute a pessimistic bound on the application performance, instead, we suppose that $M$ (equal or similar) applications are running on the same environment and are thus competing for the same resources. In the worst case, we suppose that the $M$-th application, to obtain its required service must wait, every time, for the completion of the other $M-1$ applications.

After these bounds are computed and deemed acceptable, the XMI file enters our Model Generator module. This component (realized in Java) provides, according to the SPE basic principle, two different models (the two outgoing arcs): a stand-alone performance model, namely an Execution Graph (EG) [18], and a contention based performance model, namely a queueing network (QN) model.

Let us consider first the round-trip tour for the EG model (darker arrows). Based on the methodology described in [3,7] a global EG is generated from the models of key performance scenarios represented in the SDs (according to their occurrence

**Fig. 2.** CB-SPE Tool Architecture.
probability) and its demand vectors are instantiated according to the environment parameters given by the DD. The output of the Model Generator is an XMI document representing the EG.

The link to the “Visual Modeler” is currently under development. Visual Modeler is a tool developed at the University of Roma Tor Vergata [20] that supports the graphical definition of EG and QN. An example EG with possible demand vector is illustrated in Figure 3. The developed EG Solver module then applies standard graph analysis techniques [18] to associate an overall “cost” to each path in the obtained EG as a function of the cost of each node that belongs to that path. This stand-alone analysis result gives, for each resource, the total average demand, that, in the optimal case corresponds to the average response time [11].

The Result Converter module receives in input both the application performance goals in terms of PA annotations for the different key performance scenarios and the performance results provided by the EG Solver component. A simple comparison can give insights about the hypothesized component and environment selection. Specifically, we apply, for each scenario, the metric \( \text{perf-err} \) defined by formula (1) by substituting the offered performance with the EG analysis results. Then we can state whether for the application model the selected components fulfil the performance requirements, or otherwise we have some information about the distance (\( \Delta k \)) from the desired performance.

Let us now consider the second output of the Model Generator module that is the QN model. QN model generation is a quite complex task, requiring to determine:

A. The number of service centers and their characteristics (service discipline, service times)
B. The network topology
C. The jobs in the network with their service demand and routing.
For A. and B. the necessary information can be derived from the annotated DDs and SDs. For a first assessment, for example, we associate a QN service center to each DD node. The service center scheduling discipline is derived from the PA annotation and the service can be considered exponential with rate equal to the node throughput (also annotated on the DD). Similarly, the links among the service centers are derived from the ones existing in the DD. The kind of QN (closed, open or mixed) corresponds to the characteristics of the modeled workload (closed, open and mixed, respectively) annotated on SDs.

Concerning the job characteristics (C.) we derive all the essential information from detailed models of key performance scenarios represented in the EGs. The different job classes on the network model the distinct key performance scenarios. The job service demand at the network service centers can be computed from the resource demand vectors of the EG. Similarly, the job routing corresponds to the application dynamics given by the whole EG.

Also for QN we have defined an XMI format to guarantee the interoperability of different tools.

The “Model Solver” component for the QN is a tool, called RAQS (Rapid Analysis of Queueing Systems), developed at the “Oklahoma State University”. An important element in the choice of RAQS was the possibility to describe the input information in textual form. RAQS allows for the definition and solution of different types of queueing networks with growing complexity levels [29].

5 Example of CB-SPE Application

The adopted case study is a simplified version of a software retrieval application described in [14], composed by 3/4 components that interact through different network kinds depending on their physical location. The main components are illustrated in Figure 4. There is a user that, through a specialized user interface can select and download new software in an efficient way. A software manager agent obtains the desired catalog by interacting with a database and then creates a dedicated catalog agent that helps the user to select the software and performs all the operations required for the desired service.

![Fig. 4. The application example.](image-url)
The user can choose to download some software or to browse the catalog of offered services and then select the desired software or can require a catalog refinement. This process can be repeated as many times as necessary. As in [14] we assume that the user accesses the software retrieval service through a mobile device, which can exploit only a wireless cellular infrastructure with a low bandwidth. Instead, the interactions between the SW manager and the catalog agent or the DB take place through a high speed LAN. Now let us consider how modeling and analysis of this example can be carried out through the CB-SPE tool.

**Component Developer.** Let us suppose that each component is specified following the proposed approach and that the interface of each is annotated with the offered/required performance.

**SA-Usage Profile.** The system assembler should define the different types of application users and the different Use Cases. Let us suppose, for example, that the SA defines the target application \( A \) by using four components. \( A \) is composed by two different functionalities \( F_1 = \text{Browse} \) and \( F_2 = \text{Download} \), yielding a usage frequency of 0.4 and 0.6, respectively. For the performance aspects, the SA can decide, for example, that the performance metrics he/she is interested in are the \( \text{CPU elapsed time} \) and the \( \text{Communication delay} \), with an importance factor \( w_1 \) equal to 0.7 and \( w_2 \) equal to 0.3, respectively. The performance goal he/she wants to achieve is a global response time (i.e., CPU elapsed time + Communication delay) equal to 20 ms.

**SA-Component Pre-selection.** For simplicity we assume that each component offers a single service \( S \). Thus the annotations in the component interfaces that the SA must instantiate have the form: \( \text{CPU Demand}_{ci}(S[\text{env-par}]) \) and \( \text{Comm delay}_{ci}(S[\text{env-par}]) \). Then by applying formula (1) with the performance metrics of interest and their relative \( w_k \), it is possible to select the combination of components that optimizes the perf-err metric. At the end of this step we assume that the components “User-Interf”, “SW_man”, “Cat-agent” and “DB” are selected together with “LAN-1” and a “Wireless-1” network components (resources).

**UML Modeling and PA Annotation.** \( F_1 = \text{Browse} \) and \( F_2 = \text{Download} \) operations are modeled by means of Argo-UML using User-Interf, SW_man, Cat_agent and DB. Their interactions can be modeled by SDs (Figure 5 illustrates the SD of Browse operation). The first note of the SDs describes the kind of workload with its characteristics, using the stereotype \( \text{PAClosedLoad} \). We had to define for this stereotype a non-standard attribute (see end of Step 2 in Section 3), called \( \text{PA_{sd}} \), representing the occurrence probability of the modeled scenario. Each step is then annotated with the standard \( \text{PAstep} \) stereotype, where the attribute \( \text{PAextop} \) including the message size and the network name models the communication delay involved by the step. The available resources and the possible component/node mappings are illustrated in the DD of Figure 6 with standard PA annotations. Node MU deploys the “User-Interf” component, node SM the “SW_man” and “Cat_agent” components, and node LDB the “DB” component.

**CB-SPE Tool: Best-Worst Case Analyzer.** The results obtained with the best-worst case analyzer for the case study are illustrated in Table 1 by supposing a maximum of 10 users. The performance goal of a response time of 20 ms is feasible since it is
included in the range of best-worst case, thus we proceed with the next steps. However, this analysis does not consider resource contention and the obtained results could be misleading. In fact, a stand-alone analysis gives the best results, but by introducing resource contention even the most capable CPU can rapidly become a bottleneck. So a trade-off is necessary and the SA next considers not only the best, but also “more realistic” cases.

**CB-SPE Tool: Model Generator.** By the Model generator module we first derive an EG modeling the application example. The second output of the Model generator module is a QN model including the information of the software behavior modeled by the EG and the components deployment on the given platform according to the DD of Figure 6.

**CB-SPE Tool: Performance Results.** As already stated the performance measures we are interested in are the CPU-demand and the communication delay. The EG solver module applies first some graph reduction rules. The obtained results, shown in table 2, give the total CPU demand for the different resources and the communication delay required by the application both for the LAN and for the wireless network.

![Fig. 5. SD1 for Browse operation.](image)

### Table 1. Best-Worst case results.

<table>
<thead>
<tr>
<th></th>
<th>computation</th>
<th>communication</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEST CASE</strong></td>
<td>1.26704</td>
<td>4.241776</td>
<td>5.5088</td>
</tr>
<tr>
<td><strong>WORST CASE (N=10)</strong></td>
<td>12.6704</td>
<td>42.41776</td>
<td>50.5088</td>
</tr>
</tbody>
</table>

### Table 2. EG solver results.

<table>
<thead>
<tr>
<th></th>
<th>Computational cost</th>
<th>Communication cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MU</td>
<td>SM</td>
</tr>
<tr>
<td><strong>SD1, prob=0.4</strong></td>
<td>0.4</td>
<td>0.9846</td>
</tr>
<tr>
<td><strong>SD2, prob=0.6</strong></td>
<td>2.1846</td>
<td>1.2923</td>
</tr>
</tbody>
</table>
The results obtained from the QN solver are again the CPU elapsed time and the communication delay, but they are more precise, as they include the waiting times for the contended resources. Figure 7 shows the results obtained with RAQS. It is possible to observe both local (related to a single resource) and global (related to the whole network) performance indices: node 1 corresponds to MU, node 2 to Wireless-1, node 3 to SM, node 4 to LAN-1 and node 5 to LDB.

**SA: Analysis of Results.** Finally, considering the results obtained from the analysis, we can finalize the design of the software retrieval system by committing to the selected components, or discarding some of them. From Table 2, we can observe that, in the case of a stand-alone application, the performance goal is satisfied. Figure 7 shows the results in a contention-based environment for a maximum of 10 jobs in the network. In this case the obtained average time is greater than the required one. From Figure 7 a bottleneck in the network is identified: Node 2 corresponding to Wireless-1 component (resource). Therefore, the first suggestion (also from the “result converter” module) is to remove, if possible, the bottleneck. Let us suppose, for example, that “Wireless-1” component can be substituted by a “Wireless-2” component three times faster than “Wireless-1”.

Table 3 shows the comparison of results obtained by varying the number of jobs in the network in the two cases of “Wireless-1” and “Wireless-2” components. Note that, with the new configuration, in the case of 10 users the performance goal of a response time of 20 ms is satisfied. With CB-SPE tool, it is also possible to analyze how the global response time (ATN) and average waiting time in queue at node 2 (AvTIQ_2) increase by changing the number of jobs in the network.

As this simple example shows, our analysis can be done early in the development cycle with low effort, avoiding waste of time and resources along unsuccessful design attempts. Obviously, we need to validate the feasibility of our approach on more real-
CB-SPE Tool: Putting Component-Based Performance Engineering into Practice

6 Related Work

There are not many works on CB performance analysis and engineering, due both to the novelty of the topic and to its inherent difficulty. When the emphasis is on the quantitative evaluation of performance, the approach mostly applied is measurement.
[6,27]. In [27] a method is proposed for the measurement of performance distinguishing between application-specific metrics (e.g., execution time of various functions) and platform-specific metrics (e.g., resource utilization). The automation of the process of gathering and analysing data for these performance metrics is also discussed. The major drawbacks of this approach are that it is suitable only for already implemented systems and moreover the obtained results do not show general applicability.

A different approach that partially overcomes these difficulties is presented in [6] where starting from a specific COTS middleware infrastructure, a first step empirically collecting performance measures is followed by a second step in which the obtained results are elaborated to extend their validity to a more general setting. An inherent limitation of this approach is that it leads to sound results only for a specific platform.

An approach targeted to include predictability in performance behavior of CB systems is presented in [17] (and then in [5]). The basic idea is that the “behavior of a CB system must be compositional in order to be scalable” and this requires (as in our approach) that, in addition to the descriptions of component functional behavior, performance specifications are also included. A first attempt towards a compositional approach to performance analysis is then presented mainly based on the use of formal techniques. However, as the authors claim, an engineering approach to predictability on performance is a necessary ingredient to ensure predictable components.

The papers in [10,22] propose a prototype prediction enabled technology, called PECT that integrates component technology with analysis models. The main goal of PECT is to enable the prediction of assembly level properties, starting from certifiable components, prior to component composition.

In [25] a specific performance evaluation techniques, layered queueing modeling, is applied to generate performance models for CB systems. Software components are studied under different environments.

An approach similar to [3,7] is presented in [1], where starting from Use Case, Activity and Deployment diagrams with RT-UML annotations (augmented, in some cases, so to better fit performance features) a simulation model is derived and evaluated.

Finally, a different, mainly qualitative, approach to the performance predictability/analysis of CB systems is undertaken in [2,21], where the affinity between Software Architecture (SA) and Software Component (SC) technology is outlined and exploited. This affinity is related to different aspects: (i) the central role of components and connectors as abstraction entities, (ii) the correlation of architectural style and component model and frameworks, (iii) the complementary agendas followed by the SA and SC technologies: enabling reasoning about quality attributed, and simplifying component integration. Therefore, the basic idea underlying these works is to develop a reference model that relates the key abstractions of SA and CB technology, and then to adapt and apply some existing SA analysis methods, such as ATAM, QADP, etc.

7 Conclusions

This paper presented the CB-SPE framework: a compositional methodology for component-based performance engineering and its supporting tool. CB-SPE lays on, and
adapted to a CB framework, the concepts and steps of the well-known SPE technology, and uses for modeling the standard RT-UML profile. The methodology consists of a component layer, and of an application layer, in which the documented component properties are instantiated and composed to predict the performance properties of the assembled system. The CB-SPE tool has been realized re-using existing free tools such as Argo-UML and RAQS. Although this has not been discussed here, the approach can be easily generalized to allow in turn for incremental compositionality between applications.

Future work also includes the validation of the proposed methodology by its application to case studies coming from the industrial world. Our long-term goal is in fact to include this tool in a general framework for the computer assisted discovery and composition of software components, encompassing both functional and non-functional properties.

Acknowledgements

Work partially supported by the MIUR-PRIN project “SAHARA: Software Architectures for heterogeneous access network infrastructures”, and by Ericsson Lab Italy, Rome, in the framework of the Pisatel Laboratory.

References

Component Technology and QoS Management

George T. Heineman, Joseph Loyall, and Richard Schantz

BBN Technologies
10 Moulton Street
Cambridge, MA USA
{gheinem,jloyall,rschantz}@bbn.com

Abstract. Component-based technologies make it possible to decompose complex software systems into functional software components. Unfortunately, most component technology does little to address non-functional Quality of Service (QoS) requirements that concern the run-time execution of the software. We are developing a technique to enable one to satisfy QoS requirements of software systems by decomposing and assembling Qoskets similar to the way systems are decomposed into functional components. A qosket is a packaged unit of reusable QoS-related behavior and policy. These qoskets are integrated into a component-based application to enable dynamic, adaptive QoS management. We present our design and describe our initial prototype. We conclude with a set of open questions and directions for future work.

1 Introduction

As software systems grow in size and complexity, it becomes increasingly difficult to ensure both the desired functional requirements (what the system does [2]) and non-functional requirements (how the system should accomplish the desired functionality given “the constraints of a non-ideal world” [3]). Some non-functional requirements (NFRs) are related to the process of software engineering, such as, maintainability, portability, or extensibility. We are concerned with NFRs related to the run-time execution of the software, such as performance or network bandwidth utilization. These concerns are often classified under the larger concept of Quality of Service (QoS), a term first used by the communications and networking community to describe the ability to measure and guarantee transmission rates over computer networks. The QoS concept extends to involve any “constraint” [3] that might impact the execution of software, such as CPU utilization, network bandwidth usage, or power consumption.

Advances in Component Technology [4] address the complexity of software by decomposing software into software components that conform to a specific component model. Components reduce the labor-intensive nature of software development by enabling application developers to package and organize code in a more formal, higher level manner. Components also enable a shift in perception from writing software systems from scratch to assembling systems using pre-existing software components.
Many believe that the only way to ensure end-to-end QoS requirements within a software system is to embed QoS-related statements throughout the software system; this approach leads to non-reusable software and maintenance difficulties. BBN has for a number of years investigated ways to incorporate aspects of QoS management as extensions to standards-based middleware using Quality Objects (QuO) [7]. One key objective of this approach to QoS management is to develop software that can rapidly adapt to the changing run-time conditions often found in distributed, real-time and embedded system environments. BBN is now directly investigating the ability to package QoS-related software into reusable units, called Qoskets, and to make these technologies available off-the-shelf to engineers trained in QoS (also known as qosketeers).

