Michel R. V. Chaudron
Clemens Szyperski
Ralf Reussner (Eds.)

LNCS 5282

Component-Based Software Engineering

11th International Symposium, CBSE 2008
Karlsruhe, Germany, October 2008
Proceedings

Springer

Lecture Notes in Computer Science 5282

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Michel R. V. Chaudron Clemens Szyperski
Ralf Reussner (Eds.)

Component-Based Software Engineering

11th International Symposium, CBSE 2008
Karlsruhe, Germany, October 14-17, 2008
Proceedings

Springer

Volume Editors

Michel R.V. Chaudron
University Leiden, Faculty of Science
Leiden Institute of Advanced Computer Science
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: chaudron@liacs.nl

Clemens Szyperski
Microsoft
One Microsoft Way, Redmond, WA 98052, USA
E-mail: clemens.szyperski@microsoft.com

Ralf Reussner
Universität Karlsruhe (TH), Karlsruhe Institute of Technology (KIT)
Institut für Programmstrukturen und Datenorganisation
76128 Karlsruhe, Germany
E-mail: reussner@ipd.uka.de

Library of Congress Control Number: 2008936136

CR Subject Classification (1998): D.2, D.3, B.8, C.4, J.7

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-87890-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87890-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12533800 06/3180 5 4 3 2 1 0

Preface

On behalf of the Organizing Committee we are pleased to present the proceedings of the 2008 Symposium on Component-Based Software Engineering (CBSE). CBSE is concerned with the development of software-intensive systems from independently developed software-building blocks (components), the development of components, and system maintenance and improvement by means of component replacement and customization. CBSE 2008 was the 11th in a series of events that promote a science and technology foundation for achieving predictable quality in software systems through the use of software component technology and its associated software engineering practices.

We were fortunate to have a dedicated Program Committee comprising many internationally recognized researchers and industrial practitioners. We would like to thank the members of the Program Committee and associated reviewers for their contribution in making this conference a success. We received 70 submissions and each paper was reviewed by at least three Program Committee members (four for papers with an author on the Program Committee). The entire reviewing process was supported by the Conference Management Toolkit provided by Microsoft. In total, 20 submissions were accepted as full papers and 3 submissions were accepted as short papers.

This time CBSE was held as a colocated event of COMPARCH, together with the 4th International Conference on the Quality of Software-Architectures (QoSA 2008) and two workshops: Workshop on Component-Based High-Performance Computing and Workshop on Component-Oriented Programming. This year's event was organized in Karlsruhe for which our special thanks are due to Ralf Reussner and his crew. We also wish to thank the ACM Special Interest Group on Software Engineering (SIGSOFT) for their sponsorship. The proceedings you now hold were published by Springer and we are grateful for their support. Finally, we thank the many authors who contributed the high-quality papers contained within these proceedings. The CBSE organizers also like to thank the supporters of COMPARCH 2008, namely 1&1 Internet AG and sd&m AG.

As the international community of CBSE researchers and practitioners continues to prosper, we expect the CBSE symposium series to similarly attract widespread interest and participation.

August 2008

Michel R.V. Chaudron
Clemens Szyperski

Organization

Conference Chairs

Michel R.V. Chaudron	Leiden University, The Netherlands
Clemens Szyperski	Microsoft, USA

Steering Committee

Ivica Crnkovic	Mälardalen University, Sweden
Ian Gorton	Pacific North West National Laboratory, USA
George Heineman	Worcester Polytechnic Institute, USA
Heinz Schmidt	RMIT University, Australia
Judith Stafford	Tufts University, USA
Clemens Szyperski	Microsoft, USA

Program Committee

Uwe Aßmann	Dresden University of Technology, Dresden, Germany
Mike Barnett	Microsoft Research, USA
Antonia Bertolino	CNR Research, Pisa, Italy
Judith Bishop	University of Pretoria, Pretoria, South Africa
Ivica Crnkovic	Mälardalen University, Vasteras, Sweden
Dimitra Giannakopoulou	RIACS/NASA Ames, Moffet Field CA, USA
Ian Gorton	Pacific North West National Laboratory, Richland WA, USA
Lars Grunske	University of Queensland, Brisbane, Australia
Richard Hall	LSR-IMAG, Grenoble, France
Dick Hamlet	Portland State University, Portland OR, USA
George Heineman	Worcester Polytechnic Institute, Worcester MA, USA
Jean-Marc Jézéquel	IRISA (INRIA & Univ. Rennes 1), Rennes, France
Bengt Jonsson	Uppsala University, Uppsala, Sweden
Joe Kiniry	University College Dublin, Ireland
Gerald Kotonya	Lancaster University, Lancaster, UK
Magnus Larsson	ABB Corporate Research, Vasteras, Sweden
Kung-Kiu Lau	University of Manchester, Manchester, UK
Raphael Marvie	University of Lille, Lille, France
Michael Maximilien	IBM Almaden Research Center, San Jose CA, USA

Nenad Medvidovic	University of Southern California, Los Angeles CA, USA
Henry Muccini	University of L'Aquila, L'Aquila, Italy
Rob van Ommering	Philips Research Labs, Eindhoven, The Netherlands
Ralf Reussner	University Karlsruhe, Karlsruhe, Germany
Alessandra Russo	Imperial College, London, UK
Christian Salzmann	BMW Car IT, Munich, Germany
Douglas Schmidt	Vanderbilt University, Nashville TN, USA
Heinz Schmidt	RMIT University, Australia
Jean-Guy Schneider	Swinburne University of Technology, Melbourne, Australia
Judith Stafford	Tufts University, USA
Asuman Sünbül	SAP Research, Palo Alto CA, USA
Clemens Szyperski	Microsoft, USA
Massimo Tivoli	University of L'Aquila, L'Aquila, Italy
Wolfgang Weck	Independent Software Architect, Zürich, Switzerland
Dave Wile	Teknowledge Corp., Los Angeles CA, USA

Major Supporters

1&1 Internet AG, Karlsruhe, Germany
sd&m AG, Munich, Germany

Table of Contents

Performance Engineering

Extra-Functional Properties: Security and Energy

Formal Methods and Model Checking

Verification Techniques

Run-Time Infrastructures

Methods of Design and Development

Component Models

Automating Performance Analysis from Taverna Workflows

Rafael Tolosana-Calasanz[1], Omer F. Rana[2], and José A. Bañares[1]

[1] Instituto de Investigación en Ingeniería de Aragón (I3A)
Department of Computer Science and Systems Engineering
University of Zaragoza, Spain
rafaelt@unizar.es, banares@unizar.es
[2] School of Computer Science, Cardiff University, UK
o.f.rana@cs.cardiff.ac.uk

Abstract. Workflow systems provide support for combining components to achieve a particular outcome. Various approaches from software engineering have been utilized within such systems, such as the use of design patterns to support composition, and the use of a software engineering lifecycle to support workflow construction and execution. As components used within a workflow may be implemented by third parties, it is often necessary to be able to determine the impact a particular component composition will have on the overall execution of a workflow. A method for predicting the execution time of a given workflow is proposed. First, the method obtains a model from a given workflow in an automated way. The model obtained is a Reference net – a specific type of Petri net. Features of Reference nets can subsequently be exploited, such as the possibility of building hierarchical workflow models which can facilitate the modelling process. The Reference nets are extended so that each task in the model is parameterised with a time value, representing the execution time of the task. We propose several timing profiles: those obtained from real measurement of the workflow system, from stochastic and constant values which allow us to test the model behaviour under specific situations.

Keywords: Workflow Performance Models, Petri Nets.

1 Introduction

During the last decade, workflow systems have been widely used in the business and scientific communities, primarily for composing applications from third party components. Identifying the execution time of a workflow, particularly when such third party components are used, is often an important requirement. As scientific workflows may involve long running tasks, characterisation of execution time is often used by a designer to determine the suitability of particular components. Hence, identifying bounds on execution time of a workflow is an important requirement for supporting component composition within such systems.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 1–15, 2008.

The execution time of workflows can be estimated by constructing a performance model of the workflow. Many similarities between workflows and component-based systems can be established: component-based systems can be seen as complex systems which are built by assembling basic components, similar in fact to existing workflow systems. A key difference, especially in scientific workflows, is that some of the tasks are likely to be executed in parallel, whereas prediction techniques covered by the Component-Based Software Engineering (CBSE) do not consider parallelism.

Petri nets have been widely used for modelling complex concurrent systems, for specifying workflows [1] and even as a development tool for implementing workflow systems [2]. They provide clear and precise formal semantics and an intuitive graphical notation. Additionally, many different types of analysis can be accomplished on a Petri net model, including reachability, deadlock or liveness analysis. We make use of Reference nets, a special type of High-level Petri net, along with the Renew tool [3], which can interpret them. Reference nets and Renew overcome the main limitation of other High-level Petri nets, such as their static nature, and also support hierarchical and object-oriented modelling. This hierarchical modelling enables groups of components to be combined for analysis. This can be particularly useful when modelling workflows that involve a large number of components. Furthermore, the Renew tool supports simulations of timed Reference nets, that is, Reference nets with timing extensions. Thus, once a workflow model is obtained and properly parameterised, its overall execution time can be estimated. We propose three timing profiles for use with timed Reference nets: constant values, probability distributions (including stochastic values) and values obtained from instrumenting the real workflow system.

A method for predicting the execution time of a given workflow is proposed. First, the method obtains a timed Reference net model from a given workflow in an automated way and then this model is simulated with a timed profile. We focus on the use of the Taverna workflow system – which shares many similarities with other scientific workflow systems such as Kepler and Triana. It is important to highlight that, even though Taverna is widely used within the scientific workflow community, in particular in bioinformatics, general purpose workflows – which can be composed of third party components – can also be modelled with the system. In this paper, a particular scientific workflow has been used to illustrate the process of automatically deriving a performance model from the workflow description. Subsequent analysis of this single workflow is then used to explain why such a model is useful.

This paper is organised as follows, in Section 2 a brief background knowledge about Petri nets and Taverna is given. In Section 3 related work is reviewed briefly. Section 4 shows how to obtain timed Reference net models from workflows in an automated way and, as an example, a model from a currently used Taverna workflow is obtained. In Section 5, performance figures are provided for illustrating the technique with an example. Finally, the conclusions and future work are provided.

2 Background

An ordinary Petri net can be defined informally as a bipartite directed graph which consists of places, transitions, arcs and tokens (see [4] for a formal and more extensive definition). There are many extensions to ordinary Petri nets such as High-level Petri nets or timed Petri nets which provide higher levels of abstraction and improve the modelling potential of ordinary Petri nets. We use a specific type of High-level Petri net, Renew's Reference nets. Renew [3] is a Reference net interpreter and a Reference net graphical modelling tool which is also used in this work for the simulation of our Reference net based models. Renew's Reference nets are a special subtype of Net-within-Nets [5]. The characteristics and behaviour of ordinary Petri nets are also present in Reference nets, but they provide many other useful extensions. The most distinctive ones are:

- Reference net tokens can also be a Java object or another Reference net, and all of these nets can communicate with each other by means of synchronous channels. A synchronous channel can be established only between two nets and it allows a net to send (invoke) a message to other net in a synchronous way.
- Reference net transitions can be equipped with a variety of inscriptions such as expressions, actions or guards. Expression and action inscriptions are ordinary expressions evaluated while the net simulator searches for a binding of the transition, the result of this evaluation can be applied for influencing the binding of variables that are used elsewhere. Guard inscriptions are expressions that are prefixed with the reserved word `guard`, so that a transition can only be fired when all of its guard inscriptions evaluate to true.
- Reference net arcs can also have inscriptions. By default, an arc without inscription transports a black token (the token of ordinary Petri nets), but when arcs have variables as inscriptions, they transport Java objects or other nets.
- Renew's Reference nets are dynamic, and in this way differ from other High-level Petri net approaches. This dynamism is achieved by means of the `new` construct that can be part of an expression inscription at a transition, as a result a new instance of an object net can be created at execution time. This feature totally overcomes the static nature of other high-level Petri nets. Renew's Reference net token domain is formed by the set of all possible tokens, including any Java variables or objects and any Reference net instances.