We are concerned about end-to-end QoS management which cannot be localized to a single or even a handful of components. Succeeding in this arena will enable systematic approaches towards the construction of software systems with complex QoS needs. We also are particularly concerned with providing dynamic, adaptive QoS management that will be needed for future systems’ complex (and changing) QoS requirements. We are developing a solution that, in the long term, will enable one to satisfy the necessary QoS requirements of software systems by decomposing and assembling Qoskets similar to the way systems are decomposed into functional components. Our solution will have immediate benefits

1. Add capabilities (B1) – Once there is support for dynamic, adaptive QoS management, it will be possible to build software systems that can alter their functional capabilities or resource usage in response to changes in the environments.
2. Reduce effort (B2) – Developers will not need to make as many individual decisions for ensuring QoS. The real pay-off will be the automated support for assembly rather than hand-customization of the individual parts.

We believe that any solution must have the following characteristics

1. Incorporate standardized component technology (C1) – By using standardized component technology, one minimizes the dependence on vendors of proprietary software. Existing analysis, design, or deployment tools for the standardized technology can be used “as is”.
2. Interoperate with non-component software (C2) – One cannot assume that all software elements of the final software system are packaged and deployed according to the agreed-upon component model. There may be substantial legacy software assets.
3. Integrate with system of systems (C3) – Any QoS management of resources, especially end-to-end QoS, takes place within a larger context. There will be no centralized QoS manager; rather, there will be federations of semi-autonomous managed systems.

1.1 Decomposition and Separate Responsibilities

Figure 1 presents our vision that combines the separation of roles (between application assembler and qosketeer) and the ability to decompose problems in each
space. The bold vertical line depicts when the action of satisfying QoS Requirements can be separated from the action of satisfying functional requirements. The dashed horizontal line divides monolithic and compositional development. On the left side of Fig. 1, the functional and QoS requirements are addressed simultaneously; below the dashed line, the software is decomposed into software components. On the right side of Fig. 1, there is a separate effort to ensure QoS requirements; below the dashed line we can decompose our QoS requirements whereas above the line, developers have no choice but to address all QoS requirements simultaneously.

The upper left box in Fig. 1 represents Case 1, the traditional monolithic software development where both Function and QoS requirements are addressed simultaneously by the software developers. In Case 2, the lower left box, the software is no longer monolithic and is decomposed into components; however individual component developers must be aware of QoS requirements and embed solutions to these QoS requirements within the components themselves. This leads to software that is not robust (in response to changes in executing environment) and cannot easily be modified to guarantee complex QoS requirements. In both these cases, the system is typically unable to support the dynamic adaptation of QoS requirements at run-time.

Starting from the premise that it is both meaningful and possible to separate QoS and functionality, there are two additional cases to consider. The upper-right box in Fig. 1 depicts Case 3 when there is a clear separation of roles between the functional developers and the qosketeers, but the underlying software code-base is still monolithic. The lower-right box in Fig. 1 depicts the ideal Case 4, where functionality and QoS are addressed separately and decomposition occurs independently, that is, whether to produce or assemble components (the small rectangles) or Qoskets (the small hourglasses).

There are two intermediate solutions before achieving the ideal Case 4. Case 3a matches BBN’s QuO technology for developing QoS modules that can be composed within a monolithic (i.e., non component-based) application to ensure end-to-end QoS requirements [7,8]. Such a system could be developed in an object-oriented language but the system remains monolithic without indepen-
dently deployable components. Case 3b provides a single “point of control” for managing the desired QoS requirements (i.e., the code to support QoS is monolithic) within a component-based application. This latter case is undesirable because it becomes increasingly complex and impossible to reuse QoS support between multiple applications; it also does not scale.

1.2 Motivating Problems

We motivate our work with examples that show the local and global characteristics of QoS management.

Local Example. Assume there is an Unmanned Aircraft Vehicle (UAV) that delivers images to a processing ground station. While the flight software for the UAV is carefully engineered to operate within tightly constrained restrictions (i.e., scheduling, resource usage) for the duration of its operation, more flexibility is allowed for the mission software. Specifically, the designers can identify numerous resource tradeoffs, such as power consumption, on-board CPU utilization, and network bandwidth requirements; there are also application tradeoffs, such as the amount of processing requested, the type of processing requested, and the nature of the data being processed. Instead of fixing these tradeoffs in advance, the UAV could operate differently based upon the current mission mode (an application-level concept) and available resources; for example, during navigation the highest image quality would be distributed but during an engagement a lower image quality might be acceptable to increase on-board sensor processing.

Global Example. We also address QoS management at a global “system of systems” level. Figure 2 shows how the in-flight UAV processing described above is simply part of a larger system composed of Command & Control (C2) ground stations, a Command Activity and Operations Center (CAOC), other remote UAVs, and even some simulated UAVs and C2 stations. The boxes in Fig. 2
represent computers with standard PC operating systems or proprietary UAV hardware that are networked together using Ethernet, radio, or satellite communication links. In this picture, there are three communication paths that must cooperate (and compete) for shared resources on each computer, such as CPU utilization, network bandwidth, or disk usage. We need to develop QoS management solutions to support the (changing) needs of applications over this space, for example, as new applications are added or as mission priorities change.

2 Existing QoS Management Approaches

There are a variety of approaches towards enabling QoS management within component-based applications. We briefly summarize these in this section.

Static Analysis/Prediction of QoS Requirements: Given an assembly of software components, together with relevant meta data, some researchers propose to analyze or predict statically that a given component assembly will meet its QoS requirements. De Jonge et al. present a technique for predicting the resource consumption for a component (for example, memory usage) by analyzing scenarios of resource usage [9]. Hissam et al. use rate monotonic analysis (RMA) to predict the latency of component execution at assembly time [12]. Such static approaches nicely complement the dynamic, adaptive enforcement of QoS requirements.

Run-Time Enforcement of QoS Policies: Rather than perform complex up-front analysis, it is possible to monitor the run-time behavior of a component assembly to determine if QoS policies have been met. Vecellio and Thomas present an approach (similar to QuO described in Sect. 2.1) that augments the component middleware (in their case, Enterprise JavaBeans) to ensure compliance with separately encoded policies [10,11]. This effort is the most closely related to ours; the difference lies in QuO’s contract definition. Ciuhandu and Murphy have designed a reflective load-balancing service for component-based middleware for ensuring QoS requirements [13].

Standards-Based QoS Middleware Extensions and QoS Frameworks: The power of Middleware comes from its ability to simplify applications by providing complex services, such as transactions, persistence, and security. Some believe Middleware needs to be extended to offer QoS services as well. While this is a daunting task, given the complexity of QoS management, there are some successes to date. QoS Enabled Distributed Objects (Qedo) implements QoS extensions to the CORBA Component Model (http://qedo.berlios.de); they are interested in support for stream-based communication and negotiation of QoS contracts between components at design time. There are real-time CORBA systems (such as TAO, http://www.cs.wustl.edu/~schmidt/TAO.html) that intend to satisfy performance and deadlines requested by the executing components. Our focus on dynamic, adaptive QoS management means we will always be
seeking solutions that go beyond what can be assured at assembly time. Finally, some researchers propose QoS frameworks within which all QoS requirements are specified and then supported [14]. We feel such approaches will ultimately prove too costly to implement and certify.

2.1 Quality Objects (QuO)

QuO is a framework for enabling distributed applications to adapt in response to a changing environment [7]. In a traditional distributed application, a client makes a method call on a remote object through its functional interface. The call is processed by middleware on the client’s host, delivered over the network to the middleware on a remote host, and processed by the remote object. As shown in Fig. 3, a QuO application inserts additional steps to this process. The QuO runtime monitors the state of QoS before each remote invocation through the use of System Condition objects (or SysConds) that provide a standard way to measure heterogeneous resources, such as CPU utilization or bandwidth usage. Delegate objects intercept in-band communication over the middleware and the local contract decides the appropriate behavior to apply. The contract is defined by a set of nested regions that describe the possible QoS states in the system. Each region is guarded by a predicate over SysCond values. Based upon the current QoS state, the contract could (1) specify additional processing to perform; or (2) allow the method call to proceed as is; or (3) redirect the invocation to a different method; or (4) invoke a callback on the application to alter its execution.

To summarize, QuO involves numerous additional entities:
– delegate objects intercept in-band communication between objects
– system condition objects (SysConds) extract information about the Network (such as bandwidth utilization), the Middleware implementation (such as access patterns or other reflective information), or the Client and Server objects carrying out the communication

![Fig. 3. Enabling QoS with QuO.](image-url)
– contracts perform the adaptive behavior by monitoring the values of SysConds
– callbacks enable contracts to contact elements of the application layer

These entities are deployed throughout the application, making it a challenge for packaging or reuse. QuO provides an abstraction of QoS contracts to the application assembler so most of these entities are inserted “under the hood” of the application which makes it a challenge to simultaneously deploy the application and the QoS mechanisms. Finally, it becomes a challenge to manage sets of multiple contracts. There are many characteristics of the QuO technology that cannot be described here for space reasons; full details are found in [7,8] or at http://www.dist-systems.bbn.com. Preliminary results on the use of Qoskets and QuO can be found in [7].

3 Proposed Approach

Figures 4a and 4b show a simplified view of the relevant elements of the problem space. Figure 4a depicts an assembly of functional components using a standards-based component technology such as CORBA Component Model (CCM). The Sender component sends image data generated by Process to a Distributor component on a server which, in turn, distributes the image to multiple remote Receiver components.

Software components are assembled together, communicating with each other through explicit and published interfaces. In some cases, the communication is a synchronous method invocation, in others it is the delivery of an asynchronous event. Each component executes within a container (shown as a gray box in Fig. 4) that provides the run-time environment for the component; there may be multiple components co-existing in the same container. Figure 4b shows the middleware and network layers that enable the execution of the distributed real-time and embedded system (as sketched by the directional lines). The containers in Fig. 4a are part of the middleware layer while other elements of the middleware are offered as services to the components in the application layer.

![Fig. 4. Two Perspectives.](image)
To ensure that end-to-end QoS requirements of the component assembly are met, there are many basic questions that must be addressed; among them are the following:

- Who should take the lead in QoS assurances for a software system?
- Where in Fig. 4b should QoS-enabling technologies be placed?
- What QoS artifacts need to be created?
- Where in Fig. 4a should QoS artifacts (i.e., qoskets) be placed?
- What changes are required for the software component lifecycle to address QoS?
- What happens when Qoskets are composed together?

In this paper we present our initial directions towards answering these questions, and conclude with specific milestones that we envision for future work.

3.1 Initial Directions

To enable the rapid integration of dynamic, adaptive QoS management with existing component technologies, we augment standards-based component middleware technologies with Qoskets. The application assembler is responsible for connecting components together to satisfy the functional requirements of the software system; this effort is supported by tools provided by the standards-based middleware and the resulting component assembly is described by an assembly file. We assume the application is already decomposed into functional components and the qosketeer must only apply Qoskets to ensure dynamic, adaptive end-to-end QoS management.

A Qosket is a packaged unit of reusable QoS-related software. A Qosket is each of the following, simultaneously:

- A collection of cross-cutting implementations – a set of QoS specifications (and supporting implementations) that are woven throughout a distributed application and its constituent components to monitor and control QoS and systemic adaptation.
- A packaging of behavior and policy – an encapsulation of adaptive QoS behavior and a policy for using that behavior (in the form of contracts and other artifacts).
- A unit of behavior reuse – an element supporting a single property (i.e., performance, dependability, or security).

This initial definition of a Qosket consolidates the various QuO technologies into bundled reusable behaviors. This consists of the following artifacts: QoS Contracts, System Condition Objects (SysConds), Callback Objects, Delegate Templates and Delegate Instances, Glue Code and other helper methods.

Each Qosket has as an element a qosket component that conforms to the component model defined by a standards-based middleware and provides access to the QoS-related technology supported by the Qosket. The qosket component is deployed in exactly the same manner as functional components. We envision
the need for a qosketeer tool called QoSTool that processes an existing assembly file (for example, as shown in Fig. 4a) to generate a new assembly file that integrates the qosket components with the functional components (for example, as shown in Fig. 5). The use of QoSTool occurs when the distributed real-time and embedded software system is assembled and deployed; when the software system executes, the QoS technology specified by the Qoskets enforces and ensures the required dynamic, adaptive QoS management at run-time. The long term vision is to enable application assemblers to decompose and assemble software systems from functional components and to simultaneously satisfy the necessary QoS requirements by decomposing and assembling Qoskets.

Who Should Take the Lead in QoS Assurances for a Software System? The person whose role is qosketeer should be the QoS representative for the software system. We expect that this is already a role that the application assembler plays today, although in an informal capacity. There is no need to put together a new QoS team; far better to train application assemblers in key skills (and technologies) needed to ensure QoS through components. Note that many organizations already have sophisticated development processes in place that address their existing QoS needs. The qosketeer may have to create programming standards or other guidelines to train component developers for certain domain-specific QoS capabilities; in general, however, we expect the individual component designers will be unaffected by the need to ensure QoS.

Where Should QoS-Enabling Technologies Be Placed? Since containers form such a prominent role in standards-based component middleware, our initial idea was to use off-the-shelf containers “as is” and encapsulate all QoS-related capabilities in the qosket components; we still believe this is the proper direction to head. An alternative view is to extend the containers to embed QoS knowledge within. The use of extended containers will be an attractive alternative for those attempting to optimize the component middleware to operate on resource constrained systems, as is common with distributed and real-time embedded systems; for now, we are avoiding optimization concerns in our discussion. We also believe this latter approach is more costly to design and implement.

CCM components (both functional components and qosket components) are deployed according to assembly files that describe the connections between components. To enable the integration of qosket components with functional components there needs to be a tool that (1) generates appropriate wrappers to intercept events going into (sinks) or out of (sources) a component as well as method invocations into a component (facets) or out of a component (receptacles); (2) creates assembly files to wire the generated wrappers together with the existing functional components. BBN will design this tool called QoSTool from the experience gained in manually carrying out these tasks. QoSTool will directly support qosketeers.

What QoS Artifacts Need to Be Created? QoS contracts form the basis for specifying QoS behavior, as has already been documented and demonstrated
using BBN’s QuO technology. The various sub-elements that make up a Qosket all have great reuse potential, which means that qosketeers developing Qoskets will not be forced to design and create Qoskets from scratch. Since the qosket component conforms to standards-based component middleware, there will need to be tools to help qosketeers rapidly create these if the Qoskets are to be applied to different component middleware technologies. At this time, we will craft the qosket components by hand. The QoS contract for a Qosket will be the primary way in which the QoS behavior of the Qosket is defined and made visible. The QuO technology currently provides a proprietary process for ensuring the Qosket sub-elements are distributed throughout the run-time of the underlying middleware [7]. BBN is currently upgrading QuO to support CCM which will enable the Qosket sub-elements to be deployed according to a CCM standard.

To optimize within resource-constrained environments, it is possible to reduce the operational footprint of components by more tightly integrating the qosket components with the functional components. While we do not pursue this direction further in this document, the option could be added to QoSTool.

Where Should QoS Artifacts (i.e., Qosket Components) Be Placed? The distinction between Qosket and qosket component needs to be made clear. A Qosket is composed of numerous elements while the qosket component provides a CCM-conforming element that supports the QoS behavior as encapsulated by the qosket. When a qosketeer applies a qosket component to an assembly, there are many sub-elements that must be distributed throughout the system. The insertion of the qosket component occurs at assembly and deployment time and so the distribution of sub-elements (such as system condition objects for monitoring, callback objects for contacting the application, and delegate wrappers for intercepting method invocations) also occurs at assembly and deployment time. These sub-elements will all be present at run-time (located throughout the distributed system) as already occurs with BBN’s QuO.

We envision there will be generic Qoskets that are made “domain-specific” for use within a software system or are contextualized for use within a specific product. This approach also prevents Qoskets from having to be constructed from scratch and enables families of Qoskets to be built around core concepts, such as network bandwidth or CPU management.

What Changes Are Required for the Software Component Lifecycle to Address QoS? Each component is packaged with meta data that describes relevant information about the component, such as the vendor, its footprint, or system requirements. The set of meta data is standardized according to the component model. We will extend this meta data to include attributes that contain primitive QoS information. For example, the dynamic memory requirements for a component could be known and declared in the meta data (i.e., requires 4K allocated bytes). As we design and package existing QoS-related software as Qoskets we will define standard QoS-related meta data attributes that will enable the QoS management of the components. These “styles of use” will be encoded within QoSTool and will enable downstream design tools to include this
information when analyzing component assemblies. Such meta data information will be composed of range values (i.e., requires between 200-475 ms to execute), exact information (i.e., spawns one additional thread), or parameterized values (i.e., requires 32N bytes of memory where N is the number of times function “open” is called).

We also expect that the ability to properly manage the QoS expectations for some components will demand that the component expose functional interfaces (also called knobs). In the same way that we will develop stylized attributes in the meta data, we will define standardized interfaces that prove useful to QoS management. We expect that over time (and certainly with the help of organizational memory) key interfaces could be defined, standardized by the qosketeer, and used by component developers. These interfaces would be “owned” by the qosketeer but the implementations would be part of each component, and thus be owned by the component designers. BBN will make its stylized attributes and knobs available to accelerate the integration of third party components and QoS-related technologies.

In the context of the UAV scenario, imagine a component that makes image information from a camera available as a JPEG file. The JPEG standard allows a variety of control over the size and quality of the image (i.e., whether to use standard or progressive encoding, the scale of compression). If the component exposed these application-specific options through a functional interface (i.e., a knob) then the QoS-management of the component could alter these settings.

Often generic components are produced that must be customized for a particular platform to be used. During customization, the meta data information associated with the component could be further refined based upon exact statistical information known for the platform. Turning to the UAV scenario again, it may be a firm requirement that the image quality never fall below a certain threshold; this value could be encoded in the meta information for each component and be used during QoS management.

Some QoS issues can only be decided when the functional assembly of components has been assembled. Using either sophisticated analysis techniques, simulation, or actual testing, the qosketeer assigns relevant QoS meta data values to the components in the assembly. The qosketeer also designs the relevant QoS contracts and generates qoskets to drop into the assembly.

**What Happens when Qoskets Are Composed Together?** To achieve the premise of Qosket decomposition, QoSTool must have sophisticated logic to detect when the application of multiple Qoskets requires special handling. The interaction between Qoskets is similar to the feature interaction problem that has long been studied in telecommunications software and software in general. We will not provide a solution for the general case of Qosket composition; rather, we will identify families of Qoskets (i.e., those that manage CPU, those that manage network bandwidth) and provide algorithms for valid compositions (or heuristics if the algorithms prove unwieldy). To provide a specific example of why Qosket composition matters, consider a compressor Qosket that compresses data at the expense of extra processing and a manager Qosket that manages the
application’s use of the CPU. By applying the compressor Qosket, the cycles available to the manager Qosket are reduced. Other compositional issues arise when one considers the hierarchical composition of Qoskets; these issues will be addressed in the future.

4 Current Status

Given the Sender/Distributor/Receiver example earlier, consider how one could add the following capabilities to the system:

1. Higher priority receivers must receive data before lower priority receivers
2. Ensure that all receivers receive an image within a fixed time window; the image can be scaled to accommodate available resources
3. Enable receivers to specify a minimum acceptable image quality
4. Enable sender to perform opportunistic processing when resources allow

One could easily imagine adding any number of similar characteristics to an application. Using our Qosket solution, Fig. 5 shows how we can introduce various qosket components (shown as hourglasses) into the application assembly. We omit many details for brevity; the assembly pictured here is currently being designed and prototyped.

The Sender component generates reconnaissance images; based upon available resources, additional “features” can be added to the image, such as GPS location or automatic target recognition (shown with an outline). The Distributor component delivers this image to numerous Receivers, perhaps scaling the image to suit the available resources of the client receiver. The original Sender, Distributor, and Receiver components exist without change, and all desired QoS behaviors are encapsulated within the qosket components.
5 Future Direction

This paper presents initial results in our efforts to componentize units for QoS management. Moving forward, we know there are many questions yet to be answered, each of which will lead to fruitful research discussions.

5.1 Qosket vs. Qosket Component

There is a distinction between Qoskets, an idealized reusable unit of QoS behavior, and qosket components, the executable units enabling the QoS management that conform to a specific component technology. To date, we have developed qosket components for Mico CCM and are currently developing qosket components for CIAO CCM and Boeing’s BoldStroke [1]. We expect to be able to design an abstraction (much like language bindings in CORBA) to enable these qosket components to be generated from executable specifications of the different component technologies. We can also build on the templates already developed by QuO [7].

5.2 Coordinating Multiple Qosket Components

While decomposing QoS requirements into Qoskets is essential to manage complexity, we have yet to provide a comprehensive solution for the coordinating the efforts of multiple Qoskets. At one level we need to ensure that the information gathered by SysCond objects are shared and used by the Qoskets that need them. Currently, QuO contracts interact with each other in an ad hoc manner; we desire a standardized approach to qosket interaction.

5.3 Composing Multiple QoS Behaviors

Once Qoskets are interacting with each other, we need to ensure that one can compose QoS behaviors by describing the interaction patterns between qoskets. We will not attempt to ensure arbitrary composition of Qoskets; rather, we are targeting specific QoS categories (CPU utilization, network bandwidth) and specific policies (i.e., reservations, prioritizations). Within this focused domain we should be able to accurately model and analyze the compositions to determine whether the QoS requirements are met.

5.4 Investigate QoS Mechanisms

Qoskets are ideal for providing standardized interfaces to complex QoS services. To date, BBN has investigated numerous services, such as CPU prioritization, scheduling, multi-level resource management, fault tolerance, and security. Rather than waiting for the underlying middleware to support the desired characteristics, we are pursuing numerous investigations towards encapsulating the desired behaviors within Qoskets.
Acknowledgments

We would like to thank Gary Duzan and Praveen Sharma of BBN for their help in integrating BBN QuO technology with Mico CCM. This work is funded by a DARPA contract from the Programmable Composition of Embedded Software Program, http://dtsn.darpa.mil/ixo/programdetail.asp?progid=37. This paper has been cleared for public release. George Heineman is on sabbatical from Worcester Polytechnic Institute.

References


Abstract. Scientific computing on massively parallel computers presents unique challenges to component-based software engineering (CBSE). While CBSE is at least as enabling for scientific computing as it is for other arenas, the requirements are different. We briefly discuss how these requirements shape the Common Component Architecture, and we describe some recent research on quality-of-service issues to address the computational performance and accuracy of scientific simulations.

1 Introduction

Massively parallel scientific computing, like its counterparts in the commercial sector, must contend with perpetually increasing software complexity. In scientific domains, software complexity arises from the desire to simulate intrinsically complicated natural phenomena with increasing fidelity. Current high-performance parallel simulations of this nature include climate modeling, nanotechnology, magnetohydrodynamics, quantum chromodynamics, computational biology, astronomy, and chemistry, and more recently, multiscale and multiphysics hybrids of two or more of these.
The motivation for component-based software engineering (CBSE) in scientific simulations is largely the same as that for other pursuits: components are a uniform way of compartmentalizing complexity in building blocks for applications. In this paper we present a brief overview of the requirements of CBSE for high-performance scientific computing, and we present the Common Component Architecture (CCA) approach, on which the computational quality-of-service (CQoS) work is based.

Component-based environments offer a degree of flexibility over traditional monolithic scientific applications that opens new possibilities for improving performance, numerical accuracy, and other characteristics. Not only can applications be assembled from components selected to provide the best performance, but they can also be changed dynamically during execution to optimize desirable characteristics. The quality-of-service (QoS) aspects of scientific component software that we consider in this paper differ in important ways from more common component-based sequential and distributed applications. Although performance is a shared general concern, high sequential and parallel efficiency and scalable performance is a more significant requirement in scientific component design and deployment. The factors that affect performance are closely tied to the component’s parallel implementation, its management of memory, the algorithms executed, and other operational characteristics. In contrast, performance quality of service in nonscientific component software focuses more on system-related performance effects, such as CPU or network loads. The composition of scientific components also affects their individual performance behavior, suggesting the need for QoS metrics that measure cross-component performance.

Scientific component software is also concerned with functional qualities, such as the level of accuracy achieved for a particular algorithm. When components can operate under various functional modes while employing the same external interface and can switch between modes during execution, different service requirements can arise. Moreover, the interaction of the functional qualities with the performance qualities of scientific components makes dynamic service mechanisms distinctly important. For example, the selection of an algorithm for a given problem must take into account a possible tradeoff between speed and reliability. When these component-specific QoS concerns are considered globally in the context of the composite component application, opportunities to enhance the computation arise.

We refer to this concept of the automatic selection and configuration of components to suit a particular computational purpose as computational quality of service (CQoS). CQoS is a natural extension of the capabilities of the component environment. The name refers to the importance of the computational aspects – both functional and nonfunctional – of scientific components in how they are developed and used. CQoS embodies the familiar concept of quality of service in networking and the ability to specify and manage characteristics of the application in a way that adapts to the changing (computational) environment. We discuss techniques to support CQoS capabilities from the viewpoint of enhancing the computational service being offered.

In this paper, we first overview the background and requirements for CBSE in scientific computation. Next, we briefly describe the CCA. We then discuss the concept of computational quality of service as it applies to components for high-performance scientific applications, and we describe an initial implementation of a CQoS environment.
that is being integrated with the CCA technology. We conclude with prospects for future work.

2 CCA Overview

Apart from a social reticence to accept solutions not developed within their own scientific communities, researchers are particularly concerned about the performance implications of a relatively new approach such as CBSE. Timescales for message-passing operations on modern supercomputers are measured in microseconds, and memory latencies in nanoseconds. The conventional rule of thumb is that environments that incur a performance cost in excess of 10 percent will be rejected outright by computational scientists. In addition, scientists are concerned about the impact of applying new techniques to extensive bases of existing code, often measured in hundreds of thousands of lines developed over a decade or more by small groups of researchers; extensive rewriting of code is expensive and rarely justifiable scientifically.

While there have been a number of experiments with commodity component models in a high-performance scientific context [1,2], so far they have not had noticeable acceptance in the scientific community. Unfortunately, various aspects of commercial component models tend to limit their direct applicability in high-performance scientific computing. Most have been designed primarily with distributed computing in mind, and many have higher overheads than desirable, even where multiple components within the same address space are supported. Support for parallel computing is also a crucial consideration. The effort required to adapt existing code to many commercial component models is often high, and some impose constraints with respect to languages and operating systems. For example, in high-end computational science, Java is still widely viewed as not providing sufficient performance, making an approach like Enterprise JavaBeans unattractive; and almost no supercomputers run Windows operating systems, limiting the applicability of COM.

The scientific high-performance computing (HPC) community has made some tentative steps toward componentlike models that are usually limited to a specific domain, for example Cactus [3], ESMF [4], and PALM/Prism [5]. While successful in their domains, these approaches do not support cross-disciplinary software reuse and interoperability.

In response, the Common Component Architecture Forum [6] was launched in 1998 as a grassroots initiative to bring the benefits of component-based software engineering to high-performance scientific computing. The CCA effort focuses first and foremost on developing a deeper understanding of the most effective use of CBSE in this area and is proceeding initially by developing an independent component model tailored to the needs of HPC.

Space constraints require that we limit our presentation of the CCA here; however, further details are available at [6], and a comprehensive overview will be published soon [7]. The specification of the Common Component Architecture defines the rights, responsibilities, and relationships among the various elements of the model. Briefly, components are units of encapsulation that can be composed to form applications; ports are the entry points to a component and represent interfaces through which components interact – provides ports are interfaces that a component implements, and uses ports are
interfaces that a component uses; and the framework provides some standard services, including instantiation of components, as well as uses and provides port connections.

The CCA employs a minimalist design philosophy to simplify the task of incorporating existing HPC software into the CCA environment. This approach is critical for acceptance in scientific computing. CCA-compliant components are required to implement just one method as part of the gov.cca.Component class: the component’s setServices() method is called by the framework when the component is instantiated, and it is the primary means by which the component registers with the framework the ports it expects to provide and use. Uses ports and provides ports may be registered at any time, and with the BuilderService framework service it is possible programmatically to instantiate/destroy components and make/break port connections. This approach allows application assemblies to be dynamic, under program control, thereby permitting the computational quality-of-service work described in Section 3. Furthermore, this approach ensures a minimal overhead (approximately the cost of a virtual function call) for component interactions [8].

Most parallel scientific simulations use a single-program / multiple-data (SPMD) paradigm, in which an identical program runs on every process/processor, using the Message Passing Interface (MPI) [9] or an equivalent message-passing mechanism over an interconnection fabric. This approach sometimes is relaxed to the multiple-program/multiple-data (MPMD) pattern, which includes multiple communicating instances of SPMD programs. Analogously, the CCA’s model is that of many “same-process” component assemblies instantiated as a parallel cohort across all participating processes (see Figure 1). In direct contrast with a distributed object model of components (e.g., CORBA), component connections occur within a single process for maximum performance. Interprocess communication, usually MPI, is left to the components themselves without CCA interference. Both single-component/multiple-data and multiple-component/multiple data paradigms are supported, analogous to SPMD and MPMD programs.

3 Computational Quality of Service

Quality of service is often associated with ways of implementing application priority or bandwidth reservation in networking. Here computational quality of service (CQoS) refers to the automatic selection and configuration of components to suit a particular computational purpose. While CBSE helps to partition complexity in parallel simulations, it also presents its own problems. For example, if data is distributed across all participating processors (Fig. 1), each component must deal with the distributed data as it is presented; it is almost never efficient to redecompose the problem optimally.
for each component. If the components are thorough black boxes, then there would be no mechanism to optimize this decomposition over all components interacting with it. However, if metadata is provided either as part of the static information associated with the component repository, or as dynamic information computed in real time, a “resource-controller” component could configure its peer components by taking the global situation into consideration (see Fig. 2). This special-purpose component interprets mechanistic, performance, or dependency metadata, provided by its peer components, to make an optimal solution within the context of an entire application or a local container component. For more information on CCA containers, see [10].

This approach not only solves CBSE problems but presents new opportunities, primarily that of being able to dynamically replace poorly performing components. Component concepts help to manage complexity by providing standard building blocks; these concepts also enable a degree of automation at a high level. Here we will describe how CBSE in scientific computing provides opportunities to automate scientific simulations for better performance and accuracy.

CQoS metadata may be used to compose or dynamically adapt an application. A detailed design of an infrastructure for managing CQoS-based component application execution was proposed in [11]. The CCA enables the key technology on which CQoS depends, including component behavior metadata and component proxies for performance modeling or dynamic substitution. By associating CQoS metadata with a component’s uses and provides ports, one can effectively express that component’s CQoS requirements and capabilities.