Renew supports timed Reference nets, where the time concept is based on introducing a global clock used to model time. Then, a time stamp is attached to each token denoting the time when it becomes available. This time stamp may be modified by time inscriptions annotated in the arcs. Thus, time inscriptions in output arcs specify that a token is only available after some time. The output arc delay may be a constant value, a calculated or random variable. The output arc delay cannot influence the enabling of a transition, but only the time stamps of the generated tokens. A delay is added to an arc by adding to the arc inscription

the symbol and an expression that evaluates to the number of time units, e.g. let `t` be a positive integer and `x+1` an integer token value, the output arc inscription `x+1@t` indicates that the token value `x+1` has to be moved after `t` time units. Time inscriptions can also be added to input arcs, these kind of inscriptions require that a token remains available for a given time before enabling the transition.

The Taverna workbench is a tool for editing Simple Conceptual Unified Flow Lanaguage (SCUFL) scripts, which define workflow as a network of processors and links, and vizualizing their enactment by a built-in workflow enactment engine. The SCUFL language is primarily aimed at users who currently use web forms or scripting languages to interact with web resources. The Taverna workbench is extensively used in the bioinformatics community, and has also been integrated within the `myExperiment.org` portal.

3 Related Work

Extensions to ordinary Petri nets (High-level Petri nets) are commonly used for implementing workflows. A specific type of High-level Petri nets, the Hierarchical Petri nets, has been applied [6] for expressing hierarchical workflows in Grid environments within the *KW-f project*. One of the most important advantages of this approach is that the design models obtained can be simplified by means of composite transitions, representing subworkflows. However, the derived Hierarchical Petri nets are static, and cannot change once they have been specified. Other approaches propose QoS workflow models [7] for predicting time, cost and reliability. Nonetheless, this proposal may require modifying workflow systems for supporting QoS management and computing the QoS metrics.

In the Performance Engineering community, traditionally, three methods have been proposed, sometimes in complementary ways, to reveal how a system performs: direct measurement, simulation and analytical techniques. Although all of them allow system engineers to undertake testing before development, and at a lower cost, both simulation and analytical methods share the goal of creating a performance model of the system/device that accurately describes its load and duration of the activities involved. Performance models are often described in some formalisms including queuing network models [8], stochastic process algebra [9] or stochastic Petri nets [10] that provide the required analysis and simulation capabilities. A great number of these studies try to derive Petri net performance models from UML diagrams [11,10] and to compute performance metrics such as response time.

One aim of the CBSE field is to enable the prediction of *extra-functional* system properties such as performance and reliability. There exist several approaches which estimate the performance of a component-based system. In [12], the performance of a system is estimated by specifying the internal components in a parametric way and dividing the model creation among the developer roles. In consequence, the specification process requires a deep knowledge of each component and no empirical and real measurement of the system is needed. In contrast, another interesting approach [13] considers systems' components as black

boxes where no internal knowledge of them is known. Therefore, each system component requires an intensive real measurement of its performance.

On the other hand, the Business Process Execution Language for Web Services (BPEL [1]), has emerged as the *de facto* standard for workflow implementation for business processes. The use of BPEL for representing scientific workflows is still limited. BPEL can be translated into Petri nets as pointed out in [14].

4 Automated Model Construction

Most workflow languages can be described formally by context-free grammars and, in consequence, workflows may be parsed and translated into other languages automatically. In particular, they could be translated into Petri nets, because the formalism of Petri nets, despite some limitations [15], can express most workflow operations or patterns properly. In this work, this fact is exploited in order to obtain a Petri net based model from a given workflow. Figure 1 shows our automated model construction, first there is a parser, a process that receives a workflow as an input, analyses its structure and builds a parse tree as an output by following well-known parsing techniques [16]. Then, the model constructor process takes that parse tree and a set of transformation rules (mappings from the workflow language patterns to the Petri net model patterns) and builds a timed Reference net model. In some cases, the structure of that obtained Petri net model can be analysed so that deadlocks could be searched in it by using some of the existing techniques at Petri net theory such as reachability graph analysis and tools such as GreatSPN. It is important to note that both the parser and the model constructor are not part of Renew, in fact, they are completely different systems. Renew is used in this paper for simulating the obtained model as it will be shown in Section 5.

One of the advantages of our approach is that Petri net models provide a visual representation of the aspects related to performance (response time, in particular) of the real system, that is, a simplified and graphical view of the original workflow: elements of the workflow that do not affect the execution time are deliberately suppressed in the model.

In order to obtain models automatically from Taverna workflows, specific parsing and model constructor components should be developed as well as the transformation rules (the Taverna workflow language has been formally defined in [17]). Here, a prototype parser and model constructor were developed. The parser takes a Taverna workflow and builds a parse tree. The parse tree along with the transformation rules are used by the model constructor for automatically derive a Petri net based performance model.

Essentially, a Taverna workflow consists of a collection of processors with data and control links among them. A data link establishes a dependency between the output of a processor and the input of another one, whereas a control link between two tasks indicates that a processor can only begin its execution after some other processor has successfully completed its execution. Processors are

[1] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

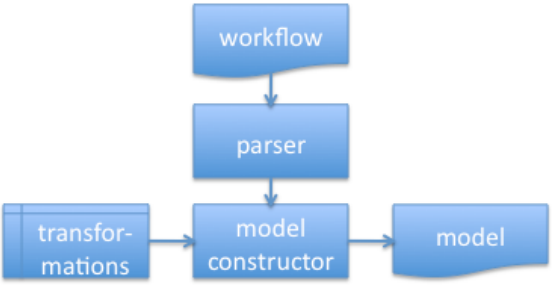


Fig. 1. Automated Model Construction Process

implemented either as local Java classes, or as Web Services. The system is also provided with specific Java-based processors which allow the user to express the choice pattern. Another interesting feature of Taverna's workflow language is its implicit iterations. Thus, when two processors are interconnected by a data link, the output of a processor is linked to the input of the other. In case the input is expected to be single, but the output happens to be multiple, for each component in the multiple output, Taverna invokes the latter processor once.

Therefore, the transformation rules from Taverna's workflow language to our timed Reference net based models are:

- A Taverna processor can be modelled by a Petri net transition and its input and output data by the corresponding input and output Petri-net places (Figure 2a). In contrast, a Java-based processors implementing choice should be modelled by the Petri net choice pattern shown in Figure 2c. Other elements such as Taverna processors implementing constant values, such as the String-type constant processors, do not have to appear in the model as they only provide constant data in constant time, and it does not determine the performance of the original workflow.
- A data dependency between two tasks in Taverna can be modelled by connecting two transitions with a place and its corresponding directed arcs as in Figure 2b. The place between the transitions models the data being passed between the two tasks.
- A control dependency between two tasks can be modelled in the same way by connecting two transitions with a place and its corresponding directed arcs as shown in Figure 2b. In contrast, in this case, the place models just the control dependency.
- Implicit iterations could be modelled explicitly, but there is no real advantage in doing this, and the original workflow structure would be distorted. Therefore, we prefer to conserve the same structure and not to model them explicitly.

Later, in Section 5, the model will be parameterised properly for estimating the execution time of implicit iterations, as well as the execution time of choices and tasks.

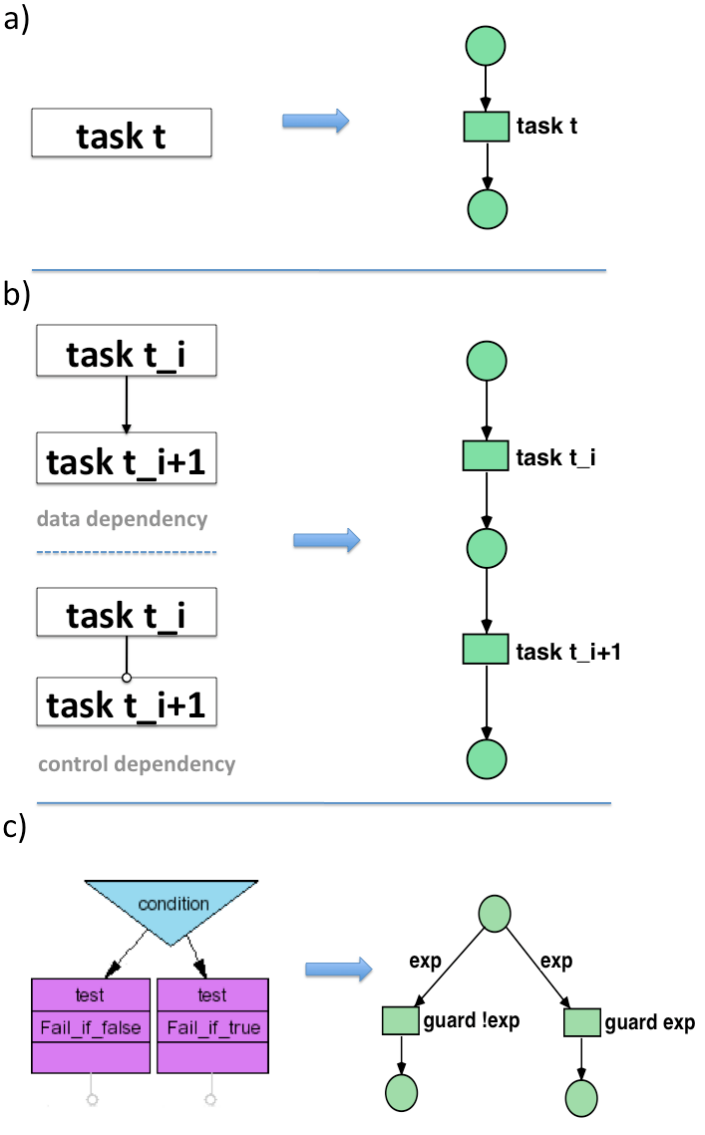


Fig. 2. Transformations of Taverna's workflow language elements into High-level Petri net elements

4.1 Pathway to PubMed Workflow Example

Taverna's *Pathway to PubMed* workflow– taken from *myexperiment* [2] – represents a real workflow example. It takes a lists of terms or words as an input, builds a query and retrieves publication information from the *PubMed* database.