CQoS employs global information about a simulation’s composition and its environment, so that sound choices for component implementations and parameters can be made. Building a comprehensive CQoS infrastructure, which spans the algorithms and parallel decomposed data common to scientific simulations, is an enormous task but, given the need to automate the cooperation of algorithmically disparate components, a necessary one. The research reported in the rest of this section is a first step toward this aim and thus first addresses problems that interest the scientific simulation community.

**Performance Measurement and Monitoring.** The factors that affect component performance are many and component dependent. To evaluate component CQoS, one must have a performance system capable of measuring and reporting metrics of interest. We have developed a performance monitoring capability for CCA that uses the TAU parallel performance system [12] to collect performance data for assessing performance metrics for a component, both to understand the performance space relative to the metrics and to observe the metrics during execution. After performance data have been accumulated, performance models for single components or entire applications can be constructed.
An accurate performance model of the entire application can enable the automated optimization of the component assembly process.

Automated Application Assembly. CCA scientific simulation codes are assemblies of components created at runtime. If multiple implementations of a component exist (i.e., they can be transparently replaced by each other), it becomes possible to construct an “optimal” CCA code by choosing the “best” implementation of each component, with added consideration for the overhead of any potentially necessary data transformations. This construction requires the specification of quality attributes with which to discriminate among component implementations. In this discussion, we will focus on execution time as the discriminant.

Performance data can be measured and recorded transparently via the proxy-based system described in [13]. Component interface invocations are recorded, resulting in a call graph for the application. The net result of a fully instrumented run is the creation of data files containing performance parameters and execution times for every invocation of an instrumented component as well as a call graph with nodes representing components, weighted by the component’s execution time.

Performance models are created through regression analysis of the data collected by this infrastructure. The call-graph is also processed to expose the cores, or components that are significant from the perspective of execution time. This processing is done by traversing the call tree and pruning branches whose execution time is an order of magnitude less than the inclusive time of the nodes where they are rooted. Since component performance models can be constructed from performance data collected from unrelated runs or from unit tests, the models consequently scale, at worst, as the total number of component implementations. The final composite model for a component assembly reduces to a summation over the performance models of each of the components in the cores. At any point before or during the simulation, the performance models of each of the component implementations are evaluated for the problem’s size to obtain the execution times of any component assembly prior to choosing the optimal set. Once an optimal set of components have been identified, the performance modeling and optimization component, named Mastermind, modifies the existing component assembly through the BuilderService interface introduced in Section 2.

Adaptive Polyalgorithmic Solvers. While application assembly is typically done once before a scientific simulation starts, often the same set of component implementations does not satisfy CQoS requirements throughout the application’s entire execution. Many fundamental problems in scientific computing tend to have several competing solution methods, which differ in quality attributes, such as computational cost, reliability, and stability. For example, the solution of large-scale, nonlinear PDE-based simulations often depends on the performance of sparse linear solvers. Many different methods and implementations exist, and it is possible to view each method as reflecting a certain tradeoff among several metrics of performance and reliability. Even with a limited set of metrics, it is often neither possible nor practical to predict what the “best” algorithm choice for a given choice may be. We are in the initial stages of investigating dynamic, CQoS-enhanced adaptive multimethod linear solvers, which are used in the context of solving a nonlinear PDE via a pseudo-transient Newton-Krylov method. Depending on the problem, the linear systems solved in the course of the nonlinear solution can
have different numerical properties; thus, a single linear solution method may not be appropriate for the entire simulation. As explained in detail in [14], the adaptive scheme uses a different linear solver during each of the three phases of the pseudo-transient Newton-Krylov algorithm, leading to increased robustness and potentially better overall performance.

4 Conclusion

CBSE provides a mechanism for managing software complexity and enabling hundreds of scientists to participate in the development of large-scale simulation software, something currently lacking in scientific computing. The CCA model of component-based development offers a standard approach to component and application construction that is specific to parallel scientific computing but also generally applicable to many domains within computational science. The CCA has already been proven successful in several scientific domains, including climate modeling [15], combustion [16], and computational chemistry [17].

The emergence of these component-based scientific codes has motivated the development of an abstract infrastructure for describing computational quality-of-service (CQoS) requirements and capabilities. CQoS requires an environment that contains services for monitoring and managing performance data, analyzing static and dynamic performance information, optimizing application assembly, and adaptively substituting components. A CQoS environment should be developed in a manner consistent with a CBSE methodology to maintain coherence with the engineering of scientific component applications. The work described here demonstrates the utility of such an environment and lays the groundwork for it. As parallel computing hardware becomes more mainstream, our hope is to see a corresponding increase in commodity simulation components that can be easily used to build parallel scientific applications.

Acknowledgments

This work has been supported in part by the U.S. Department of Energy’s Scientific Discovery through Advanced Computing initiative, through the Center for Component Technology for Terascale Simulation Software. We thank Nick Trebon (University of Oregon) for his help in implementing the performance measurement infrastructure as well as the call graph pruning algorithms. We also thank Sanjukta Bhowmick (Penn State University) for her work in implementing the adaptive polyalgorithmic solvers and Gail Pieper (ANL) for copyediting a draft of this paper.

References

A Framework for Reliability Assessment of Software Components

Rakesh Shukla, Paul Strooper, and David Carrington

School of Information Technology and Electrical Engineering,
The University of Queensland, St. Lucia 4072, Australia
{shukla,pstroop,davec}@itee.uq.edu.au

Abstract. This paper proposes a conceptual framework for the reliability assessment of software components that incorporates test case execution and output evaluation. Determining an operational profile and test output evaluation are two difficult and important problems that must be addressed in such a framework. Determining an operational profile is difficult, because it requires anticipating the future use of the component. An expected result is needed for each test case to evaluate the test result and a test oracle is used to generate these expected results. The framework combines statistical testing and test oracles implemented as self-checking versions of the implementations. The framework is illustrated using two examples that were chosen to identify the issues that must be addressed to provide tool support for the framework.

1 Introduction

With an escalating sophistication of computer applications, the demands for complex and large-scale software are increasing. There is a growing trend to build complex software by integrating software components. As a result, concerns about the reliability of software components are becoming more important.

The reliability of a software component is a probability prediction for failure-free execution of the component based on its usage requirements. Hardware components are different from software components because they wear out. While the reliability of hardware components can often be characterised by exponential decay over time, the reliability of a particular version of a software component does not change over time. Determining an operational profile and test output evaluation are two difficult and important problems that must be addressed in software reliability engineering [10, 11].

1. Operational profile: An operational profile is a set of input events and their associated probabilities of occurrence expected in actual operation. Determining an accurate operational profile for software is difficult in general and it is very difficult for many software components, because it requires anticipating the future use of the component.

2. Output evaluation (test oracle): An expected result is needed for each test case to check the test output (or behaviour). The mechanism used to check these expected results is called a test oracle. A test oracle is an essential part of reliability assessment, because for high reliability a large number of test cases are required and the behaviour must be checked for every test case.
While a number of proposals for evaluating component reliability have been made, including determining and specifying operational profiles, the execution of test cases and the determination and checking of expected outputs is often ignored in these proposals. In this paper, we propose a conceptual framework for the reliability assessment of software components that incorporates both test case execution and output evaluation. In the framework, it is the component user’s responsibility to define the operational profile of the component and it is the developer’s responsibility to supply the component. The responsibility for developing a test oracle belongs to the developer, or to the user when the developer cannot be expected to deliver an oracle, such as for COTS (Commercial Off The Shelf) components. Two examples illustrate the applicability of the framework and are used to identify the practical problems and important issues that must be addressed to provide tool support for the framework.

The paper is organised as follows. Section 2 presents related work. Section 3 introduces our framework for the reliability assessment of software components. Section 4 describes our experience in applying the framework to two examples. Section 5 presents our conclusions and future work.

2 Related Work

We focus on the reliability of the current version of a piece of software and assume that any faults found during testing will be removed and testing will be recommenced. The goal of reliability assessment [3] is to determine the failure probability with a predefined confidence. Ammann et al. [1, 3] devise stopping rules for the testing of software for reliability assessment.

McGregor et al. [9] propose the Component Reliability (CoRe) method that supports the empirical measurement of the reliability of a software component. With CoRe, various roles for the component are identified and an operational profile is determined for each role, from which a reliability test suite is created and executed. Specific support for test case execution or output evaluation is not discussed. With CoRe, the roles for the component are determined a priori and combined to determine component reliability. Our framework requires the component user to specify an operational profile and the reliability of the component will be computed separately for each different operational profile.

Determining an operational profile is difficult because it requires anticipating the future use of the software [10]. Woit [13] describes a technique for the specification of operational profiles, using small hypothetically generated operational profiles as examples. She presents a test case generation algorithm for these examples. Her work does not address test execution and evaluation. We build on her work by addressing these issues. Whittaker [12] proposes an operational profile method using Markov chains.

A test oracle is required to check the test outputs. Several methods for translating formal specifications/documentation to test oracles have been proposed. McDonald and Strooper [8] discuss passive oracles derived from a formal specification. We follow their approach in our framework, but the framework is general enough to allow other test oracles, including manually developed oracles without using a formal specification.
The important issue of how to determine component reliabilities is addressed by our framework. Bass et al. [2] raise the issue that a lack of independently certified components is an inhibitor to the success of software component technology. The framework addresses basic technical problems in third-party component certification. How component reliabilities can be composed to derive system reliability estimates has been addressed by others [6, 11] and is not discussed further in this paper.

3 Framework Overview

Fig. 1 shows an overview of the proposed framework for the reliability assessment of software components, which combines statistical testing and test oracles. The rectangles represent processes, the ovals represent outputs, the folded corner documents represent inputs and the cubes represent software components. Test case generation requires definition of an operational profile and the number of test cases to be executed. As explained below, the framework supports a variety of test oracles, including ones derived from formal specifications.

The stakeholders are the software component user and the software component developer. The framework requires the user to specify an operational profile and the number of test cases to be executed or the desired reliability for the component. The component developer supplies the component.

To determine the reliability of the component, its test outputs must be evaluated during testing. In the proposed framework, this is done through a test oracle that provides a self-checking implementation of the component, which can be implemented using inheritance [8] or delegation [5]. The oracle presents the same user interface as the component under test and is used in place of the component during test execution. As illustrated in Section 4.5, this approach is general enough to support test oracles generated from formal specifications and also manually developed oracles. The oracle can be written by the developer or by the user (or a representative). The latter will have to be the case when the developer does not supply an oracle, as is typically the case for COTS components.
The operational profile and oracle are both crucially important for the validity of the results of the framework. For example, if the operational profile does not accurately reflect the actual usage of the component, then the reliability results are meaningless for that use of the component. Similarly if the oracle does not detect an error (for example, by having the same error in the oracle as in the implementation) during testing, then the reliability estimates are going to be overly optimistic. Therefore component and test oracle development by different developers is recommended.

The framework samples from the specification of the operational profile to generate test cases, executes these test cases, and evaluates the test output. The test case generation uses the operational profile and the number of test cases specified by the user, or the number of test cases calculated from the desired reliability supplied by the user. During test case execution, the framework invokes the oracle that is implemented as a self-checking version of the implementation, which calls the implementation and then checks the behaviour. Based on the results of the test case execution, the framework calculates the reliability of the software component for the operational profile and number of test cases specified by the user, or confirms/denies that the component has the desired reliability specified by the user.

4 Examples

This section discusses the manual application of the framework to two case studies. The purpose of the case studies is to recognise the important issues that must be addressed to provide tool support for the framework. The first example is a simple stack component in which we use the hypothetically generated operational profile specification from [13], and a test oracle generated from a formal specification. In this case, we run enough test cases to confirm a desired reliability or report an error if any tests fail. The Object-Z [4] specification and implementation provide access to a bounded stack of integers. The user can add (push) an element, remove (pop) an element and query the top element.

An existing tree component that is part of the PGMGEN testing tool [7] is used as a second, non-trivial example. Determining an operational profile from actual usage data for the operations (particularly recursive operations) and generation/assignment of appropriate input values for the operations are difficult. We use an operational profile that is derived from actual usage data of the tree in an application, use a number of different oracles, to experiment with different oracle types, and calculate reliability from a given number of test cases.

4.1 Operational Profile Specification

A component fails when processing an input event in the current state generates incorrect output or state. The failure rate of a software component is the probability of encountering input events and states that will cause the component to fail. An operational profile is a description of the distribution of input events that is expected to occur in actual component operation. Woit [13] describes a method for specifying operational profiles for software modules for the stack example and we use the same operational profile.
We generate an operational profile for the tree component from traces generated from the actual use of the component. When we applied Whittaker’s method [12] to generate an operational profile from these traces, the Markov chain for the tree resulted in an unstructured and chaotic diagram. The problem is that in this case, future inputs to the component depend on more than just the last call issued. We therefore extended Whittaker’s method using a hierarchical state machine. The operational profile generation from the actual usage data proved much more difficult than originally anticipated. In particular, it proved difficult to capture the relationships between:

- output parameters of certain functions and input parameters to subsequent functions,
- multiple parameters of the same function, and
- return values of certain functions that control what happens subsequently.

These issues are ignored by operational profile research [10, 12, 13] but must be addressed for the framework to be generally applicable.

4.2 Test Case Generation

Following [13], we implemented an algorithm to generate test cases according to the stack operational profile. Woit [13] manually generates a test driver for each module from the operational profile. This was also done for the examples presented in this paper. The planned tool support for the framework will do this automatically from a generic specification of the component user interface and an operational profile.

The difficult issues of how to generate appropriate parameters for the operations of the component and how to establish relationships between them must be resolved to provide such tool support.

4.3 Reliability Estimation and Number of Tests

The reliability assessment presented in this paper aims to estimate the reliability of a software component with no known defects. As such, we anticipate no errors during testing and assume that all tests are executed without failures. Although reliability calculations are possible in the presence of failures, we expect that the component would be fixed each time a failure is reported.

The framework can be used in one of two ways: (1) to estimate the reliability for a given number of test cases or (2) to confirm that the component has a given reliability. In each case, the number of test cases required during testing is available (it is either specified by the user or can be calculated using reliability estimation) before generating and executing any test cases.

We follow Woit [13] for the reliability calculations. For the stack example, we set out to show with 99.9% confidence that its reliability is at least 0.999. We have to perform 6904 tests without failure to achieve this. For the tree component, we set out to calculate its reliability after successfully executing 1000 test cases. Moreover, we want to have 99% confidence in the reliability estimate. We can state with 99% confidence that the reliability of the tree component is at least 0.995 for the specified operational profile.
To assure that the distribution of the test cases generated from the operational profile is significant with respect to the operational profile, we use a statistical test of significance, goodness-of-fit test [13].

4.4 Test Case Execution and Output Checking

The planned tool support for the framework will automatically generate a test driver from a generic specification of the component user interface and an operational profile. When this test driver is executed, a test oracle is needed to check the results produced by the component under test.

We implement a test oracle as a self-checking implementation of the component under test. To gain familiarity with approaches to test case generation and output checking, we generated a passive Java oracle for the stack by hand from the Object-Z specification following the approach presented in [8].

For the tree component, we manually generated four different test oracles using the wrapper approach. The first three oracles are passive in that the oracle itself does not attempt to maintain the current state that the implementation should be in. Instead, the first two oracles relate the implementation state to an abstract state using an abstraction function. The third oracle gains access to the implementation state using the public interface of the component. Finally, the last oracle is an active oracle that does maintain its own state in parallel with the implementation state. Note that for the first two oracles, we assume that we have access to the state of the implementation. For the last two, we make no such assumptions.

The four oracles we implemented are:

1. A passive test oracle derived from an Object-Z specification following the approach in [8]. The oracle uses the abstract state from the Z toolkit and predicates from the specification.
2. A passive test oracle following the approach in [8] where the abstraction function relates the concrete implementation state to an abstract state that is modelled using classes from the Java JDK.
3. A passive test oracle that uses the component’s user interface to check the results. Clearly the amount of checking that can be done with such an oracle depends on how observable the internal state of the component is through its public interface. In the case of the tree component, only limited checking can be performed in this way.
4. An active test oracle that uses the state of a parallel implementation to generate the expected behaviour of the implementation. Such an approach to test oracle development involves implementing a second version of the component. Clearly this can be prohibitively expensive but since the oracle does not need to be efficient it may be substantially simpler than the original implementation.

The oracles implemented for the tree component show that the proposed approach is general enough to support a wide range of test oracles. We plan to investigate the feasibility and advantages of these different oracle types in future work.
5 Conclusion

We have proposed a framework for the reliability assessment of software components that incorporates test case execution and output evaluation. The framework requires the user of the component to define an operational profile for the component and appropriate reliability parameters. The framework needs an executable component from the developer. The framework expects an executable oracle from the developer or from the user. The framework then generates test cases, executes them, and evaluates the test results. If no failures are detected during the testing, it determines/confirms the reliability of the component as specified by the parameters defined by the user.

The application of the proposed framework to two case studies establishes practical viability and flexibility of the framework. The framework has been applied on a trivial and a more realistic component to experiment with issues in the framework. One example used a hypothetically generated operational profile and the other an operational profile generated from actual usage of the component. One example used a test oracle generated from a formal specification and the other a variety of test oracles. For one example, we estimated the reliability for a given number of test cases and for the other we confirmed a desired reliability of the component.

In this paper, the framework was applied on components implemented in Java. However, the conceptual framework can be applied to any software component technology, such as Enterprise JavaBeans or Microsoft’s COM+.

The main areas for future work are to provide tool support for the framework and apply it to an industrial case study. A general algorithm for the generation of test cases according to any operational profile is necessary. Determining an operational profile for the tree component has proven to be a difficult step. A systematic method for defining an operational profile for a component has therefore also been identified as further work. Finally, we plan to compare the effectiveness of different test oracle types for the reliability assessment of software components.

Acknowledgements

The authors are grateful for the valuable comments and help from Mark Bofinger, Alena Griffiths, Roger Duke and anonymous referees on an earlier version of this paper.

References


Performance Prediction for Component Compositions

Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer

Department of Mathematics and Computing Science, Technical University of Eindhoven,
Postbox 513, 5600 MB Eindhoven, The Netherlands
{e.m.eskenazi,a.v.fioukov,d.k.hammer}@tue.nl

Abstract. A stepwise approach is proposed to predict the performance of component compositions. The approach considers the major factors influencing the performance of component compositions in sequence: component operations, activities, and composition of activities. During each step, various models – analytical, statistical, simulation – can be constructed to specify the contribution of each relevant factor to the performance of the composition. The architects can flexibly choose which model they use at each step in order to trade prediction accuracy against prediction effort. The approach is illustrated with an example about the performance prediction for an Automobile Navigation System.

1 Introduction

Component-based software architecting is becoming a popular approach nowadays. Component-based solutions help to manage the complexity and diversity of products and to reduce lead-time and development costs. However, main advantages of this approach – reuse and multi-site development – often cause problems at component composition time, as the non-functional requirements are not satisfied even if the functional specifications are met. Therefore, architects want to estimate the quality attributes of the product at the early architecting phase. This estimation is difficult as these properties often emerge from interactions between components.

In our research, we concentrate on one of the most critical quality attributes, performance, which characterizes how well the system performs with respect to the timing requirements [19].

We aim at delivering software architects an appropriate method for predicting the performance of a component assembly. This method should allow the quick estimation of the performance of a component-based system during the architecting phase. It should support flexible selection of the abstraction level for behavior modeling to let the architects concentrate on performance relevant details only, and to trade the estimation effort against the estimation accuracy. Additionally, the method should employ well-known software engineering notations and should not require any additional skills from software architects.
2 Related Work

Prediction of the performance of an assembly of components has become a challenging research topic for different research groups. Paper [17] shows the importance of this problem, associated complications, and overview of current work in the field.

Most of the existing approaches to compositional performance reasoning are analytical ([1], [9], [16], [19], [20], and [21]). They are formal, and often based upon too strict assumptions about the systems under consideration. Modern complex software systems with hundreds of components do not satisfy these assumptions. Therefore, the analytical approaches often suffer from combinatorial explosion. Moreover, these approaches are usually considered as too complicated for industrial practice.

Performance prediction models based on statistical regression techniques ([2], [3], [3a], [8], and [14]) reflect the software behavior only in terms of formulas obtained by curve-fitting and hide architectural insight.

Some approaches (e.g., [3]) require the entire code of software system to be present and are not applicable at the early architecting phase. Simulation-based approaches ([10], [18]) usually imply the simulation of the entire software stack, with all details, which is quite time consuming. Moreover, the complex simulation models are as error-prone as the original software.

Another drawback of many contemporary approaches is the insufficient use of the knowledge about existing versions of the software, which is especially beneficial for software product families with huge amount of legacy.

3 Aspects of Composition

Let us introduce important notions that we will use to describe our approach for predicting performance of a component composition.

An activity\(^1\) is a unit of functionality. An activity is described by a directed graph whose nodes are component operations. The activity instantiates a number of activity instances, the units of concurrency. Activity instances are triggered by events with a certain arrival pattern. This pattern defines the instants at which activity instances are released. We consider the arrival patterns (e.g., periodic, a-periodic, burst arrivals) that are usual in the real-time literature [13].

Often, several activity instances need to be executed at the same time. These activities have to be allocated processors to execute upon; they can also contend for non-sharable resources (see [13]).

The number of processors and resources is usually limited. This means that several activity instances will contend for the resources and processors. To resolve these conflicts, a dedicated manager, the scheduler, is introduced. The scheduler implements a certain scheduling policy that determines the rules of access to shared resources. A detailed taxonomy of existing scheduling policies (e.g., priority-based, time-triggered, preemptive, etc.) is presented in [13].

\(^1\) In the literature, also the term “logical process” or simply “process” is often used.
4 Foundations of Composition Models

This section presents a hierarchical view on the modeling of component compositions.

The construction of the performance model of a component composition starts with the performance models for the component operations (see Fig. 1). These operations are described by black box models for predicting the processor and resource demand.

Activities call component operations to implement their functionality. Each activity is modeled by a control flow graph\(^2\). The activity models are used to estimate the processor demand of an activity instance. At this level of the hierarchy, the resource demand is considered to be independent from interactions with other activity instances.

Finally, on the top of the pyramid, it is necessary to build a model of activity composition that describes the concurrency and synchronization between different activity instances. This top-level model not only combines the resource and processor demands of all activities running concurrently, but also accounts for effects of scheduling such as blocking and preemption times.

Please notice that this hierarchical approach will only work, if there are no backward dependencies between layers. For instance, the processor or resource demand of component operations must not depend on the control flow graphs of activities and composition of activities.

The complexity of the resulting performance model increases towards the top of the pyramid from Fig. 1. The subsequent sections describe each layer in more detail.

We specify the performance of component operations and activities in isolation in terms of processor and resource demand. For the composition of activities, performance is estimated in terms of usual performance measures like response time, processor utilization and throughput.

4.1 Modeling of Component Operations

The processor demand (e.g., execution time) of a component operation may depend on many factors: input parameters of the operation, diversity parameters of the component, observable state (history) of the component, etc.

---

\(^2\) Control flow graphs are notions similar to execution graphs and UML activity diagrams.
Considering all implementation details of component operations usually leads to combinatorial explosion. It is however often the case that about 80% of the processor demand of a component operation is determined by only 20% of its code.

We therefore propose to predict the processor demand of a component operation by applying techniques similar to the APPEAR method [5], [6], [7] developed by us.

As shown in Fig. 2, the APPEAR method divides the software stack into two parts: (a) components and (b) a Virtual Service Platform (VSP). The first part consists of evolving components that are specific for a product. Evolving components are components of which an implementation exists and that are supposed to undergo changes in future versions because of maintenance. If the new components are not sufficiently similar to the existing ones, the prediction can fail because the observation data are not relevant anymore. As explained below, the existing components are used to build a statistical prediction model for the not yet existing evolutions.

The second encompasses stable components that do not significantly evolve during the software lifecycle of a product. Usually, this is a significant part of the product software. Both parts can interact with the execution environment.

As a result of an input stimulus, a component can invoke several service calls of the VSP to perform the necessary functionality. After completing these calls and the internal operations, the component produces a response to the environment. The timing dependency between the stimulus and response can be characterized by some performance measure (e.g. response time, latency, average CPU utilization, execution time).

The APPEAR method describes the resource demand of software components by means of a statistical prediction model. This model reflects the correlation between the resource demand and the performance-relevant parameters of the components: the use of services calls, input parameters, diversity parameters, etc.

This vector of performance-relevant parameters is called signature type. It abstracts from internal details of a component that are not relevant for the performance. The values of this vector are called signature instances.
The initial version of the signature type is obtained by observing the documentation of a component and its execution traces. As shown below, the signature type might need refinement during the application of the APPEAR method.

The APPEAR method consists of two main parts (see Fig. 3): (a) training the prediction model on the existing component by means of a set of performance-relevant use-cases; and (b) applying the prediction model to obtain an estimate of the performance for the evolving component for the same use-cases. The arrows in Fig. 3 denote the information flow. The first part of the APPEAR method concerns both dashed and solid arrows, whereas the second part proceeds only along the solid arrows.

During the first part of the APPEAR method, a simulation model is built for its performance relevant parts, based on the design of the component. This simulation model is used to extract the signature instances that characterize the processor demand of a component operation for the various use-cases.

The prediction model relates the signature instances to the processor demand of a component operation. It is obtained by executing the existing component for the various use-cases. The prediction model is fitted such that the prediction error is minimized, which is indicated by the feedback loop.

During this training part, the signature type might require adjusting to fit the prediction model with sufficient quality. The quality of the fit is usually determined by the regression technique and the magnitude of the prediction error.

The prediction model $P_o$ for a component operation $o$ is described by a function over the signature type of the component operation:

$$P_o : S^{(o)} \longrightarrow D, \quad D \subseteq \mathbb{R}$$ (1)

In this formula, $S^{(o)}$ denotes a signature type, and $D$ is the processor demand of the component operation. Such a prediction model can be constructed by any regression technique. For instance, one can apply (multiple) linear regression [10], [15], which assumes that the processor demand depends on the signature parameters linearly:

$$D = \beta_0 + \sum_{j=1}^{k} \beta_j \cdot s_j + \varepsilon.$$ (2)
In formula (2), $D$ denotes the processor demand; $\beta_i$ are linear regression coefficients; $s_j$ are signature parameters; $k$ is the number of signature parameters; $\varepsilon$ represents the prediction error term.

The prediction model is calibrated on the existing components. Both already existing and not-yet implemented components are described in terms of signature parameters. The stability of the VSP allows one to use the same prediction model for both existing and adapted components.

Linear regression allows one not only to estimate the regression coefficients, but also provides means for judging the significance of each signature parameter. For instance, this can be done by hypothesis testing such as t-tests (see [10], [15]).

The following assumptions must be fulfilled to apply the APPEAR method:

1. The performance of a component depends only on its internals and the use of VSP services, but not on other components.
2. The services of the VSP are independent. Since the service calls are used as input for the prediction model, there should be no interactions that significantly influence the performance, e.g. via blocking of exclusive accesses to shared resources.
3. The order of service calls does not matter.
4. A large set of use-cases for training the prediction model is available. Tools for performance measurements and regression analysis of the components are available, e.g. tracing tools.

### 4.2 Modeling of Activities

Activities are described by a control flow graph (CFG). A node of CFG can denote the invocation of a component operation invocation, a branching element, or a loop element. An edge of CFG describes the possible control flow from one node to another.

The shape of a control flow graph (CFG) has usually a significant influence on the performance of an activity instance. We decided to describe relevant branches and loops outside component operations, as this significantly simplifies modeling. Examples of primitive CFG’s that demonstrate nodes of each type are enumerated in Table 1.

<table>
<thead>
<tr>
<th>Primitive CFG</th>
<th>Description</th>
<th>Figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>A sequence of nodes</td>
<td><img src="image" alt="Simple CFG" /></td>
</tr>
<tr>
<td>Branch</td>
<td>A branching element selecting only one of two nodes is selected</td>
<td><img src="image" alt="Branch CFG" /></td>
</tr>
<tr>
<td>Loop</td>
<td>A loop element that allows iteration on a sequence of nodes</td>
<td><img src="image" alt="Loop CFG" /></td>
</tr>
</tbody>
</table>

Table 1. Primitive ACFG’s with nodes of different types.
To estimate the resource demand of an activity instance, it is necessary to model the control flow through the activity. Depending on the data dependencies between different nodes and edges of the CFG, the control flow can be modeled by different methods (see Fig. 4).

![Fig. 4. Modeling of the control flow.](image)

In simple cases, this model can be an analytical expression. For instance, the resource demand estimation $D$ of a CFG with a branching node and two component operations (see Table 1) can be expressed by the following formula:

$$D = \pi \cdot D_1 + (1 - \pi) \cdot D_2.$$  (3)

In this formula, $D_1$ and $D_2$ are processor demands of component operations. These processor demands can be either budgeted or calculated using the APPEAR method described in section 4.1. The branching probability $\pi$ is a fixed number that is assumed to be known a-priori.

In other cases, the branching probabilities and numbers of loop iterations are cumbersome to estimate due to complex dependencies on the parameters. These branching probabilities and numbers of loop iterations are then described by a signature type (relevant parameters) and the corresponding prediction model (statistical regression). For the simple case described above, this results in the following formula:

$$D = p(s) \cdot D_1 + (1 - p(s)) \cdot D_2.$$  (4)

In this formula, $p(s)$ is the probability of branching to operation 1, and $s$ denotes the signature on which this probability depends. The prediction model for calculating the value of $p(s)$ is fitted on measurements by applying statistical techniques such as generalized linear regression [15].

However, obtaining a reliable estimate of the resource demand may require simulation, if the either CFG has a sophisticated shape or estimates of resource demands and branching probabilities are not statistically independent.

---

3 By analytical modeling, we understand the use of algebraic calculus.
4.3 Modeling of Activity Composition

Concurrent activity instances are scheduled on processors and non-sharable resources. Depending on aspects such as scheduling policy, resource access control protocols, activity arrival patterns, and the way the activity instances synchronize, accounting for the concurrency may require the use of different techniques (see Fig. 5).

Accounting for concurrency

Analytical models
Stochastic models

Simulation models
Deterministic models

Fig. 5. Accounting for concurrency.

Analytical models can only be applied when the choices regarding the aspects mentioned above satisfy the assumptions and the resulting model can be constructed with reasonable effort. For instance, when activity instances arrive according to a Poisson stream and only restricted resource contention occurs, it is possible to construct a stochastic queuing model for predicting the average response time of an activity. In very simple cases, it is even possible to build a deterministic model, which is described by a formula.

However, for many systems the assumptions of analytical models are severely violated, and simulation models have to be used.

5 An Example

We illustrate our approach with a simple example of a hypothetical Automobile Navigation System (ANS). In this example, we aim at the estimation of response time of a particular activity of the system.