[2] http://www.myexperiment.org

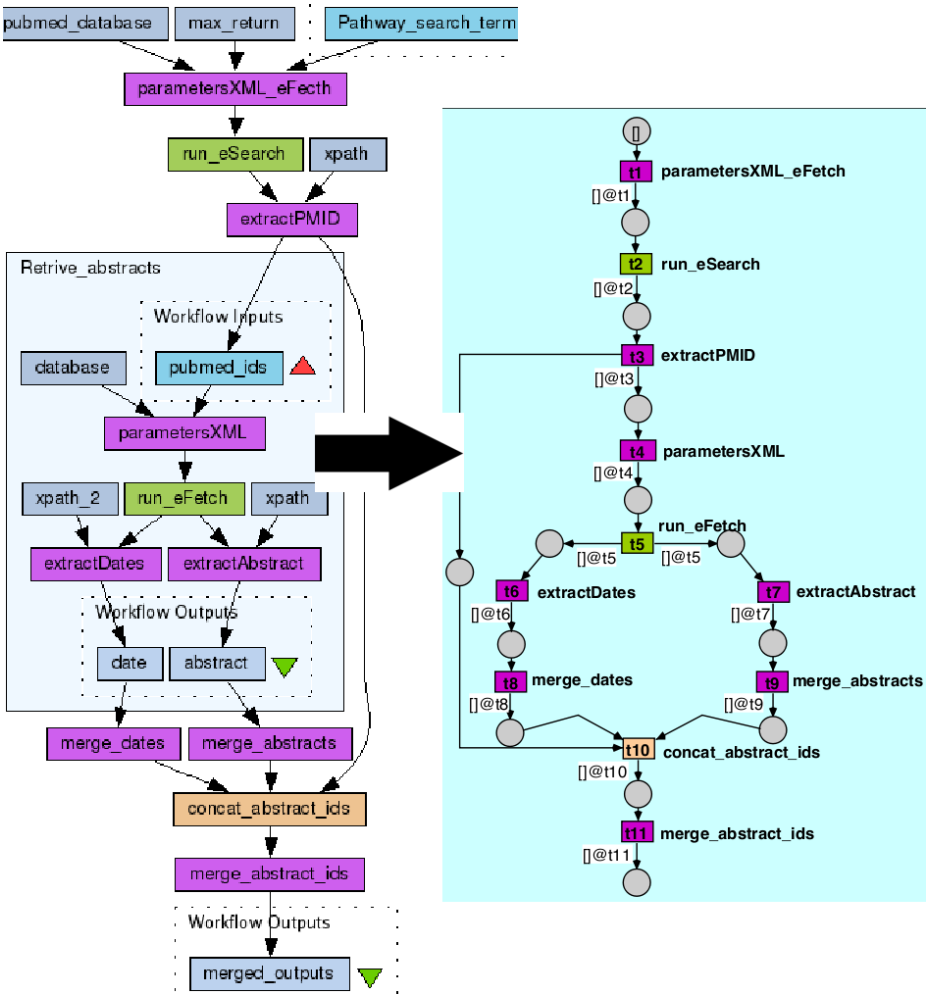


Fig. 3. Taverna's *Pathway to PubMed* workflow graphical representation, obtained from *myexperiment.org* and a timed Reference net model obtained from it automatically

Figure 3 shows a graphical representation of the workflow and a timed Reference net model obtained from it automatically. Nevertheless, the potential capability of Renew's Reference nets can be further exploited for modelling the internal structure of a workflow with varying degrees of granularity, obtaining hierarchical workflow models. In Figure 4, a hierarchical model for the *Pathway to PubMed* workflow is depicted. The workflow model can be seen as a sequence of three main parts: the initial part, the part corresponding to *Retrieve_abstracts* and the final part. The model expressing that idea is depicted in Figure 4. Each phase is modelled with the same mechanism which consists of two transitions and a place that holds the corresponding subtasks (a subworkflow) at each phase. More

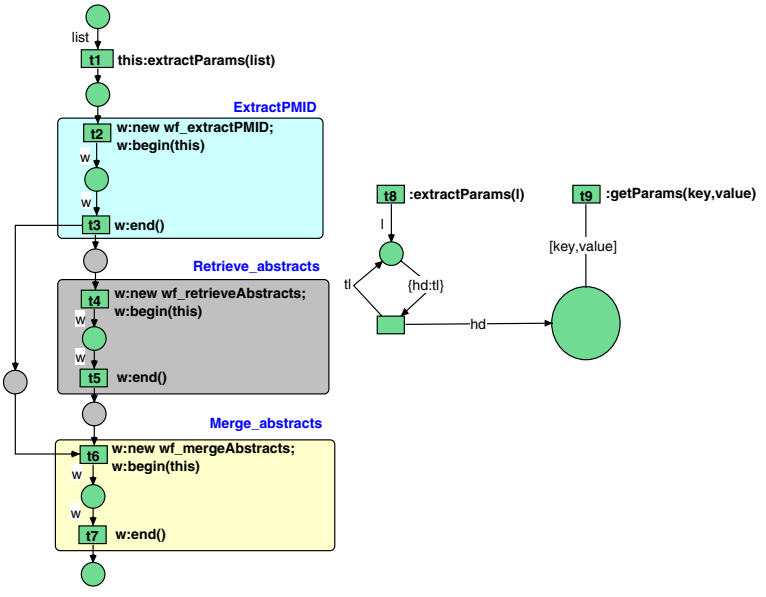


Fig. 4. Hierarchical Reference net model of Taverna's *Pathway to PubMed* workflow

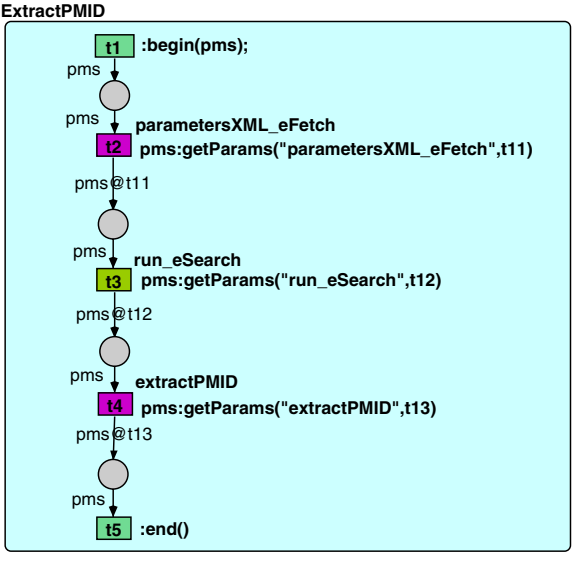


Fig. 5. Reference net model of the initial part (initial subworkflow) of Taverna's *Pathway to PubMed* workflow

specifically, when Transition t2 is fired, a new object net modelling the initial part of the workflow is created (action w:new wf_extractPMID) and its execution started (by invoking the synchronous channel w:begin(this)). Then, only when the subworkflow *wf_extractPMID* finishes, Transition t3 is fired. After that, the *Retrieve_abstracts* middle phase can start its execution. It is important to note that the arcs and place between Transition t3 and Transition t6 represents the data dependency between tasks *extractPMID* and *concat_abstract_ids* at the original workflow. Transitions t8 and t9 and its synchronous channels in the figure, related to the simulation aspects of the model, will be described in Section 5.

In consequence, between Transitions t2 and t3, the tasks corresponding to the initial phase are enacted. The *extractPMID* group of tasks is modelled by the object net depicted in Figure 5. The execution of this net is started at its initial Transition t1 (when invoked through the synchronous channel begin). The execution consists of three tasks, parameterXML_eFetch, run_eSearch and extractPMID, that are enacted one after another and connected by data dependencies. These tasks are modelled by Transitions t2, t3 and t4, respectively, and their data dependencies by the corresponding arcs and places. The subworkflow ends at Transition t5 which synchronises with Transition t3 of the system net model (Figure 4). The arc inscriptions are related to the simulation aspects that will be further explained in Section 5.

5 Performance Analysis from Model

In Section 4.1, a timed Reference net model of Taverna's *Pathway to PubMed* workflow was obtained. In this section, the model is fed with a timing profile and then simulated in Renew and a performance estimation is obtained. Here the timing aspects of the model are explained – which are key to deriving the performance model. Each output arc in Figure 5 has a parameterized time inscription. These parameters are obtained from the synchronous channel getParams at simulation time. The mechanism works as follows: the system net at Figure 4 obtains a token as the simulation input. This token consists of a list of time parameters. These parameters are sent from Transition t1 to Transition t8 which is going to store them throughout the simulation process (it is important to note that Transition t1 and t8 communicate via the synchronous channel extractParams). Each time parameter is a pair (key, value) where key represents the name of the task and value represents its associated parameterized delay, which represents an estimate of the execution time of the task. Then, the synchronous channel getParams of Transition t9 at the system net is invoked by each transition modelling a task in the subworkflows (object nets).

After extracting the parameters in the system net, the simulation will continue at the subworkflow *ExtractPMID* (Figure 5). In this subworkflow, at Transition t1, the time stamp associated with the token is 0. After the simulation of task parametersXMLFetch at Transition t2, it has a value of t11. After run_eSearch, its value is t11 + t12 and at the end of the simulation of this subworkflow, and at the beginning of the simulation of the next subworkflow (*Retrieve_abstracts*), the time stamp is t11 + t12 + t13.

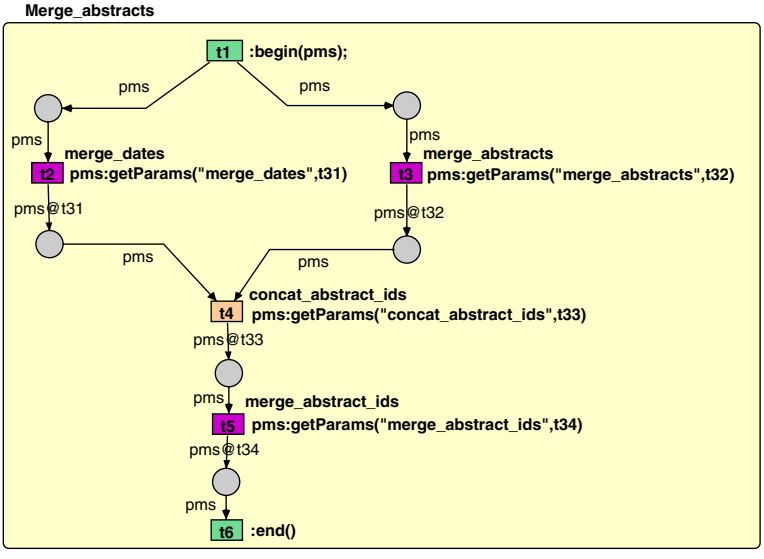


Fig. 6. Reference net model corresponding to the final part (or subworkflow) of *PubMed*

In case a task has several inputs coming from different and parallel execution flows, the task cannot be executed until all its input data are available. In our proposal, this fact can be easily modelled by the semantics of timed Reference nets and its output arc inscriptions. Consider the subworkflow *Merge_Abstracts* of Figure 6. It has two tasks called `merge_dates` and `merge_abstracts` – modelled by Transitions `t2` and `t3`, respectively – and their outputs go into task `concat_abstract_ids` – modelled by Transition `t4`. Then, Transition `t4` receives two tokens but it will not be enabled until both tokens will be available and this will happen when the token with the highest associated delay is available.

5.1 Simulations with Different Timing Profiles

The result of the overall execution time prediction relies on the value of each time parameter provided at the beginning of a simulation. Different timing profiles can be introduced in our model: (i) constant time values, (ii) probability distributions and (iii) values obtained from real measurement of the workflow system. Combination of these are also possible.

The constant time value profile could be useful when software engineers have a particular interest in exploring certain behaviours (i.e. for identifying possible critical paths and potential bottlenecks) of the workflow at specific circumstances. The probability distribution based time value profile may be required when there is little knowledge about the performance of the tasks, the result in this case will also depend on the probability distribution chosen and its characterisation. The third profile is based on real measurement by instrumenting the workflow system. Some workflow systems such as Taverna provide real

performance data of each of its tasks when enacting a workflow. Nonetheless, performance data could also be obtained by manually instrumenting the workflow system. This can be achieved by providing a wrapper for each component in the workflow. The wrapper measures execution time of the component (by using a timer to measure the time interval between the received execution request, and the subsequent result that is generated once execution completes). However, such an approach leads to additional performance overheads, and would necessitate re-writing of the original workflow. These estimated times may consider factors such as different input data, component failures, network delays or operating system and hardware architecture aspects.

5.2 Performance Prediction at the PubMed Workflow

Our *PubMed* workflow model was fed with a timing profile based on real measurement, provided by execution of the workflow in Taverna. Even though Taverna can provide some real performance data, it does not track the execution time of the tasks forming part of subworkflows and provides only the subworkflow overall execution time. Thus, as no real measurements of the tasks of subworkflow *Retrieval_abstracts* can be obtained directly from the Taverna enactor, that subworkflow was treated in the model as a black box, whereas the initial and final parts are treated as white boxes, in other words, the hierarchical model will simulate the execution of all the tasks of the initial and final subworkflows but will consider *Retrieval_abstracts* as a single execution unit or task.