5.1 The Basic ANS Functions

An ANS has to assist the driver both in constructing and maintaining a route. The basic functions of this system are the following:

1. Acquisition of route information. The ANS obtains the coordinates of a geographical position from a GPS receiver and traffic jam information from a radio receiver.
2. Constructing and controlling the route. After the user specifies the departure and destination points, the system constructs the route. The ANS checks if the user maintains the planned route.
3. Interfacing with the user. The ANS allows the user to choose the route in an interactive fashion. The system is responsible for displaying the map, route and current
position of the car during the trip. Additionally, the system notifies the user with a
dedicated message (picture and/or voice) about approaching the turns he or she
should follow. Messages are displayed only before turns and crossings, where the
driver should execute a maneuver.
The ANS consist of the components depicted in Fig. 6.

![Fig. 6. The ANS component diagram.](image)

Each component implements an activity and the interfaces for its control. The ac-
tivities are independent and have the same names as the components. Since the in-
put/output devices are assigned to only one component, the only shared resource is the
processor.

### 5.2 Activities and Component Operations of the ANS

There are four concurrent independent activities in the ANS:
- The “GPS” activity, responsible for acquiring the current coordinates from GPS,
- The “Control” activity, responsible for regular checking and updating the route
  information,
- The “Display” activity, responsible for displaying the map, route, messages, etc,
- The “User command” activity, responsible for handling user commands.

For the sake of simplicity, we restrict ourselves to the “Display” activity shown in
Fig. 7.

![Fig. 7. Activity control flow graph of the "Display" activity.](image)

The rectangles in Fig. 7 denote the invocation of component operations, and the
circles indicate branching. The arrows denote the precedence relationship. Notice that
all component operations belong to different subcomponents of the “Display” compo-
nent (see Fig. 6).

Whether the route needs displaying or not is determined by a parameter B1. The
moment of displaying a message depends on the current speed and the distance to the
next turn.
First, we need to construct a prediction model for the processor demand of the “Map” operation. This component operation draws the map on the screen. The map can include various objects: roads, houses, trees, lakes, special signs, etc. The signature type of this operation is the following:

\[ S_{\text{map}} = (N, L, Z). \]  

In formula (5), \( N \) is the number of objects such as buildings, roads, etc to be displayed; \( L \) is the level of details (determines the size of roads to be displayed); \( Z \) is the scale of the map.

The corresponding prediction model, constructed by linear regression, has the following form:

\[ D_{\text{map}} = \beta_0 + \beta_1 \cdot N_{\text{obj}} + \beta_2 \cdot L + \beta_3 \cdot Z \]  

The parameters \( \beta_j \) are fitted on the measurements collected by benchmarking the component performance for different use-cases, when the component executes in isolation. For other component operations, similar formulas hold.

Second, we construct prediction models for the branches of the “Display” activity. The user can turn on or off the displaying of the route by a switch that controls a binary variable \( B_1 \), indicating whether the route needs displaying or not. For the sake of simplicity, we assume that the probability \( P_{B_1} \) of the displaying the route is fixed.

To account for the processor demand of the “Msg” operation, a prediction model must be built for calculating the probability of taking the branch with or without calling the “Msg” operation. The model is fitted using a two-parameter signature type

\[ S_{B_2} = (V, A). \]  

In this formula, \( V \) denotes the current speed of the car, \( A \) is the type of area where user is driving. The second parameter is categorical, and it can have two values: City or Highway. On one hand, the higher the speed, the faster the user can reach the turn that he needs to take. On the other hand, it is more likely to encounter the turn in the city than on the highway.

For the regression of data models with binary responses that have a distribution from the exponential family (Poisson, binominal, etc), the so-called logit link function

\[ \eta = \log \left( \frac{p}{1-p} \right) \]  

is often used.

The prediction model for the branching probability can be described now by the following formulas:

\[ \eta = \alpha_0 + \alpha_1 \cdot V + \alpha_2 \cdot A; \quad \eta = \log \left( \frac{P_{B_2}}{1-P_{B_2}} \right). \]  

Finally, summarizing all the models above, it is possible to build a formula that estimates the processor demand of the entire “Display” activity (see Fig 7):
\[ D_{\text{display}} = D_{\text{map}} + P_{B_1} \cdot D_{\text{route}} + D_{\text{pos}} + P_{B_2} \cdot D_{\text{msg}} \]  (10)

In this formula, \( P_{B_1} \) is a binary parameter determining whether the route needs displaying; \( D_o \) denotes the processor demand of component operation \( o \); and \( P_{B_2} \) is the probability that a message has to be displayed. If we are for instance interested in estimating the worst-case processor demand of the “Display” activity, we have to choose the signature instances for calculating \( D_{\text{map}} \) and \( P_{B_2} \) such that their values are close to maximum. As a result we obtain the following estimates for the terms of formula (10):

\[
D_{\text{display}} = D_{\text{map}} + P_{B_1} \cdot D_{\text{route}} + D_{\text{pos}} + P_{B_2} \cdot D_{\text{msg}} = \\
= 11.5 + 1.0 \cdot 3.36 + 0.75 + 0.9 \cdot 2.1 = 17.5 \text{ ms}. 
\]  (11)

The processor demand of other activities can be modeled in similar way.

5.3 Activity Composition

All activities from section 5.2 are independent and periodic. They execute on a single processor scheduled by a round robin scheduler, which assigns the processor to each activity instance for a short quantum of time. The chosen quantum duration is 10 ms. We assume that context switch times are negligible in comparison with the processing demand of the activities. Since the deadlines are quite large, we also assume that all devices are polled and no interrupts occur.

We need to estimate the response time \( T_{\text{display}} \) of the “Display” activity in composition with the other activities.

Table 2 enumerates all important parameters of the ANS activities. The average execution times of these activities are obtained by applying the APPEAR method as described in section 5.2. These times are multiplied by a factor 1.2 to obtain the corresponding worst-case execution times (WCET). This factor\(^4\) is needed to ensure that the obtained WCET estimates are safe and can be used to analyze schedulability. Notice the value of this factor should be chosen carefully, taking into the account the accuracy guaranteed by the APPEAR method and the variation of processor demand estimates.

Fig. 8 demonstrates the worst-case schedule of all activities from Table 2 and a quantum size of 10 ms.

Arrival periods are subdivided into 50 ms intervals. Since the Cmd and the Disp activity have a deadline of 100 ms, they are only released in every second interval. Consequently, the GPS, User command and Display activities are released at each even interval, whereas the GPS, Control and Display activities are released at each odd interval.

---

\(4\) This number is usually based on the experience of the architects, and dependent on the type of product and software.
It is easy to show, that at least 8 ms slack remains, which proves the schedulability of the entire ANS, given that interrupts and various overheads do not exceed 8 ms in each interval.

Notice that the WCET of the Display activity is the largest of all other ANS activities. This means that a Display activity instance terminates in the worst-case later then all other activities in both odd and even 50 ms intervals.

Summarizing, the worst-case response time $T_{\text{display}}$ can be calculated by the following formula:

$$T_{\text{display}} = \max \left( W_{\text{Disp}} + W_{\text{GPS}} + W_{\text{Cmd}}, W_{\text{Disp}} + W_{\text{GPS}} + W_{\text{Control}} \right).$$

In this formula, $W_A$ denotes the worst-case processor demand of an activity $A$.

By substituting the values of worst-case processor demands from Table 2, we calculate $T_{\text{display}} = 42 ms$.

Although this example is quite simple, it should be sufficient to explain the essentials of our method. For activities that synchronize/communicate and for other scheduling disciplines, the composition becomes much more complex.

### 6 Conclusions

In this paper, we have presented an approach to performance prediction of component compositions.

A component composition is considered in terms of concurrent activities that invoke a number of component operations. The approach is stepwise: first, performance models are constructed for component operations; then the models of activities are built; finally, the concurrent activities are treated.
The approach employs the basic principles and techniques from the APPEAR method to abstract from the implementation details of component operations to a signature and to construct prediction models for branches and loops of an activity.

An important aspect of the approach is the possibility to choose between analytical and simulation modeling techniques at each step. The architect can select the appropriate technique based on (a) satisfaction of the assumptions of the techniques by the software under consideration, (b) the goals of the analysis, (c) the required accuracy, and (d) the timing budget available for the estimation.

The approach is illustrated by an example of an Automobile Navigation System. This example demonstrated the important steps of our stepwise approach.

Future work will be done in the following directions:

1. Validation of the approach with a number of industrial case studies.
2. Elaboration on rules and guidelines for selection between analytical, statistical and simulation approaches.
3. Elaboration on the constructing of prediction models for activities containing branches and loops.
4. Formalizing the notion of similarity between existing and evolving components.

Acknowledgements

We are grateful to Sjir van Loo, Henk Obbink, from Philips Research Laboratories and Michel Chaudron from Technical University of Eindhoven for constructive discussions about our approach and about the APPEAR method. The work presented in this paper was conducted within the AIMES project (EWI.4877) and funded by STW.

References

TESTEJB - A Measurement Framework for EJBs

Marcus Meyerhöfer and Christoph Neumann
Friedrich-Alexander University of Erlangen and Nuremberg
{Marcus.Meyerhoefer,sichneum}@immd6.informatik.uni-erlangen.de

Abstract. Specification of Quality of Service (QoS) for components can only be
done in relation to the QoS the components themselves are given by imported
components. Developers as well as users need support in order to derive valid data
for specification respectively for checking whether a selected component complies
with its specification. In this paper we introduce the architecture of a measurement
framework for EJBs giving such support and discuss in detail the measurement of
the well understood property of response time.

1 Introduction

The concept of component-based software development has gained much attention in
the last years as mature component models, especially on the server side have evolved.

A key factor for success in a component market is the specification, but beside
the functional interfaces additional levels should be addressed [1], defining on each
level what the “user” of the component can expect, given that the requirements of the
component are met. In the COMQUAD project1 we are focusing on the specification
of non-functional properties (NFPs) [2] of software components, using an extension of
the language CQML, called CQML+ [3] that enables the specification of NFPs, their
relationships and additional resource properties.

In order to offer support for the determination of specific properties for specification
of components as well as for runtime monitoring, we aimed at a solution that enables to
measure selected NFPs and at the same by its non-intrusive approach provides means
to monitor components. In this paper we introduce the architecture and concepts of a
measurement environment for components following the Enterprise Java Beans (EJB)
specification. Although the all-embracing concept should be capable of measuring many
different NFPs, we focus in this paper on the measurement of response time (RT) as the
importance of RT for any kind of application is commonly understood.

This work is also a preparation for deriving call dependencies between interacting
EJBs, because in an assembly of interacting components, not only the performance of
an individual component is actually what matters, but also the invocation frequentness
and dependencies. But this will be topic of future work.

1 “COMponents with QUantitative properties and ADaptivity” is a project funded by the Ger-
man Research Council (DFG FOR 428). It started October 1, 2001, at Dresden University of
Technology and Friedrich-Alexander University of Erlangen and Nuremberg.

© Springer-Verlag Berlin Heidelberg 2004
2 Related Work

The COMPAS framework [4] aims at a solution aiding developers of distributed, component-oriented software in understanding performance related issues of their applications. Therefore, it consists of a monitoring module to obtain information about a running application, a modeling module to create and annotate models of the application with performance parameters gathered and an environment for the prediction of expected performance under different load conditions.

EJB Monitor [5,6], the monitoring module of COMPAS, is developed as an own project. The monitoring approach applied is based on the proxy pattern; for each EJB in an application a proxy bean is generated, stealing the JNDI name and therefore appearing as the bean to be monitored; thereby it is possible to receive, monitor and forward the original invocations. Although transparent to the client and the other components, this “heavyweight” approach introduces an additional indirection between each caller and callee: an additional marshalling/unmarshalling between the proxy-bean and the original bean takes place for remote interfaces. Furthermore the amount of EJBs is doubled if each component should be monitored.

Commercial environments like PerformaSure from Quest Software or OptimizeIt ServerTrace from Borland offer software to monitor J2EE applications and pinpoint bottlenecks. They cover a broader spectrum of J2EE integrating tools to monitor database connections, network traffic or virtual machine parameters. For EJB-components e.g. OptimizeIt ServerTrace supports method-level drilldown of performance; however, these tools are primarily targeted to support pre-deployment teams and to reveal performance bottlenecks if occurred in productive use and therefore limit themselves to performance properties.

3 Background and Environment

The component model of our environment are the J2EE 1.3 platform specifications and the EJB 2.0 specifications. The common EJB component implements an interface that allows the remote method invocation and the control of its life-cycle. The EJB component is not viable without a special runtime environment: any EJB component has to be deployed to an EJB container. The container runs on a JVM and provides middleware services to the EJBs. These services are for example transactional support, persistence, security mechanisms and the registration to a central component registry. Our run-time environment consists of the application server JBoss running on a Sun JVM 1.4.2. JBoss is open-source software and the release version 3.2.1 fully supports the EJB component model. JBoss is based on the Java Management eXtensions (JMX). JMX allows the implementation of a modular J2EE architecture with JMX as microkernel and the modules represented as JMX’ MBeans. The most important part of JBoss in our context and related to JMX is the EJB container module.

Interceptor is a design pattern of a callback mechanism for middleware remote invocations. The component container defines the interceptor callback interface with its hook methods and the implemented concrete interceptors act as callback handlers during a method invocation. This allows the non-intrusive integration of services to the com-
ponent container. OMG has already adopted and standardized interceptors as Portable Interceptors and the J2SE 1.4 is compatible to them.

JBoss’ interceptors can be configured on a per-EJB basis: Every EJB deployed to JBoss has – according to its type – several standard interceptors such as security or logging. An installed interceptor is involved with all the J2EE invocations to the bean and thereby allows the insertion of any additional functionality into the invocation path of a component invocation – either at the client-, server-, or both sides of the invocation. But note that client- and server-side interceptors inherit from separate interfaces. The multiple interceptors that are installed for an EJB are executed consecutively during each request. The installation is done by adding a JBoss-specific XML line to the vendor-specific deployment descriptor (for the JBoss container this is META-INF/jboss.xml).

4 Proposed Solutions

One of the most basic NFPs of a component is the RT of an invocation at one of its exposed functions. We assume that there exist EJBs from many different vendors that are not available as source-code. So first of all the EJB acts as “black-box” and is only available as byte-code. At first we identify several layers in any J2EE implementation at which measurements can be taken. Afterwards, we introduce our chosen approach explaining the concept and its various applications and discuss the overhead of our method. The comprising architecture of our solution is presented in Fig. 1, but a thorough understanding will only be achieved after Sect. 4.3 has been read.

![Architecture view at the measurement framework.](image)

**Fig. 1.** Architecture view at the measurement framework.

4.1 Layers for Instrumentation

There are several levels in the J2EE environment at which measurements are possible: The most simple approach would be to instrument the clients of the EJBs to measure RT.
We focus on a scenario where existing clients are calling productive beans and with client applications not available as source code. Therefore instrumentation of clients can’t be considered and the mechanism has to be transparent to the client’s source-code – even though an adaption of the client’s environment is inevitable.

The bottom level of possible instrumentation is the Operating System (OS). The instrumentation of the kernel(-modules) allows the entire monitoring of network traffic so the J2EE traffic could be filtered. A similar idea where RMI calls are filtered by modification of the EJB server code is given in [6], but this approach is neither portable across operating systems (OS) – respectively EJB containers – nor easy to implement and to maintain for coming versions of any OS.

Especially in J2EE the layer above the OS is the Java Virtual Machine where the measurement of time-information seems possible by using standard interfaces like the Java Virtual Machine Debugger Interface or the Java Virtual Machine Profiler Interface. But again a filtration would be necessary. Another alternative at this layer is the instrumentation of the java.net.* classes [7] having the same implications as the OS instrumentation.