The *PubMed* workflow takes the input terms and builds a query. This query consists of a parameter `max_return` that limits the maximum number of *PubMed* records to retrieve, by default up to 500. Our experiments were accomplished

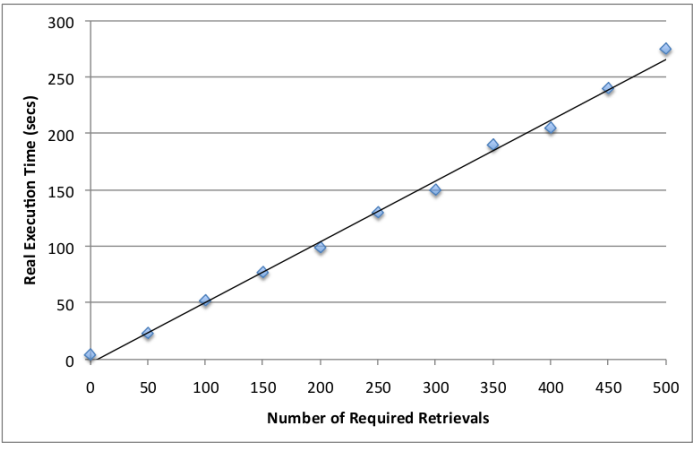


Fig. 7. Correlation between workflow data input and `Retrieve_abstracts` execution time

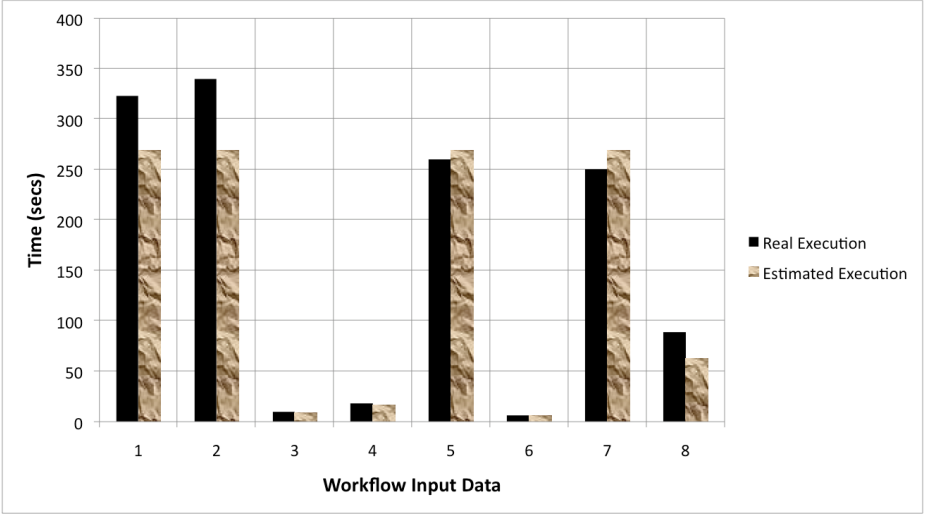


Fig. 8. Comparison between 8 real workflow execution times with 8 different inputs (queries in the end) and their corresponding estimated workflow execution times at PubMed

on the *PubMed* workflow with the same input (the term *cancer* that appears in many documents), but changing `max_return` values in the range from 0 to 500. For each value of `max_return`, execution times were obtained for each of the tasks. It was found that the execution time for *Retrieve_abstracts* dominated the other parts of the workflow, and was the key task influencing the overall workflow execution time. Additionally, there is a linear correlation between the `max_return` value and the execution times of tasks as shown in Figure 7.

In consequence, given an input data, this linear correlation can be used for obtaining the execution time of *Retrieve_abstracts* and the execution times of the rest of the tasks. In order to know the number of records that the input data is going to retrieve from *PubMed*, there is a *PubMed* component service that can calculate it at a low execution time. Then, once the timing profile is obtained, the model can be fed with it and the overall workflow performance can be estimated. It should be noted that in the workflow there are four implicit iterations and they are estimated by means of the linear correlation found. In Figure 8 some real executions of *PubMed* workflow are compared to their corresponding estimations obtained from the model. Input data 1, 2, 5 and 7 retrieve 500 *PubMed* records, whereas input data 3, 4, 6 and 8 retrieve 16, 31, 11 and 117 records, respectively.

6 Conclusions

A method for predicting the execution time of a given workflow has been proposed. A performance model from a given workflow is derived in an automated

way, focusing on the Taverna workflow system. The model obtained is a Reference net, a specific type of Petri net. The Reference nets are extended with time, so that each task in the model is parameterised with a time value, representing the execution time of the task. This model can be interpreted by the Renew tool and the workflow execution time can be estimated. As this estimation relies mainly on the time parameters of the tasks, we propose several timing profiles: profiles obtained from real measurement of the workflow system, stochastic values and constant values which in addition to performance prediction allow us to test the model behaviour under specific situations. The process is illustrated with a real workflow of Taverna: *Pathway to PubMed*, and the results from the model are compared with actual executions through the Taverna system.

A particular workflow is described to concretize the approach, which can be generalised to other systems. Workflow descriptions consist of (i) sequence operations connecting ordered series of tasks; (ii) parallel operations connecting tasks that will be executed concurrently; (iii) choice operations to select one execution flow among the alternative ones according to an evaluated condition; (iv) iteration constructs supporting the repetition of a subworkflow as long as an associated condition is true. The formalism of Petri nets, despite some limitations [15], can model most of these operations, subsequently the obtained models can be used in order to analyse and study the workflow performance.

As a future work, some kind of heuristics could be developed for simplifying the calculation of the execution times of components at the system and it could be based for instance on certain aspects such as the input data size.

Acknowledgements

The authors wish to thank Prof. Ezpeleta of the University of Zaragoza and the anonymous reviewers for their suggestions and comments. Besides, this work has been supported by the research project TIN2006-13301, granted by the Spanish Ministry of Education and Science. Rafael Tolosana's work has been supported by "Programa Europa XXI de Estancias de Investigación" reference number IT 1/08, granted by DGA (CONAID) and CAI.

References

1. Russell, N., ter Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPM Center Report BPM-06 22, BPMcenter.org (2006)
2. Tolosana-Calasanz, R., Bañares, J.A., Álvarez, P., Ezpeleta, J.: Vega: a service-oriented grid workflow management system. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part II. LNCS, vol. 4804. Springer, Heidelberg (2007)
3. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for petri nets: Renew. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)

4. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of IEEE 77, 541–580 (1989)
5. Valk, R.: Petri nets as token objects - an introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)
6. Alt, M., Hoheisel, A.: Petri Nets. In: Workflows for e-Science, pp. 190–207. Springer, Heidelberg (2007)
7. Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. Web Semantics: Science, Services and Agents on the World Wide Web 1(3), 281–308 (2004)
8. Menasce, D.A., Gomaa, H.: A method for design and performance modeling of client/server systems. IEEE Transactions on Software Engineering 26(11), 1066–1085 (2000)
9. Bernardo, M., Ciancarini, P., Donatiello, L.: Aempa: a process algebraic description language for the performance analysis of software architectures. In: WOSP 2000: Proceedings of the 2nd international workshop on Software and performance, pp. 1–11. ACM Press, New York (2000)
10. Bernardi, S., Merseguer, J.: Performance evaluation of uml design with stochastic well-formed nets. J. Syst. Softw. 80(11), 1843–1865 (2007)
11. Hu, Z., Shatz, S.M.: Mapping uml diagrams to a petri net notation for system simulation. In: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004), pp. 213–219 (2004)
12. Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: WOSP 2007: Proceedings of the 6th international workshop on Software and performance, pp. 54–65. ACM, New York (2007)
13. Hamlet, D.: Software component composition: a subdomain-based testing-theory foundation. Softw. Test. Verif. Reliab. 17(4), 243–269 (2007)
14. Aalst, W.M.P., Lassen, K.B.: Translating workflow nets to bpel. Technical report. In: BETA Working Paper Series, WP145, Eindhoven University of Technology (2005)
15. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In: Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, Technical Report DAIMI PB-560, pp. 1–20 (2002)
16. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compiler: Principles, Techniques and Tools, 2nd edn. Pearson Education, London (2007)
17. Turi, D., Missier, P., Goble, C., Roure, D.D., Oinn, T.: Taverna workflows: Syntax and semantics. In: E-SCIENCE 2007: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing, Washington, DC, USA, pp. 441–448. IEEE Computer Society, Los Alamitos (2007)

An Empirical Investigation of the Effort of Creating Reusable, Component-Based Models for Performance Prediction

Anne Martens[1], Steffen Becker[2], Heiko Koziolek[3], and Ralf Reussner[1]

[1]Chair for Software Design and Quality
Am Fasanengarten 5, University of Karlsruhe (TH), 76131 Karlsruhe, Germany
[2]FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
[3]ABB Corporate Research, Wallstadter Str. 59, 68526 Ladenburg, Germany
{martens,sbecker,koziolek,reussner}@ipd.uka.de

Abstract. Model-based performance prediction methods aim at evaluating the expected response time, throughput, and resource utilisation of a software system at design time, before implementation. Existing performance prediction methods use monolithic, throw-away prediction models or component-based, reusable prediction models. While it is intuitively clear that the development of reusable models requires more effort, the actual higher amount of effort has not been quantified or analysed systematically yet. To study the effort, we conducted a controlled experiment with 19 computer science students who predicted the performance of two example systems applying an established, monolithic method (Software Performance Engineering) as well as our own component-based method (Palladio). The results show that the effort of model creation with Palladio is approximately 1.25 times higher than with SPE in our experimental setting, with the resulting models having comparable prediction accuracy. Therefore, in some cases, the creation of reusable prediction models can already be justified, if they are reused at least once.

Keywords: Performance Prediction, Empirical Study, Controlled Experiment.

1 Introduction

As current applications always ask for maximum performance, performance problems are continuously prevalent in many software systems [20]. Model-based prediction methods [1] try to tackle these problems during early design phases to avoid the problem of implementing architectures which are not able to fulfil certain performance goals. They counter the still popular ”fix-it-later” attitude towards performance problems. Many of these methods use designer-friendly UML-based models for software developers, and transform them into formal models (e.g., queueing networks, stochastic Petri-nets, stochastic process algebras), from which performance measures (e.g., response times, throughput) can be derived analytically or via simulation.

During the last decade, researchers have proposed several monolithic prediction approaches (such as SPE [20], uml2LQN [15], umlPSI [2], survey in [1]) and several

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 16–31, 2008.

component-based (CB) prediction approaches (such as CB-SPE [7], ROBOCOP [8], and Palladio [6], survey in [5]). CB approaches try to leverage the benefits of componentry in the sense of Szyperski [21] by reusing well-documented component specifications. This is of particular interest for performance prediction methods, as CB software designs limit the degree of freedom for implementation by (at least partially) reusing existing components. This can also lead to higher performance prediction accuracy. In addition, reusable component prediction models can be composed isomorphically to the software architecture, thereby lowering the effort of performance modelling.

Palladio features highly parametrised component performance specifications, which are better suited for reuse than those of other approaches, because they include more context dependencies (i.e., dependencies to external service calls, usage profile, resource environment). The effort for creating such parametrised, CB models is naturally higher than for throw-away models. However, until now this higher effort has not been investigated systematically. Therefore, it is an open question when it is justified.

Based on this observation, we conducted a controlled experiment comparing the effort of applying SPE (as an example for a method with throw-away models) and Palladio (as an example for a method with reusable models). In this paper, we present the results for the following question: (Q1) What is the duration of modelling and predicting with both methods? As we wanted to assess the effort of applying the methods without bias, we let 19 computer science students apply the methods in an experimental setting. They analysed two CB software systems and assessed the performance impact of additional five design alternatives (e.g., introducing caches, replication, etc.). By using tools accompanying the methods (SPE-ED and PCM-Bench), they predicted response times for two different usage profiles. Therefore we assessed the effort for the combination of applying the method and the corresponding tools.

Our results show that modelling the whole task (that is the initial system and five additional design alternatives) took in average 1.25 times longer with Palladio than with SPE. Interestingly, modelling only the initial architecture took in average 1.81 times longer. The students spent most of the time modelling the control flow and debugging their models, to make them valid for the analyses tools.

In a second paper [13], we further analysed the accuracy of the predictions achieved by the students compared to a sample solution. Additionally, we searched for reasons for the achieved prediction accuracy by analysing the models created during the experiment and evaluating questionnaires filled out by the participants after the experiment. For reasons of self-containedness, sections 2.3, 3.2 - 3.4, 5.1 and 6 are similar in both papers, as they describe and discuss the common experiment setting.

The contributions of this papers are (i) the design of an experimental setting for comparing performance prediction methods, allowing the replication of the study, and (ii) a first quantification of the effort required to produce reusable prediction models.

This paper is organised as follows. Section 2 presents the basics of model-driven performance prediction and briefly introduces SPE and Palladio. Afterwards, Section 3 explains the experimental design, before Section 4 illustrates the results. Section 5 discusses the validity of the empirical study and provides potential explanations for the results. Related work is summarised by Section 6, while Section 7 concludes the paper and sketches future work.

2 Model-Driven Performance Prediction

2.1 Background

Several model-driven performance prediction approaches have been proposed [1], all of which follow a similar process model (Fig. 1). First, developers annotate plain software design models (e.g., UML models) with estimated or already measured performance properties, such as the execution time for an activity or the number of users concurrently issuing requests.

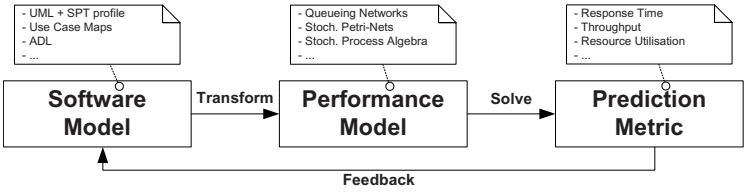


Fig. 1. Performance Prediction Process

Second, model transformations automatically convert the annotated software models into established performance formalisms such as queueing networks (QN), stochastic Petri nets (SPN), or stochastic process algebras (SPA). Existing analytical or simulation-based solution techniques then automatically derive and report performance measures, such as response times for specific use cases, maximum throughputs, or the utilisation of resources, which is crucial for identifying performance bottlenecks. Developers compare the predicted results to their requirements and decide whether to change their design or to start implementation. Only a few approaches implement an automated feedback of the prediction results into the software design model.

For our experiment, we compared our component-based Palladio method [6] with the mature, monolithic Software Performance Engineering (SPE) method [20]. We chose SPE as it has been applied in practice and provides a reasonably usable tool support, unlike many other approaches [11] solely proposed by academics. The following two sections briefly describe the two approaches, which both follow the process model sketched above.

2.2 SPE

The SPE method was the first elaborated, practically applicable comprehensive approach for early, design-time performance prediction for software systems [19]. SPE primarily targets software architects and performance analysts during early development stages. They identify key scenarios (i.e., use cases critical to the overall system performance) and set performance goals for the scenarios (e.g., max. response time) based on the requirements.

Afterwards, developers use a software execution model (Execution Graph, EG) to describe steps within such a performance-critical scenario. EGs are similar to UML activity diagrams and allow annotating each step with resource requirements, for example the number of needed CPU instructions.

With a so-called overhead matrix, software resource requirements in EGs (e.g., a database access) can be mapped to system resources (e.g., 10 ms for a hard disk access per database access). Several scenarios and the corresponding user arrival rates on different machines can be combined to form a system execution model.

EGs do not necessarily reflect actual componentisation of a system, but provide an abstraction of the most performance-relevant steps in a scenario. This is useful for conducting performance analyses as early as possible during the life-cycle of a system, when many details are still unknown. It also limits the developers' effort for initial modelling. However, dependencies on the specific project context are not made explicit, but are mixed with component specifics. Thus, it is usually not possible to readily reuse the resulting performance models when reusing the software components. Additionally, the models cannot be used for model-driven development, as their performance-related abstraction does not provide enough information for other purposes like code generation.

The SPE methodology has been applied in industrial settings. Several anonymised case studies are provided in [20].

2.3 Palladio Component Model

The Palladio Component Model (PCM) [6] is a meta-model for specifying and analysing component-based software architectures with focus on performance prediction.

This meta-model is divided among the separate developer roles of a component-based development process: The component developer produces independent, reusable component specifications. The other roles (software architects, system deployers, domain experts and quality-of-service analysts) provide information on the project-specific context, such as binding of the components, their allocation to hardware and their usage. The meta-model provides each role with a domain-specific language suited to capture their specific knowledge [6].

To support the creation of reusable component performance models, the component specifications are parametrised by influence factors whose later values are unknown to the component developer. In particular, these are the performance measures of external service calls, which depend on the actual binding of the component's required interfaces (provided by the software architect), the actual resource demands which depend on the allocation of the components to hardware resources (provided by the system deployer), and performance-relevant parameters of service calls (provided by the domain expert).

The parametric behavioural specification used in the PCM as part of the software model is the *Resource Demanding Service Effect Specification* (RD-SEFF) which is a control and data flow abstraction of single component services, also similar to UML activity diagrams. It specifies control flow constructs like loops, or branches only if they affect external service calls. Additionally, they abstract component internal computations in so called *internal actions* which only contain the resource demand (e.g. reading 100 Bytes from a hard drive) of the action but not its concrete behaviour. Calling services and parameter passing are specified using *external call actions*, which only refer to the component's required interfaces to stay independent of the component binding. Hence, unlike EGs, RD-SEFFs reflect the componentisation of the system and allow to create component specifications that can be reused in other project contexts. In this

Table 1. GQM plan overview

Goal	**Empirically investigate the effort to create and analyse performance prediction models using Palladio and SPE**	
Question 1	What is the duration of predicting the performance?	
Metric 1.1	Average duration of a prediction	$avd_a = avg(\{d_p \mid p \in P_a\})$
Metric 1.2	Break down of the duration	$avdact_{a,i} = norm_{d_a}(avg(\{dact_{i,p} \mid p \in P_a\}))$
Hypothesis 1.1	A Palladio prediction needs 1.5 as long as an SPE prediction	$avd_{Pal} = 1.5 \cdot avd_{SPE}$
Hypothesis 1.2	For both approaches, the largest time fraction is needed to model the system	

experiment, we thus measure the additional effort required to reflect the componentisation in the Palladio models (in contrast to the SPE models).

3 Empirical Investigation

For the empirical investigation, we formulated a goal, one question and derived metrics using the Goal-Question-Metric approach [4]. The goal of this work is:

> **Goal**: Empirically investigate the effort to create and analyse performance prediction models using Palladio and SPE.

For each metric, hypotheses were formulated to support the evaluation of the metrics and answering the question. The same metrics can also be used when repeating this experiment. Details are presented in section 3.1.

We conducted the investigation as a controlled experiment. Section 3.2 presents the experiment's design, section 3.3 describes the preparation of the participants. The tasks and the experiment execution are presented in section 3.4 and 3.5, respectively.

3.1 Questions and Metrics

For each metric, we have formulated hypotheses to support the evaluation of the metrics and answer the question. After an informal explanation, we give a formal description for the metrics. Table 1 summarises goal, question, metrics, and hypotheses.

Q1: What is the duration of predicting the performance? To evaluate the effort for making a prediction, we looked at the time needed, i.e. the duration, because time (in terms of person-days) is the major factor of effort and costs. For an empirical study of the effort of any software development technique, it is inevitable to include the used tools. Thus, here we measured the effort for the combination of applying the method (SPE and Palladio) and the corresponding tools (SPE-ED and PCM-Bench).

Metric 1.1 is the average duration of making a performance prediction. The duration includes reading the specification (ra), modelling the control flow (cf), adding resource demands (rd), modelling the resource environment (re), modelling the usage

profile (up), searching for errors (err) and analysing (ana). Metric 1.2 breaks down the overall duration into the duration of the different activities of a performance prediction mentioned above.

Our hypothesis 1.1 was that a Palladio prediction needs 1.5 times as long as an SPE prediction. We based this hypothesis on experience from the field of code reuse cost models, where a median relative cost of writing for reuse of 1.5 over several studies was detected by [16, p.29], with a standard deviation of 0.24. Furthermore, hypothesis 1.2 is that the entire modelling, including cf, rd, re, and up, is the largest fraction of the duration with both approaches, which should be the case as the analysis is automated. Still, as the tools are not equally matured and Palladio uses simulation, which takes more time than SPE's analytical solution, the hypothesis is not beyond doubt. Additionally, we did not know whether the results can be readily interpreted by the users, and we wanted to check this assumption.

In the following, the metrics are defined formally. Each variable is defined only once and keeps that definition throughout this work. Let $A = \{SPE, Pal\}$ be the approaches under study. With $a \in A$, let P_a be the set of participants applying approach a.

Metric 1.1: For each participant $p \in P_a$, the duration d_p of making a performance prediction is measured. The duration is averaged over all participants. To do so, the function avg is defined as the arithmetic mean of a set of real values.

Metric 1.1: For $a \in A : avd_a = avg(\{d_p \,|p \in P_a\})$

Metric 1.2: Let $Act = \{ra, cf, rd, re, up, err, ana\}$ be the set of different performance prediction activities mentioned above. We measured the duration of each of the single steps $i \in Act$ for each participant $p \in P_a$ and named it $dact_{i,p}$. We averaged it over all participants and normalised it, i.e. gave it as a percentage of the overall duration avd_a.

Metric 1.2: For $i \in Act, a \in A$:
$avdact_{a,i} = norm_{d_a}(avg(\{dact_{i,p} \,|p \in P_a\}))$

3.2 Experiment Design

The study was conducted as a controlled experiment and investigated the effort with participants who are not the developers of the approaches. The participants of this study were students of a master's level course (see section 5.1 for the discussion of student subjects). In an experiment, it is desirable to trace back the observations to changes of one or more independent variables. Therefore, all other variables influencing the results need to be controlled. The *independent variable* in this study was the approach used to make the predictions. Observed *dependent variables* were the duration of making a prediction and the quality of the prediction to ensure a minimum quality.

The experiment was designed as a changeover trial as depicted in figure 2. The participants were divided into two groups, each applying an approach to a given task. In a second session, the groups applied the other approach to a new task. Thus, each participant worked on two tasks in the course of the experiment (inter-subject design) and used both approaches. This allowed to collect more data points and balanced potential differences in individual factors such as skill and motivation between the two experiment groups. Additionally, using two tasks lowered the concrete task's influence and

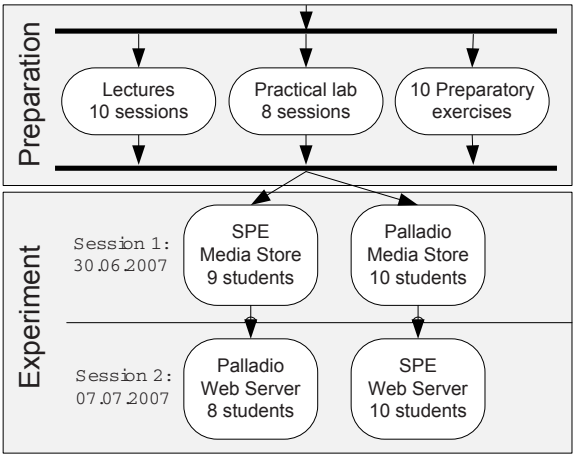


Fig. 2. Experiment design

increased the generalisability. We balanced the grouping of the participants based on the results in the preparatory exercises: We divided the more successful half randomly into the two groups, as well as the less successful half, to ensure that the groups were equally well skilled for the tasks. We chose not to use a counter-balanced experiment design, as we would need to further divide the groups, which would disturb the balancing between the groups. We expected a higher threat to validity from the individual participant's performance than from sequencing effects.