The next level is the J2EE server respectively the EJB container. One alternative is to enhance the source-code of the EJB container. The best location to introduce the sensors for RT seems to be the framework that automatically generates the EJB Home and the EJB Object. But this would result in the duty to maintain the code for coming versions of the container and lacks the portability to other container implementations.

The highest level before the already rejected application level are the EJBs themselves. At this level the client-side RT can obviously not be measured but a server-side RT could be provided as described for the “EJB Monitor” in Sect. 2.

The non-intrusive instrumentation of the EJB container by using a callback mechanism like interceptors seems most promising because both client- and server-side are considered.

4.2 Response Time Interceptor

In a client-server setting there are several definitions of RT possible [8]; however, the client-side response time $T_{\text{resp,client}}$ including communication overhead and the server-side response time $T_{\text{resp,server}}$ are the most important ones. The injection of interceptors on both sides of the invocation allows for the transparent measurement of $T_{\text{resp,client}}$ and $T_{\text{resp,server}}$ by timestamping the delegation.

The placement of the corresponding response time interceptors (RTI) is crucial, because the sequence of execution of the interceptors is directly related to the textual order in the manifest/jboss.xml-file. The top one is invoked first and the bottom one invoked at last before the call is delegated to the EJB itself. Therefore the RTIs have to be inserted as the last one ($T_{\text{resp,server}}$) respectively the first one ($T_{\text{resp,client}}$) in the interceptor chain. The possibility to place the RTIs arbitrarily enables additional semantics of the measured times.

As mentioned in Sect. 3 the interceptors of client- and server-side do not share a common interface so they are functionally different, but we converge the behavior to a common logical one: a response time interceptor RTI, has to take four timestamps (TS) as shown in Fig. 2, whereas $i \in [1; n]$ equals the logical position of the RTI instance in the
Fig. 2. The general behavior of RTIs.

RTI chain – only the \( n \) RTIs are iterated, not the other interceptors. Two are needed for calculating the RT and the remaining ones in order to account for the inserted overhead. The overhead between TS\(_1\) and TS\(_2\) mainly consists of initialization and gathering of minimal context information. Between TS\(_3\) and TS\(_4\) more has to be done: collection of additional context information, calculation of RTs, construction and transmission of a transfer object (TO) to an asynchronous storage mechanism (see Sect. 4.3).

Because the callback to the RTIs produces overhead, there are two views at both RTs. The \textit{uncleansed RT} ignores this fact and is calculated as \((\text{TS}_3 - \text{TS}_2)\). The \textit{cleansed RT} considers the overheads – each interceptor calculates and transmits the accumulated correction value (ACV), shown exemplarily for RTI\(_i\):

On the inward way \( \text{ACV}_{(i-1)} := \text{ACV}_{(i-1)} + (\text{TS}_2 - \text{TS}_1) \), and on the outward way \( \text{ACV}_{2n-(i+1)} := \text{ACV}_{2n-(i+1)} + (\text{TS}_4 - \text{TS}_3) \).

In order to account for the overhead of the RTI\(_x\) : \( x > i \) the received correction value \( \text{ACV}_{2n-(i+1)} \) has to be used, but as it contains the whole overhead and not just the one starting at TS\(_2\) of RTI\(_i\), that part represented as \( \text{ACV}_{(i)} \) has to be subtracted. Therefore the cleansed RT is calculated as \((\text{TS}_3 - \text{TS}_2) - (\text{ACV}_{2n-(i+1)} - \text{ACV}_{(i)})\).

The TSs are taken by a JNI module that uses OS system calls like \texttt{gettimeofday} due to the lack of precision of \texttt{System.currentTimeMillis()}, which is the fallback of the wrapping Java class if the module was not deployed to the client. The unit in contrast to the accuracy is in either case \( \mu \text{s} \). Global synchronization of clocks is not needed because the RT is a difference of local times. As soon as there is the interest to derive causal relations by temporal order the synchronization of clocks could be useful.
The ordinary use-case consists of one RTI on both client- and server-side measuring client- and server-side RT, whereby client-side RTIs are only required for call dependency derivation, because they allow to identify invocations by real client applications.

The interceptor chain consists always of interceptors correlated to the provided middleware services like transactional support or security. The overhead of their executions can be measured by wrapping the interceptor of interest between two RTIs and storing not only the RT but also the TSs. The TSs of an individual RTI is now represented as \( \text{TS}_x^{(i)} \) where \( x \in \{1, 2, 3, 4\} \) and \( i \) as above. The overhead of a chosen non-RTI interceptor that is wrapped between RTI\(_{(i)}\) and RTI\(_{(i+1)}\) can be calculated as follows:

\[
(\text{TS}_1^{(i+1)} - \text{TS}_2^{(i)}) + (\text{TS}_3^{(i)} - \text{TS}_4^{(i+1)})
\]

The interceptor-based measuring approach therefore allows not only to identify hot-spot EJBs and performance bottlenecks, but even to track down the constituents of the RT experienced by a client by wrapping the interceptors implementing middleware services. By using the interceptor mechanism on a per-EJB basis (see Sect. 3), a developer is enabled to monitor just some selected components, e.g. long-running ones which then might be split up in subtasks. Such configuration can even be changed at runtime using the hot-deployment feature of JBoss, facilitating the transparent “attachment” of the measurement mechanism to an already deployed component.

4.3 Storing the Measurement Data

The information gathered by the RTIs during \( \text{TS}_1 \) to \( \text{TS}_2 \) and \( \text{TS}_3 \) to \( \text{TS}_4 \) is wrapped as TO. A JMX’ DataCollector-MBean (DCMB) is used as buffered and therefore asynchronous storage mechanism for the TOs. The buffer reduces overhead during call-time by taking the cost-intensive persistency mechanism out of the return path, because the TOs are only transferred to the MBean but not flushed until the buffer’s (manageable) capacity exceeds. Apparently the client-invocation is also made fail-safe against the underlying database (DBS), because an error can only occur during the flushing to the DBS, ergo when the client-call is all over and done. The DCMB implements a publisher that allows subscribers to register. Currently a basic subscriber stores the TOs of interest to the DBS; adoption of a generic and transparent storage mechanism like Java Data Objects is planned. The comprehensive architecture of our solution has been shown in Fig. 1.

The DCMB is available in a direct and cost-efficient way only at server-side. The client-side RTIs have to transfer the TOs to the server’s MBean by some remote call. Because of the static nature of interceptors this has to be done before the call is allowed to return to the client. The remote access to the MBean is done by a Proxy-EJB, but as soon as the new JMX Remote API 1.0 is integrated into JBoss an adoption of our implementation is planned. As an optimization of client-side’s overhead only the last client-side RTI on the return path collects all client-side TOs and transfers them by only one remote call to the DCMB.

Because the measurement of the RT is transparent to the client, a client that is interested in its RT has to query the DBS using an unambiguousness tuple that represents one of its invocations (e.g. invocation-near TS, IP, parameters). But most assumed clients are not interested in their RT values after all because analysis is done by the administrators or developers and takes place only by the stored data in the DBS.
4.4 Overhead

A software approach to measurement has always some overhead that is never accountable exactly. Besides the efforts to minimize the overhead and to maximize the accuracy the following influences have to be considered\(^2\): On server-side the JNI module should always be deployed, otherwise the accuracy is reduced to \(\text{System.currentTimeMillis()}\). Although the unit of \(\text{currentTimeMillis()}\) is \(\text{ms}\), its accuracy can only be assumed “in units of tens of milliseconds” [9].

The accountable overhead introduced by the RTIs that results in a greater perceived response time by the client can be gained for each RTI and call individually by the stored data as \((\text{TS}_2 - \text{TS}_1) + (\text{TS}_4 - \text{TS}_3)\). Measurements have shown that the insertion of one RTI on the server-side – this one will have to create both TOs for MetaData and RT – introduces overhead around 1.0 \(\text{ms}\) (the RTI on client-side introduces more overhead as discussed below). More RTIs increase this number only by 0.1 \(\text{ms}\) each, because most overhead is related to the first and last one.

Because the \(\text{TS}_4\) of the first RTI (RTI\(_1\) being the last returning one) can’t be stored its overhead is not accountable. Mostly the RTI\(_1\) will be at client-side transferring the client-side TOs by a remote call to the DCMB so its overhead has to be respected. Measurements have shown that its duration is in units of 10 \(\text{ms}\). Therefore, we use a separate thread for this storage, allowing the RTI to return after 400 \(\mu\text{s}\) (about 289.325 \(\mu\text{s}\) for thread creation, as the average of 10000 tests). Anyway, for analysis this is a minor issue because the \(\text{TS}_3\) of the RTI\(_1\) represents nearly the same time the client would receive the result if the RTIs would not be present.

The JNI mechanism introduces non-accountable overhead. Tests showed that the JNI call to the Linux’ \text{gettimeofday}(2)\) lasts about 0.970 \(\mu\text{s}\) as the average of 10000 tests. The pure \text{gettimeofday}(2) call by a C program takes about 0.254 \(\mu\text{s}\) for the same number of tests.

Calculation of the ACVs and putting them into the forward- or backward-passed environment maps in order to transmit them to the next participant in the logical RTI chain must inevitably take place after the \(\text{TS}_2\) and \(\text{TS}_4\) are done. Temporarily timestamping the relevant code-segments and run-time printing of the differences showed that each RTI adds about 5 \(\mu\text{s}\) of never accountable overhead. If necessary, this knowledge can be used during analysis for a more accurate interpretation of the data. Finally the gap between RTI\(_1\)’s return and the client’s receiving as discussed in Sect. 4.1 was measured to be below 1 \(\text{ms}\).

5 Conclusion

In this paper we have introduced the concepts and the implementation of a measurement environment for components following the EJB specification. Based on the interceptor pattern, it has been shown how this is applied for measurements in the open-source application server JBoss. Following a discussion of several possible layers of instrumentation,

\(^2\) All measurements were taken on a standard PC system with an AMD K7 1400 MHz processor and 1 GB SDRAM running Linux SuSE 8.2, Kernel version 2.4.20-4GB-athlon.
the application of our concept is explained in detail for response time measurement: bas-
sically, it is possible to not only measure client- or server-side response times, but also
determine delays introduced by standard interceptors in the call chain wrapping them
with RTIs. Because a software solution always introduces some overhead, we have
shown how to calculate cleansed response times using correction values (ACVs) and
have discussed the costs of JNI and the transmissions between interceptors which are
not included in the ACVs.

We believe that the usage of interceptors allows for measurement of many different
properties of components besides the response time, e.g. jitter or throughput. By now
using the information stored by the RTIs not only the response time can be determined,
but also call-dependencies between the different EJBs of an application. In subsequent
work we plan to analyze those dependencies more deeply and to extend our framework
to some selected additional properties.

References

   IEEE Computer 32 (1999) 38–45
2. Meyerhöfer, M., Meyer-Wegener, K.: Estimating non-functional properties of component-
based software based on resource consumption. In: SC 2004: Workshop on Software Com-
   Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse,
4. Mos, A., Murphy, J.: A framework for performance monitoring, modelling and prediction of
   component oriented distributed systems. In: Proceedings of the third international workshop
   on software and performance. (2002) 235–236
5. Mos, A., Murphy, J.: Performance monitoring of Java component-oriented distributed applica-
   tions. In: 9th International Conference on Software, Telecommunications and Computer
   In: 9th International Conference on Software, Telecommunications and Computer Networks.
   the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages,
8. Röttger, S., Zschaler, S.: Model-driven development for non-functional properties: Refine-
   http://java.sun.com/j2se/1.4.1/docs/api/java/lang/System.html.
    Engineering, Toulouse, France. In Bruel, J.M., ed.: Proc. 1st Intl.Workshop on Quality of
    Service in Component-Based Software Engineering, Toulouse, France, Cépaduès-Éditions
    (2003)
Model-Based Transaction Service Configuration for Component-Based Development

Sten Loecher

Dresden University of Technology
Department of Computer Science
Sten.Loecher@inf.tu-dresden.de

Abstract. An important part of the software engineering process in today’s component technologies is the integration of business logic and infrastructure services. In this paper, we investigate the current situation regarding transaction management services and discuss existing problems. We then present a conceptual framework for a model-based transaction service configuration approach and explain its constituent parts. The framework is based on metamodelling and thus can directly be used for efficient development of tool support.

1 Introduction

Two primary objectives of software engineering methodology are to cope with the inherent complexity of large software systems and to provide concepts to enable better reuse of already existing artifacts. For this purpose, component technologies currently used to build server-side business applications, e.g., Enterprise JavaBeans (EJB) [1], integrate two paradigms. On the one hand, component-based development (CBD) [2] is used to structure the application-specific logic or business logic, respectively. On the other hand, the separation of aspects [3] is used to decouple application-independent logic from the application-specific part. Application-independent logic comprises infrastructure services like security, persistence, and transaction management. At run-time, these services are provided by the runtime environment of the application, also known as container. However, the business logic and infrastructure services have to be integrated eventually. This can be done either programmatically by inserting control statements into the source code of components or declaratively by attaching pre-defined attributes to the provided operations of components. The attributes are then interpreted at run-time by the container to apply services properly.

The focus of our work is on declarative configuration of transaction management services. A transaction service configuration describes the transactional behavior of an application. This includes transaction demarcation, dependencies between transactions, and the behavior in case of failures. We concentrate on the declarative configuration approach, because we consider it more suitable for efficient software development. This is justified by the capability to clearly separate the design decisions regarding business and transaction logic and their implementations [4]. As a result of the separation of

⋆ This work was supported by grants of the Deutsche Forschungsgemeinschaft, Graduiertenkolleg 191.

© Springer-Verlag Berlin Heidelberg 2004
concerns, an independent processing of the two aspects by domain experts is possible. Moreover, the configuration of transaction services by a pre-defined set of attributes provides a simple, yet precise, and explicit way to specify the required transactional behavior of a component. This is an important prerequisite for predictable behavior.

However, we see a number of problems with the declarative approach as applied currently in component frameworks. These problems can be categorized into methodological and technological issues. From a methodological point of view, matters of responsibility and configuration time have to be discussed. Technically, more flexibility and a better integration of CBD and transaction service configuration is required. To provide adequate means for the configuration of transaction services and to tackle these problems, we base our work on a conceptual framework that enables model-based configuration. Because we would like to provide a basis for efficient development of tool support, we use metamodeling to describe these concepts.

After a discussion of related work in the following section, the current problems regarding the configuration of transaction management services are elaborated in Section 3. The model-based configuration approach and is described in Section 4, which also discusses the conceptual framework and its constituent parts. We conclude by giving a summary of the paper and provide information about future work in Section 5.

2 Related Work

Transaction services are discussed extensively in the context of Architecture Description Languages (ADL). These languages generally comprise three basic constituents, namely components, connectors, and architectural configurations [5]. Components are used to model the business logic of an application whereas connectors are means to model the interaction between components and to describe infrastructure services. Architectural configurations describe actual architectural structures consisting of components and connectors. Transaction services are dealt with by several authors. In [6] a quite general discussions about connectors with respect to transaction services can be found. [7] contains a valuable investigation about the impact of connectors to the behavior of components. Formal approaches to describing connectors and transaction services can be found, for example, in [8]. In our work, we have to deal with similar issues and can therefore use the work on ADLs as source of conceptual information.