Before handing in, the participants' solutions were checked for minimum quality by comparing the created models to the respective reference model. This acceptance test included the comparison of the predicted response time with the reference model's predicted response time as well as a check for the models' well-formedness.

3.3 Student Teaching

The 19 computer science students participating in the experiment were trained in applying SPE and Palladio during a one-semester course covering both theory and practical labs. For the theory part, there was a total of ten lectures, each of them took 1.5h. The first three lectures were dedicated to foundations of performance prediction and CBSE. Then, two lectures introduced SPE followed by five lectures on Palladio. The three additional lectures on Palladio in comparison to SPE were due to its more complex meta-model which allows for reusable prediction models. Note, that this also shows that reusable models require more training effort. In parallel to the lectures, eight practical labs took place, again, each taking 1.5h. During these sessions, solutions to the accompanying ten exercises were presented and discussed. Five of these exercises practised SPE and five Palladio.

The exercises had to be solved by the participants as homework. We assigned pairs of students to each exercise and shuffled frequently to get different combinations of students work together and exchange knowledge. This was assumed to lower the influence

of individual performance in the experiment. Each exercise took the students 4.75h in average to complete.

Overall, the preparation phase was intended to ensure a certain level of familiarity with the tools and concepts, because participants who failed two preparatory exercises or an intermediate short test were excluded from the experiment.

3.4 Experiment Tasks

To be applicable for both SPE and Palladio, the experiment tasks can only contain aspects that can be realised with both approaches. For example, the tasks cannot make use of the separate roles of Palladio, because these roles are not supported by SPE. Thus, each participant needs to fulfil all roles.

For reasons of compatibility, both experiment tasks had similar set-ups. The task descriptions contained component and sequence diagrams documenting the static and dynamic architecture of a CB system. The sequence diagrams also contained performance annotations. The resource environment with servers and their performance properties was documented textually. The detailed task description is available on-line in [12]. For each system, two usage profiles were given, to reflect both a single-user scenario (*UP1*) and a multi-user scenario leading to contention effects (*UP2*). Additionally, they differed in other performance relevant parameters (see below).

In addition to the initial system, several design alternatives were evaluated. This reflects a common task in software engineering. Four design alternatives were designed to improve the system's performance, and the participants were asked to evaluate which alternative is the most useful one. Three of these alternatives implied the creation of a new component, one changed the allocation of the components and the resource environment by introducing a second machine. With the final fifth alternative, the impact of a change of the component container, namely the introduction of a broker for component lookups, on the performance should be evaluated.

The two systems were prototypical systems specifically designed for this experiment. In the first session, a performance prediction for a web-based system called Media Store was conducted. This system stores music files in a database. Users can either upload or download sets of files. The size of the music files and the number of files to be downloaded are performance-relevant parameters. The five design alternatives were the introduction of a cache component that kept popular music files in memory, the usage of a thread pool for database connections, the allocation of two of the components to a second machine, the reduction of the bit rate of uploaded files to reduce the file sizes and the aforementioned usage of a broker.

In the second session, a prototypical Web Server system was examined. Here, only one use case was given, a request of an HTML page with further requests of potential embedded multimedia content. Performance-relevant parameters were the number of multimedia objects per page, the size of the content and the proportion of static and dynamic content. The five design alternatives were the introduction of a cache component, the aforementioned usage of a broker, the parallelisation of the Web Server's logging, the allocation of two of the components on a second machine and the usage of a thread pool within the Web Server.

The participants using the Palladio approach were provided with the initial repository of available components without RD-SEFFs. It made the tasks for SPE and Palladio more comparable, because the participants still had to create the RD-SEFFs with the performance annotations, which is similar to the creation of an EG in SPE.

3.5 Experiment Execution

The group of 19 computer science students was divided into two groups as shown in figure 2. We conducted two sessions, each with a maximum time constraint of 4.5 hours. One participant did not attend the second session due to personal reasons, thus, only 18 students took part. The participants were asked to document the duration of the activities given in metric 1.2 and to fill in a questionnaire with qualitative questions at the end of the session.

Four members of our chair were present to help with tool problems, the exercise, and the methods, as well as to check the solutions in the acceptance tests. This might have distorted the results, because they might have influenced the duration. The more problems were solved by the experimentators, the less time the participants might have spent on solving them themselves. To avoid this effect, the participants were asked to first try to solve problems on their own before consulting the experimentators. To be able to assess a possible influence of this help, we documented all help and all rejections in the acceptance tests [12].

Because many participants did not finish the task within 4.5 hours in both sessions, the time restriction was loosened afterwards and they were allowed to work another 2.5 hours (session 1) and 2 hours (session 2). In both sessions, three (session 1) respectively two (session 2) participants were not properly prepared, as they needed a lot of basic help or were not able to finish even the initial system prediction. Thus, the results of these three / two participants could not be used. All other participants modelled the initial system and at least one design alternative. Because two participants failed using both approaches, omitting their results does not advantage one of the approaches. Additionally, the time constraints did not distort the results for the initial system prediction, because every remaining participant finished the initial prediction well before the end of the experiment.

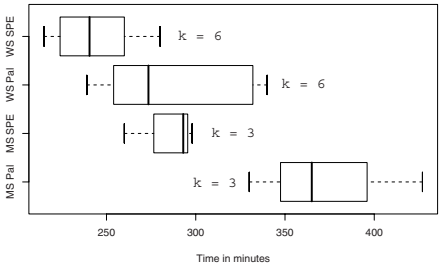
Overall, in session 1, three of the remaining seven participants using Palladio and seven of the nine participants using SPE were able to finish all design alternatives. In session 2, the eight participants using SPE finished all design alternatives, as well as six of the eight participants using Palladio. The acceptance test ensured that the created models were meaningful. As a result, the average deviation of the predicted response time from a reference solution was only about 10%.

4 Results

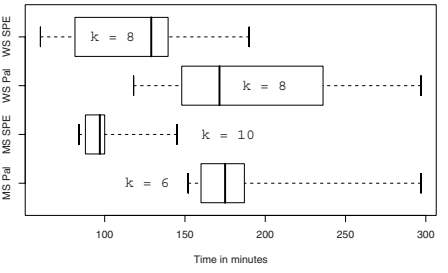
4.1 Metric 1.1: Average Duration of Making a Prediction

First, we evaluated metric 1.1 for the whole experiment task (=: scope wt), thus the duration d_p includes the duration of analysing the initial system and all design alternatives. In neither session, all participants were able to finish the respective task within the

extended time constraints, especially for Palladio. We first looked at those participants who finished the whole task with one approach a: Let k_a be this number of participants. To not favour one approach, only the results of the $k = max(k_{Pal}, k_{SPE})$ fastest participants from both groups were evaluated for metric 1.1, so that for both groups, the slower participants were left out.



(a) Metric 1.1: Duration of whole task (wt) for both approaches and both systems



(b) Metric 1.1: Duration of only the initial system (is), for both approaches and both systems

Figure 3(a) shows the results of metric 1.1 for the four combinations of approaches and systems in a boxplot, showing the minimum, the lower quartile, the mean, the upper quartile and the maximum for all groups and systems. The number of evaluated results is $k = 3$ for the Media Store (MS) and $k = 6$ for the Web Server (WS).

To get more data points, metric 1.1 was also evaluated for the analysis of the initial system only without design alternatives (=: scope is), now considering all participants (except the aforementioned excluded ones). Figure 3(b) shows the resulting boxplot, including the time to read the exercise.

Table 2 shows the average metric 1.1 for all aforementioned combinations. Additionally, we compared how much longer it takes in average to make the Palladio prediction compared to making the respective SPE prediction. These factors are shown as avd_{Pal}/avd_{SPE}. In average over both scopes, the duration for a Palladio prediction was 1.4 times the duration for an SPE prediction.

We tested our initial hypotheses using Welch's t-test [22], as we cannot assume identical variances for the distributions, and chose a significance level of 0.05. For the initial system, the hypothesis 1.1 is not rejected in a two sided test (p=0.15). Using one sided tests, we found that it is unlikely that students using Palladio needed less that 1.5 times the effort than students using SPE (p=0.08), although not significantly. Overall, it is

Table 2. Metric 1.1: Duration of making a prediction in minutes

	Whole task		Avg	Initial system		Avg	Avg
	MS	WS		MS	WS		
avd_{Pal}	374	285	329.5	203	191	197	263
avd_{SPE}	284	243	263.5	99	119	109	186
$\frac{avd_{Pal}}{avd_{SPE}}$	1.32	1.17	1.25	2.05	1.61	1.81	1.41

significant that students using Palladio did need more effort than students using SPE for the initial system only (p=$1.7 * 10^{-4}$). The statistical power of these tests were larger than 0.9 and thus satisfactory. For the whole system, i.e. the actual experimental task, hypothesis 1.1 is rejected (p=0.009). Students using Palladio needed significantly less than 1.5 the time than students using SPE, as the opposite is rejected with p=0.004. The statistical power of these two tests is 0.65 and 0.78, respectively, and barely sufficient [18]. Still, students using Palladio needed significantly more time than students using SPE for the whole task as well, as the opposite is rejected (p=0.01,power 0.78).

4.2 Metric 1.2: Break Down of the Duration

We first looked at the break down of the duration as measured in metric 1.1 into the different activities for the initial system only (scope is), because it represented a creation of the models from scratch and we had more data points for it.

Reading (ra) was only an initial reading of the task description, all participants had to read excerpts of the task again while modelling, which was included in the modelling time. For SPE, the participants did not give a separate time for the annotation of resource demands (rd) , but included this time into modelling of the control flow (cf) or of the resource environment (re). Each experiment task contained two usage profiles, so the duration of their modelling, searching for errors and analysing was measured separately for each usage profile and then averaged.

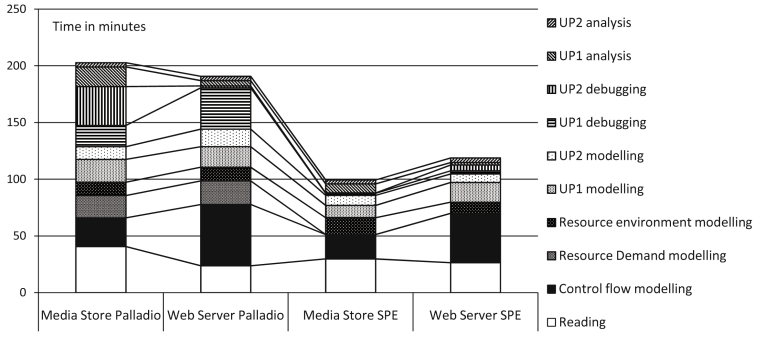


Fig. 3. Metric 1.2: Break down of the duration for the initial system (scope is)

Figure 3 shows the break down of the duration of making a prediction for the initial system, without design alternatives (scope is). It is visible that the entire modelling, including cf, rd, re, and up, was the major activity for both approaches, as expected. The results indicate that hypothesis 1.2 can be held.

Notable results are found for the time needed for searching for errors (err) and the analysis (ana). However, participants using Palladio spent much more time on searching for errors, i.e. fixing wrong or missing parameters: 20%. Here, the participants using SPE only spent 2% (Media Store) and 6% (Web Server), of their time. The proportion of the analyses was fairly constant for the approaches and differs only in the system under study: For the Media Store system, participants spent about 10% of their time in average for the analyses, for the Web Server, only 4%.

The duration of the whole task, i.e. modelling all design alternatives (scope wt) was also composed down to these aspects. The duration of reading $avdact_{a,ra,wt}$ was relatively smaller, because it had just been queried once at the beginning of the task. The other ratios stayed approximately the same. Due to space limitations, we omit the charts here.

5 Discussion

5.1 Threats to Validity

To enable the reader to assess our study, we list some potential threats its validity in the following. We look at the internal, construct, and external validity (a more thorough discussion can be found in [12]).

The *internal validity* states whether changes of an experiment's independent variables are in fact the cause for changes of the dependent variables [23, p.68]. Controlling potential interfering variables ensures a high internal validity. In our experiment, we evaluated the pre-experiment exercises and assigned the students to equally capable groups based on the results to control the different capabilities of the participants. A learning effect might be an interfering variable in our experiment, as the students finished the second experiment session faster than the first one.

A potential bias towards or against Palladio was threatening the internal validity in our experiment, as the participants knew that the experimenters were involved in creating this method. However, we did not notice a strong bias from the collected data and the filled-out questionnaires, as the participants complained equally often about the tools of both approaches.

The *construct validity* states whether the persons and settings used in an experiment represent the analysed constructs well [23, p.71]. Palladio and SPE are both typical performance prediction methods involving UML-like design models. The SPE approach has no special support for component-based systems, and was chosen for the experiment due to its higher maturity compared to existing CBSPE approaches. To allow a comparison, we designed the experimental tasks so that not all specific component-based features of Palladio (e.g. separation of developer roles in component-based development, performance requirements using quantiles) were used.

While our experiment involved student participants, we argue that their performance after the training sessions was comparable to the potential performance of practitioners. Most of the students were close to graduating and will become practitioners soon. Due to the training sessions, their knowledge about the methods was more homogeneous than the knowledge of practitioners with different backgrounds. With a homogeneous group of participants, the significance of the results is even improved. Studies, such as [10], suggest the suitability of students for similar experiments.

The *external validity* states whether the results of an experiment are transferable to other settings than the specific experimental setting [23, p.72]. While we used medium-sized, self-designed systems for the students to analyse, we modelled these system designs and the alternatives after typical distributed systems and commonly known performance patterns [20], which should be representative for the usually analysed systems.

We tried to increase the external validity of our study by letting the participants analyse two different systems, so that differences in the results could be traced back to the systems, and not the prediction methods. Effects that are observed for both tasks are thus more likely to be generalisable to other settings.

Still, the systems under study were modelled on a high abstraction level due to the time constraints of such an experiment. More complex systems would increase the external validity, but would also involve more interfering variables thus decreasing the internal validity. Furthermore, the available information at early development stages is usually limited, which would be reflected by our experimental setting.

5.2 Potential Explanations for the Results

Using SPE, the predictions can be done significantly faster. Using Palladio takes 1.17 to 2.05 times longer, depending on the system under study and the nature of the task. The proportion is higher if we look at the prediction of the initial system only. However, this is not a realistic setting, because a usual task in performance engineering is the comparison of several alternatives. For the evaluation of several alternatives, using Palladio only takes 1.17 or 1.32 times longer. To a certain extent, this can be explained by the reuse character of this scenario: For the prediction of design alternatives, the EGs of SPE were copied and adapted, which is faster than creating new models from scratch, but still a considerable effort. However, for Palladio, the RD-SEFFs of the most components can be reused as is due to their parametrisation, and only single components, their assembly and allocation need to be changed.

To a certain extent, the extra time needed for making Palladio predictions could be traced back to the duration of searching for errors. This might be partly caused by the immaturity of the tool and the limited understandability of the error messages. Using Palladio, more problems occur during creation of the model and searching for errors before the simulation, but the number of problems in the later acceptance tests after simulation is lower than with SPE. The PCM-Bench performs more consistency checks on the models than the SPE-ED tool, thus predictions with the PCM-Bench seem more reliable. However, both tools still allow wrong parameter settings or wrong modelling.

Still, the participants using SPE also needed less time to model and analyse the systems. However, in this experimental setting, not yet considering potential time-savings when reusing models in other projects, SPE is favoured, because it allows to create the models on a higher abstraction level and thus faster. The resulting SPE models are not meant for reuse, which is the case for Palladio models. Furthermore, existing components were not reused in the systems under study and no code was generated from the resulting Palladio models, which might have affected the combined effort of design and implementation. The influence of possible reuse on the effort, however, is deliberately not subject of our experiment and needs further studies.

Next to differences of the approaches presented here, we also found that the results differ for the two systems under study. For the Web Server, both the duration of modelling the control flow and the variance of the overall duration is considerably higher for both approaches.

5.3 Implications for Further Research

Our experiment has several implications for further research. The study could be repeated with a larger sample size to allow a better and more precise quantification of the additional effort. Furthermore, the actual reuse of the created parametrised models in terms of applicability, effort and quality need to be studied. Also more complex, and less componentised systems could be evaluated with the approaches. We also plan to investigate whether cost models on the effort of creating reusable code [16] are suitable for assessing the overhead effort of creating reusable performance prediction models.

For comparative studies between different approaches, a component-based reference system can help avoid researchers applying their methods on their own model examples, which are often tuned to show specific benefits but not general applicability. A recent joint effort by more than 15 research groups has taken steps into this direction by specifying CoCoME (Common Component Modelling Example) [17], which could be used for comparative studies.

6 Related Work

Basics about the area of *performance prediction* can be found in [20,14]. Balsamo et al. [1] give an overview of about 20 recent approaches based on QN, SPN, and SPA. Becker et al. [5] survey performance prediction methods specifically targeting component-based systems. Examples are CB-SPE [7], ROBOCOP [8], and CBML [24].

Empirical studies and controlled experiments [23] are still under-represented in the field of model-based performance predictions, as hardly any studies comparable to ours can be found. Balsamo et al. [3] compared two complementary prediction methods (one based on SPA, one on simulation) by analysing the performance of a naval communication system. However, in that study, the authors of the methods carried out the predictions themselves. Gorton et al. [9] compared predicted performance metrics to measurements in a study, but only used one method for the predictions.

Koziolek et al. [11] conducted a study similar to this one. They compare predictions with SPE [20], Capacity Planning [14], and umlPSI [2] with measurements of an implementation. It attested SPE the most maturity and suitability for early performance predictions and influenced our decision to compare Palladio to SPE.

7 Conclusions

We have conducted an empirical investigation to quantify the higher effort for creating reusable, component-based models for performance prediction in relation to create throw-away models. After substantial training, we let 19 computer science students apply the SPE method and the Palladio method to predict the response times of two example systems. We found that the effort for applying Palladio on the whole task was in average 1.25 times the effort for applying SPE. Our results indicate that in some cases, the effort of creating reusable models for performance prediction can already be justified if the models are reused at least once, if the costs for the reuse itself are low. If the models are reused more often, the additional upfront effort pays off even more.

The results are useful for both practitioners and researchers. Practitioners, such as software architects and performance analysts, get a first quantification of the higher effort to create reusable, component-based models, which they could use in front of management to justify higher upfront costs for modelling. Researchers obtain a reusable experimental setting, which is the basis for future replications of the experiment. The results suggest that it is worthwhile to put more research effort into creating reusable models, because their creation can quickly pay off. However, our study cannot give a definite, overall answer to the questions raised, as the results are also confined to our specific experimental setting.

Our investigation opens up future directions for research. We conducted one of the first empirical studies comparing two performance prediction approaches. The study could be repeated with a larger sample size to allow a better quantification of the additional effort as well as a validation of the results. Furthermore, it has to be assessed whether the promised reusability of the models can be achieved in more complex or less componentised systems. Moreover, the analysis of factors influencing the effort, especially the nature of the systems under study, is an issue for future research.

Details on the Experimental Settings and the Results. can be found in [12], available online at
http://sdq.ipd.uka.de/diploma_theses_study_theses/completed_theses.

Acknowledgements. We would like to thank Walter Tichy, Lutz Prechelt, and Wilhelm Hasselbring for their kind review of the experimental design and fruitful comments. Furthermore, we thank all members of the SDQ Chair for helping prepare and conduct the experiment. Last, but not least, we thank all students who volunteered to participate in our experiment.

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE TSE 30(5), 295–310 (2004)
2. Balsamo, S., Marzolla, M.: A Simulation-Based Approach to Software Performance Modeling. In: Proc. of ESEC/FSE, pp. 363–366. ACM Press, New York (2003)
3. Balsamo, S., Marzolla, M., Di Marco, A., Inverardi, P.: Experimenting different software architectures performance techniques. In: Proc. of WOSP, pp. 115–119. ACM Press, New York (2004)
4. Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering - 2 Volume Set, pp. 528–532. John Wiley & Sons, Chichester (1994)
5. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 169–192. Springer, Heidelberg (2006)
6. Becker, S., Koziolek, H., Reussner, R.: Model-based Performance Prediction with the Palladio Component Model. In: Proc. of WOSP, February 5–8, 2007, pp. 54–65. ACM Sigsoft (2007)

7. Bertolino, A., Mirandola, R.: CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 233–248. Springer, Heidelberg (2004)
8. Bondarev, E., de With, P.H.N., Chaudron, M.: Predicting Real-Time Properties of Component-Based Applications. In: Proc. of RTCSA (2004)
9. Gorton, I., Liu, A.: Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications. IEEE Internet Computing 7(3), 18–23 (2003)
10. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects - A comparative study of students and professionals in lead-time impact assessment. Empirical Software Engineering 5(3), 201–214 (2000)
11. Koziolek, H., Firus, V.: Empirical Evaluation of Model-based Performance Predictions Methods in Software Development. In: Reussner, R., Mayer, J., Stafford, J.A., Overhage, S., Becker, S., Schroeder, P.J. (eds.) QoSA 2005 and SOQUA 2005. LNCS, vol. 3712, pp. 188–202. Springer, Heidelberg (2005)
12. Martens, A.: Empirical Validation of the Model-driven Performance Prediction Approach Palladio. Master's thesis, Carl-von-Ossietzky Universität Oldenburg (November 2007)
13. Martens, A., Becker, S., Koziolek, H., Reussner, R.: An empirical investigation of the applicability of a component-based performance prediction method. In: EPEW 2008, Palma de Mallorca, Spain (accepted, 2008)
14. Menascé, D.A., Almeida, V.A.F., Dowdy, L.W.: Performance by Design. Prentice-Hall, Englewood Cliffs (2004)
15. Petriu, D.C., Wang, X.: From UML description of high-level software architecture to LQN performance models. In: Nagl, M., Schürr, A., Münch, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, Springer. Heidelberg (2000)
16. Poulin, J.S.: Measuring software reuse: principles, practices, and economic models. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)
17. Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.): The Common Component Modeling Example: Comparing Software Component Models. LNCS. Springer, Heidelberg (to appear, 2008)
18. Sachs, L.: Applied Statistics: A Handbook of Techniques. Springer, New York (1982)
19. Smith, C.U.: Performance Engineering of Software Systems. Addison-Wesley, Reading (1990)
20. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Reading (2002)
21. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press, Addison-Wesley, Reading (1998)
22. Welch, B.L.: The generalization of student's problem when several different population variances are involved. Biometrika 34, 28–35 (1947)
23. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers, Dordrecht (2000)
24. Wu, X., Woodside, M.: Performance Modeling from Software Components. SIGSOFT SE Notes 29(1), 290–301 (2004)

Deploying Software Components for Performance

Vibhu Saujanya Sharma[1] and Pankaj Jalote[2]

[1] Accenture Technology Labs India, IBC Knowledge Park, 4/1 Bannerghatta Road, Bangalore, India
vibhu.sharma@accenture.com*
[2] Dept. of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India
jalote@cse.iitd.ac.in

Abstract. Performance is a critical attribute of software systems and depends heavily on the software architecture. Though the impact of the component and connector architecture on performance is well appreciated and modeled, the impact of component deployment has not been studied much. For a given component and connector architecture, the system performance is also affected by how components are deployed onto hardware resources. In this work we first formulate this problem of finding the deployment that maximizes performance, and then present a heuristic-based solution approach for it. Our approach incorporates the software architecture, component resource requirements, and the hardware specifications of the system. We break the problem into two subproblems and formulate heuristics for suggesting the best deployment in terms of performance. Our evaluation indicates that the proposed heuristic performs very well and outputs a deployment that is the best or close to the best, in more than 96% cases.