Several authors also recognized the problems and weaknesses of current practice in transaction service configuration. In [9], the current situation regarding transaction management services in EJB is investigated. Prochazka proposes in [10] a new approach to declaratively configure transaction services, the so-called NT&CT approach. This approach provides a rather generic set of attributes with respect to transaction models by which an application can be configured. A container that supports the required transactional concepts has also been developed. Although a significant contribution to improve the situation for transaction service configuration, methodological issues are not considered within that work. The authors of [11] propose a framework that allows the definition of configuration attributes for transaction services, which is also an objective of our work.
3 Analysis of Current Situation

As introduced in the first section, declarative configuration of transaction management services is currently based on the concept of pre-defined configuration attributes attached to the provided operations of a component. The EJB framework, for example, provides a set of six predefined attributes that can be used to describe the required transactional functionality of an operation, namely NotSupported, Required, Supports, RequiresNew, Mandatory, and Never. To illustrate the use of such attributes, the classical example for motivating transactions is used. Be CreditTransfer a component that implements the business logic for transferring funds between accounts. For this purpose, it provides an operation transfer via the interface ICreditTransfer that guarantees to send debit and credit messages to the corresponding accounts, which must exist:

context ICreditTransfer::transfer(String a1, String a2, Real amount)
  pre : IAccountHome.existsAccount(a1) and IAccountHome.existsAccount(a2)
  post: let acc1:IAccount = IAccountHome.getAccount(a1), acc2:IAccount = IAccountHome.getAccount(a2)
        in acc1.debit(amount) and acc2.credit(amount)

The correct execution of the transfer operation in a concurrent and distributed environment requires the debit and credit operation to be executed within the scope of a transaction. For this, the transfer operation is configured using the attribute RequiresNew. This tells the container to start a new transaction upon the invocation of the transfer operation. The configuration information is usually stored in so-called deployment descriptors, which are associated to each component and read by the container. For a more detailed discussion of the EJB transaction service configuration capabilities, see for example [1].

Although simple and efficient, we see a number of problems with the attribute-based approach as applied currently in the EJB framework and likewise in other component technologies:

Attribute Set. First of all, we think that the provided set of pre-defined attributes is too restricted with respect to the number of possible configurations that can be specified. This is clearly a result of supported transaction models. Although advanced transaction models and techniques [13] have been developed and applied within database management systems, this situation is not reflected in current component technologies and transaction processing standards like [1] and [14], respectively. From our point of view [15], which is shared by other authors [10, 9], this situation is not satisfactory because server-side business applications participate in

---

1 For describing the behavior of the transfer operation, we use the Object Constraint Language [12]. We also make the common distinction between life-cycle management interfaces (Home) and business logic interfaces.
long-running transactions that require advanced transaction models and techniques [13]. Therefore, greater flexibility is required in a sense of extending the pre-defined set of attributes to support the variety of models and techniques. Also, more transparency is needed with regard to the information encoded within the pre-defined attributes. Attributes encode, for example, information about client context handling, transaction demarcation, exception throwing, and inter-transaction dependencies [10]. To increase the usability of the attributes for users and tool developers, this information must be made explicit.

**Responsibilities.** Responsibilities for the usage of the pre-defined attributes are not clearly established. In the EJB-specification, for example, the task to attach attributes to the provided operations of a component is assigned to either the component provider or the application assembler. If neither of them fulfills this obligation, the deployer of the application is expected to carry out the configuration. From a methodological point of view, this situation is not satisfactory.

**Configuration Time.** Another problem is the time of configuration. The actual configuration is done on a rather ad-hoc basis, i.e., after assembling the application. To recognize design errors in the transaction design early in the software engineering process, which is required to avoid possibly high life-cycle costs, a model-based configuration approach is favorable and must be supported accordingly.

**Component-Based Concepts.** Finally, a problem of current component technologies is the missing explicit declaration of context dependencies of components, which is a prerequisite for CBD. In particular for the case of transaction management services, there exists no way of describing the transactional functionality required by used components. For the CreditTransfer component, for example, the requirement of running the credit and debit operations within the same transaction started upon invocation of transfer cannot be declared.

To put it in a nutshell, we see the need for more flexible and transparent configuration mechanisms, clearly established responsibilities for the configuration process, transaction service configuration based on models, and a better integration of component-based concepts. To provide more adequate means for the configuration of transaction services and to meet these needs, we base our work on a conceptual framework, which is introduced in the following section.

### 4 Model-Based Transaction Service Configuration

Figure 1 illustrates the basic idea of a model-based configuration procedure. We have chosen a model-based approach to transaction service configuration because it allows an early recognition of errors in the transaction design and increases predictability of the application through analysis.

The lower part of Figure 1 shows the system view. An application is assembled from pre-fabricated components. A configuration is attached to the application, describing the required services, such as security, persistence, and transactions. The container, which provides these services, interprets the configuration during the execution of the application. The transaction service usually implements a specific transaction model. The upper part of Figure 1 illustrates the model view. Starting point for the model-based engineering approach is a repository of component specifications. Components
Model-based configuration procedure.

and corresponding specifications are delivered by component providers. An application designer builds an application model from these specifications, which is illustrated by arrow one. The transaction designer configures the application using configuration models. This is illustrated by arrow two. Finally, depicted by arrow three and four, the application assembler builds the actual application and configures it based on the corresponding models.

In our work, we focus on the activities combined by arrow two in Figure 1, namely the configuration of application models, their analysis regarding required properties like deadlock freedom, and the adjustment of configurations to meet properties more adequately. For this purpose, we have defined a conceptual framework depicted in Figure 2, which is a refined version of the gray rectangle in Figure 1. The framework comprises three core elements, namely a business logic model, a configuration model, and an integrated model. For describing the concepts on which these models are based, we use metamodelling. Metamodelling provides a good basis for efficiently developing tool support. The metamodels can be used, for example, to automatically generate an infrastructure using technologies like the Java Metadata Interface (JMI) [16] and corresponding implementations. Such an infrastructure consists of classes and interfaces of a specific programming language that represent the corresponding metamodel and allows the programmer to manage instances of it. This infrastructure can be used as a basis to implement analysis algorithms, for example.

The business logic model, which is the result of assembling component specifications, contains the essential information of the application-specific part. This includes information about the employed components and their provided and required interfaces as well as specifications relating these interfaces to each other. In our opinion,
Fig. 2. Conceptual framework for model-based configuration.

a complete formal specification of a component is too much for practical purposes [17]. Therefore, we require only information necessary for the analysis of transaction service configurations to be specified, such as messages sent by a component as well as their temporal ordering. The business metamodel is based on the study in [18] since the underlying concepts of the defined core modeling language match our requirements, in particular the concept of local histories, which model the history of the state of individual objects.

The configuration model describes the properties of the transaction management service required by the application or business model, respectively. The dashed arrow in Figure 2 is used to denote dependencies which result from associations between model elements of the configuration model and business logic model. The introduction of a separate configuration model has two reasons. First, the conceptual dissimilarity to the business logic model can be handled and second, multiple configuration metamodels can be designed. The conceptual dissimilarity to the business model results from the underlying abstract models that are used when declaratively configuring transaction logic. Whereas the business logic is described using concepts like action expressions [18], transactional logic is specified based, e.g., on transaction context propagation. Multiple configuration models are needed to provide tailored configuration capabilities for the different roles participating in the development process, to support the variety of transaction models respectively concepts, and to enable transaction design on different levels of abstraction.

The configuration model is transformed and merged with the business model. This step, which must be supported by according tools, is depicted by the thick arrows in Figure 2. The resulting integrated model provides the common conceptual basis for describing the semantics of transaction configurations, performing analysis, comparing configurations, and adjusting configurations to specific requirements. These requirements regard functional and non-functional properties. A functional requirement can be, for example, the absence of deadlocks. Non-functional properties can relate to issues of optimization. The integrated model is used by development tools exclusively. Adjustments to configurations are made visible to transaction designers by updating the configuration model accordingly. It is emphasized, that the integrated model provides an open solution, which enables to easily extend the set of configuration models. An important point regarding our work is the practical relevance of the proposed concepts. Therefore, possible analysis scenarios for the integrated model are investigated, includ-
ing basic compatibility checks of configurations, more thorough static analysis based on accordingly generated structures as demonstrated in [19], and model checking.

To conclude this section, it is noted that the proposed approach aligns quite well with the idea of the Model Driven Architecture (MDA) [20, 21]. MDA is based on the idea, that the software engineering process is based on models and driven by transformations of these models according to specific requirements. Our conceptual framework is based on metamodelling and important steps in the configuration process are realized by transformation and merging. Our work focuses on platform independent models (PIM) and corresponding transformations. To apply our models to current technologies, they have to be transformed to platform specific models (PSM) like EJB specific ones by corresponding tools.

5 Summary and Outlook

In this paper, we investigated the current situation regarding the declarative configuration of transaction management services in component-based technologies like EJB. Four problems have been identified, namely the lack of flexible and transparent configuration capabilities, the need for clarifying the responsibilities during configuration, the late configuration time, and the lack of concepts to support the component-based paradigm more adequately.

To overcome these problems, we proposed a model-based configuration approach for transaction management services that aligns with the idea of MDA [20]. The approach is based on a conceptual framework comprising a business model, multiple configuration models and an integrated model. The configuration models are transformed and merged with the business model into an integrated model, which serves as basis for analysis and transformation.

The contribution of this paper regards technological as well as methodological issues. A conceptual framework for the configuration of transaction management services has been proposed. This framework integrates with a transaction design method that regards multiple roles in the software engineering process, different transaction models, and several levels of abstraction.

We are currently elaborating such a transaction design method. A prototype to demonstrate the feasibility of our approach is in development. To provide an unambiguous description of the meaning of the integrated model, we are also working on adequate means for efficiently describing the semantics. First results are based on the method used in [18]. In the long term, we plan to integrate our concepts into experimental case tools and to map the described configurations to existing platforms.

Acknowledgement

The author would like to thank Steffen Zschaler for his valuable remarks and suggestions to this paper.
References

<table>
<thead>
<tr>
<th>Author Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Åkerholm, Mikael</td>
<td>194</td>
</tr>
<tr>
<td>Alda, Sascha</td>
<td>59</td>
</tr>
<tr>
<td>Ancilotti, Paolo</td>
<td>209</td>
</tr>
<tr>
<td>Armstrong, Rob</td>
<td>264</td>
</tr>
<tr>
<td>Becker, Steffen</td>
<td>68</td>
</tr>
<tr>
<td>Bernholdt, David E.</td>
<td>264</td>
</tr>
<tr>
<td>Bertolino, Antonia</td>
<td>233</td>
</tr>
<tr>
<td>Biyani, Karun N.</td>
<td>48</td>
</tr>
<tr>
<td>Bruneton, Eric</td>
<td>7</td>
</tr>
<tr>
<td>Carrington, David</td>
<td>272</td>
</tr>
<tr>
<td>Cervantes, Humberto</td>
<td>130</td>
</tr>
<tr>
<td>Chaudron, Michel</td>
<td>162</td>
</tr>
<tr>
<td>Choi, Si Won</td>
<td>40</td>
</tr>
<tr>
<td>Coupaye, Thierry</td>
<td>7</td>
</tr>
<tr>
<td>Cremers, Armin B.</td>
<td>59</td>
</tr>
<tr>
<td>Defour, Olivier</td>
<td>217</td>
</tr>
<tr>
<td>Elwasif, Wael R.</td>
<td>264</td>
</tr>
<tr>
<td>Eskenazi, Evgeni</td>
<td>280</td>
</tr>
<tr>
<td>Fioukov, Alexandre</td>
<td>280</td>
</tr>
<tr>
<td>Forghieri, Alessandro</td>
<td>84</td>
</tr>
<tr>
<td>Fredriksson, Johan</td>
<td>194</td>
</tr>
<tr>
<td>Fröberg, Joakim</td>
<td>146</td>
</tr>
<tr>
<td>Gai, Paolo</td>
<td>209</td>
</tr>
<tr>
<td>Guidi, Giacomo</td>
<td>209</td>
</tr>
<tr>
<td>Hall, Richard S.</td>
<td>130</td>
</tr>
<tr>
<td>Hammer, Dieter</td>
<td>280</td>
</tr>
<tr>
<td>Heineman, George T.</td>
<td>249</td>
</tr>
<tr>
<td>Hutchinson, John</td>
<td>114</td>
</tr>
<tr>
<td>Inverardi, Paola</td>
<td>84</td>
</tr>
<tr>
<td>Jézéquel, Jean-Marc</td>
<td>217</td>
</tr>
<tr>
<td>Jonkers, Hans</td>
<td>5</td>
</tr>
<tr>
<td>Kim, Soo Dong</td>
<td>40</td>
</tr>
<tr>
<td>Kotonya, Gerald</td>
<td>114</td>
</tr>
<tr>
<td>Kulkarni, Sandeep S.</td>
<td>48</td>
</tr>
<tr>
<td>Leclercq, Matthieu</td>
<td>7</td>
</tr>
<tr>
<td>Lipari, Giuseppe</td>
<td>209</td>
</tr>
<tr>
<td>Loecher, Sten</td>
<td>302</td>
</tr>
<tr>
<td>Loyall, Joseph</td>
<td>249</td>
</tr>
<tr>
<td>Malony, Allen D.</td>
<td>264</td>
</tr>
<tr>
<td>Maydl, Walter</td>
<td>178</td>
</tr>
<tr>
<td>McInnes, Lois C.</td>
<td>264</td>
</tr>
<tr>
<td>Medvidovic, Nenad</td>
<td>23</td>
</tr>
<tr>
<td>Meyerhöfer, Marcus</td>
<td>294</td>
</tr>
<tr>
<td>Mikic-Rakic, Marija</td>
<td>23</td>
</tr>
<tr>
<td>Min, Hyun Gi</td>
<td>40</td>
</tr>
<tr>
<td>Mirandola, Raffaella</td>
<td>233</td>
</tr>
<tr>
<td>Möller, Anders</td>
<td>146</td>
</tr>
<tr>
<td>Muskens, Johan</td>
<td>162</td>
</tr>
<tr>
<td>Neumann, Christoph</td>
<td>294</td>
</tr>
<tr>
<td>Nierstrasz, Oscar</td>
<td>1</td>
</tr>
<tr>
<td>Nolin, Mikael</td>
<td>146</td>
</tr>
<tr>
<td>Norris, Boyana</td>
<td>264</td>
</tr>
<tr>
<td>Overhage, Sven</td>
<td>68</td>
</tr>
<tr>
<td>Plouzeau, Noël</td>
<td>217</td>
</tr>
<tr>
<td>Presutti, Valentina</td>
<td>84</td>
</tr>
<tr>
<td>Quema, Vivien</td>
<td>7</td>
</tr>
<tr>
<td>Ray, Jaideep</td>
<td>264</td>
</tr>
<tr>
<td>Reussner, Ralf H.</td>
<td>68</td>
</tr>
<tr>
<td>Sandström, Kristian</td>
<td>194</td>
</tr>
<tr>
<td>Schantz, Richard</td>
<td>249</td>
</tr>
<tr>
<td>Sebastianis, Maurizio</td>
<td>84</td>
</tr>
<tr>
<td>Shende, Sameer</td>
<td>264</td>
</tr>
<tr>
<td>Shukla, Rakesh</td>
<td>272</td>
</tr>
<tr>
<td>Simons, Kevin</td>
<td>122</td>
</tr>
<tr>
<td>Soundarajan, Neelam</td>
<td>138</td>
</tr>
<tr>
<td>Stafford, Judith</td>
<td>122</td>
</tr>
<tr>
<td>Stefani, Jean-Bernard</td>
<td>7</td>
</tr>
<tr>
<td>Strooper, Paul</td>
<td>272</td>
</tr>
<tr>
<td>Taborda, Louis J.M.</td>
<td>100</td>
</tr>
<tr>
<td>Tivoli, Massimo</td>
<td>84</td>
</tr>
<tr>
<td>Trimarchi, Michael</td>
<td>209</td>
</tr>
<tr>
<td>Tyler, Benjamin</td>
<td>138</td>
</tr>
</tbody>
</table>