1 Introduction

Software performance is an important attribute, especially for systems which handle a large number of transactions. It is imperative for these systems to be responsive and to be able to serve a large number of clients simultaneously. Performance of such systems depends on its software architecture (particularly the component and connector view which captures a dynamic structure of the system [5]), and the characteristics of the individual components in the architecture. The architecture and the resource requirements of the individual components affect the way the software system will use the available resources and determines the average amounts of processing required on different types of hardware resources. The machine hardware capabilities in turn determine how fast the different processing demands can be met, and determine the rate at which the client request

* The first author was a graduate student at Indian Institute of Technology Kanpur, when this research work was conducted. This forms a part of his Ph.D. thesis [10].

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 32–47, 2008.

or jobs will be serviced. Many models have been proposed to evaluate the impact of the component and connector architecture of a system on its performance [1,11,14].

Another key factor affecting system performance is deployment of the software components onto the available hardware. For a given component and connector architecture, how the components are deployed on the available hardware affects the performance of the system. The ideal deployment is one in which the hardware devices are adequately utilized, and the software components do not have to face much contention while utilizing the resources, thus resulting in high performance. On the other hand, a bad deployment can severely degrade the system performance, even with efficient software components and fast hardware. Most architecture-based performance evaluation approaches focus on modeling the impact of the component and connector structure on performance, and assume that deployment is given. Some of them recognize the importance of deployment in performance and list it as a future area [7,9].

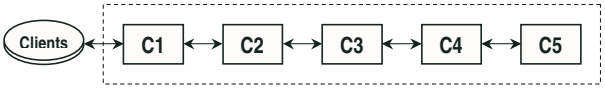
In this paper we address the problem of finding the deployment of software components in a given component and connector model onto the available hardware so as to maximize performance. For simple systems the best deployment may be easy to determine as all possible allocations can be examined. But it is not so for systems with complex software architecture, where a large number of possible options exist for deploying each component.

We develop heuristic-based solution approaches for this problem in this paper. We break the component deployment problem into two sub problems and formulate heuristics for suggesting the best component deployment in terms of performance. We then evaluate the heuristics and compare them using a large set of randomly generated system configurations and identify the best heuristic combination. The proposed heuristic combination gives either the best possible deployment or a deployment very close to the best in more than 96% cases.

In the next section, we look at an example system to appreciate the impact of component deployment on performance and we define the problem formally section 3. In section 4 we describe how by modeling the software architectures using discrete time Markov chains, the total resource demands of components can be determined. In section 5 we present and evaluate various heuristic solutions for the component deployment problem. Section 6 gives some examples, and section 7 concludes the paper.

2 Impact of Deployment on Performance

To appreciate the usefulness of finding solutions to the problem of component deployment, let's first investigate the difference that deployment can make to the system performance. Consider a tiered architecture based system with five components C1 to C5, as shown in Figure 1. In this system 80% of the requests which come to C1 also proceed to C2. The other 20% are served and replied



Percentage of user requests that go from:	
C1 to C2 onwards	80%
C2 to C3 onwards	100%
C3 to C4 onwards	50%
C4 to C5	80%

Average resource demands per visit for tiers (in ms)		
Tier	CPU demands	Disk I/O demands
C1	50	30
C2	60	30
C3	30	70
C4	50	70
C5	50	100

Average communication size per visit (in Kbs)		
Between	Request – >	Reply < –
Client - C1	5	100
C1 - C2	4	5
C2 - C3	5	5
C3 - C4	2.5	5
C4 - C5	4	5

Fig. 1. The example tiered architecture

back by C1 itself. Similarly all requests reaching C2 go further on to C3 while only 50% requests reaching C3 go ahead to C4. Finally 80% of the requests arriving at C4 require service from C5. We assume that the amount of average CPU processing and average disk I/O demands associated with these component running on a standard machine are as given in Figure 1.

Suppose this system is to be deployed on a set of three machines M1, M2 and M3, each with a single CPU and disk. Also, suppose that the machines M1 and M2 are standard machines, i.e., their CPU and disk processing is as fast as the ones used to ascertain the resource demands of the software components, while M3 has a standard CPU but its Disk processing is twice as fast as those. We assume that the network link between the clients and M1 has a uplink (i.e. from clients to M1) capacity of 10Mbps and a downlink capacity of 100Mbps. Further each of the network connectors between the machines has a capacity of 10Mbps for uplink and 10 Mbps for downlink traffic for each pair of machines. The information regarding the average communication size per visit for the software tiers is given in Figure 1. Note that only the communication between adjacent tiers across different machines amounts to network traffic.

Lets assume that a software designer suggests the deployment of components as C1 and C3 at M1, C2 at M2 and C4 and C5 at M3, possibly aiming to deploy the components which need to perform the most disk I/O per component visit at the machine with the faster disk. Consider another deployment where we put C1 on M1, C2 on M2 and C3, C4, C5 on M3.

Using the two deployments, we construct two different performance models of the system using the approach described in [11]. We solve the performance models for a client range of 1 to 100 each with an average thinktime of 1 seconds and plot the average throughput and response time of the two systems using the designer's deployment and our deployment respectively. The results are shown in Figure 2. We can clearly see that the deployment of the software components has a significant impact on performance. In this example, the deployment given by the designer causes the maximum throughput of the system to be limited to about 70% of what the system could provide if the other deployment is used.

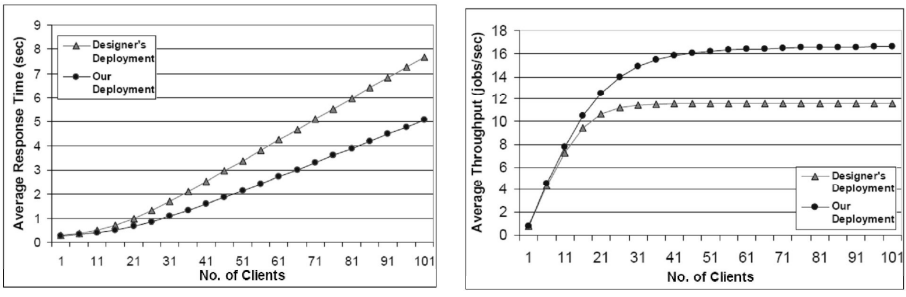


Fig. 2. The average response times and average throughputs for the two deployments

3 The Deployment Problem

We state the component deployment problem formally here. Let C_1, C_2, ..., C_n be n software components of a given software architecture. Each component C_i has its per visit resource requirements $(cpu_i, disk_i, NW_i)$ associated with it. Consider that cpu_i and $disk_i$ denote the component's average CPU processing and the average disk I/O requirements per visit respectively, and NW_i denotes a vector containing the average amount of network traffic that the component generates towards other components, per visit.

The software system has to be deployed on a system of m machines M_1, M_2, ..., M_m. For each machine M_j we have the speed ratings of its CPU and disk given as r_{cj} and r_{dj} respectively, which signify the multiplicative factors by which these devices are faster than those used to measure the per visit demands of the components. Similarly the capacities of the network links in the system are given. If $L(j)$ denotes the set of components deployed on the machine M_j, the deployment problem is to find each of the $L(j)$'s, such that the overall performance is maximized.

Note that Little's Law [13] implies that the average response of a system is inversely proportional to its throughput and thus maximizing throughput implies minimizing the response time. At first, this seems to indicate that one needs to construct the equivalent performance model for the system for evaluating and comparing different deployments. However this is not mandatory. It is well known that the maximum or limiting throughput that a system can support is dependent on the most heavily loaded resource of the system [11,13]. So the problem of maximizing the limiting throughput of the whole system reduces to the problem of minimizing the highest total service requirement on the resources which constitute the system. Thus instead of creating an elaborate performance model, only the value of highest resource requirement per job from among all the devices is needed, to compare different component deployments. The lesser this value, the better the deployment.

For a system with n components and m machines, there are a total of m^n possible deployments to choose from and selecting the deployment that provides the best performance. This makes the problem hard to solve. A restricted form of

the general deployment problem has been studied as the task-processor allocation problem (TAP). A task is an entity that consumes some processing time on the machine when executed, and the goal of TAP is to maximize performance by assigning a set of tasks to a set of machines. However, even this simpler problem is NP-hard in general [3,6]. There do exist a few approximation algorithms for TAP, which reach within some bound of the performance of the optimal deployment. A simple algorithm is to deploy or schedule the tasks one by one (arbitrarily) such that a task is always assigned to the least utilized machine. This greedy algorithm provides a good approximation to the optimal performance [3].

There are a few recent related approaches such as the optimization approach in [8] which forms a cost function, and then uses it to find the optimal allocation of resources to distributed applications. However that work focuses primarily on grid-based systems. The approach presented in [9] allows a user to specify an architectural configuration and initiates the process of actual deploying the components, while assuring the validity of the specified configuration. Their approach can then dynamically monitor the new deployment for comparing its performance with the old configuration. This is very different from the deployment problem that we study here. In [7] the focus is primarily on runtime reconfiguration and changes in the system while minimizing down-time and does not consider the performance aspects. Significantly, both of these approaches ([7,9]) consider incorporating performance aspects while deploying components as future work. Overall, deploying components for maximizing system performance is an important problem and to the best of our knowledge it has not been duly explored till now. In fact it will become even more important with the advent of web-services and service-oriented architecture, with many organisations deploying systems which are inherently distributed in nature, making deployment a key issue.

Before we move on to the solution, one should note that the general deployment problem is different from TAP because of two reasons. Firstly, software components are inherently different from simple 'tasks' in that they utilize some CPU and perform some disk I/O, and send and receive messages, for each job that they service. Hence unlike a 'task' which only has a (single type of) processing demands, a component has in fact at least two types of requirements or demands (CPU processing and disk I/O) on each machine. Secondly, in a component-based system, the load (or resource demand) that the component puts on the resources it uses is not just the component resource demands, but it also depends on its usage as governed by the software architecture of the system. A software component may be executed more than once or even less than once on an average for servicing a job, depending on the software architecture and thus the actual values of the resources that the components demand from the machine they are deployed on, are not immediately clear and need to be calculated. One needs to model the software architecture to calculate the usage of components, and we address this in the next section.

Note that besides the communication time, network I/O among components often leads to CPU as well as disk I/O demands. If bandwidth is high,

communication time can be small and network I/O reflects itself primarily in the CPU and I/O demands of the component. For this study, we assume that the machines on which the components are to be deployed, are connected by high bandwidth connectors and the communication time is small in magnitude compared to the CPU and disk I/O times for each job. Thus, we consider CPU and disk as primary resources in the system.

4 Determining Total Component Demands Using a DTMC Model

Understanding the total resource demands generated by a software component on the hardware resources is important while deploying them. Note that this may not be the same as per visit resource requirements as a component may be visited many times during a job. To compute the total resource demands, we model the software architecture of a system using a discrete time Markov chain (DTMC). DTMCs have been used in literature, for the purpose of modeling and evaluation of component based systems [2,11] for reliability and performance estimation.

A DTMC represents a stochastic process with discrete state and index space whose dynamic behavior is such that 'probability distributions for its future development depend only on the present state and not on how the process arrived in that state' [13]. A DTMC is characterized by its states and transition probabilities among the states. While modeling software architectures using DTMCs, either a single state or a set of states of the DTMC represent the software component in execution at any point in time. Transitions between states are governed by the transfer of control from one component to the other and appropriate probabilities are assigned according to the behavior of the system [10,11]. Reaching an absorbing state i.e. a state from which there is no transition to other states, indicates the successful completion of a job. Other states are termed transient. For example, Figure 3 shows the DTMC model for the tiered architecture specified earlier in Figure 1. This is a DTMC model with two states for each tier, with the upper one representing forward flow of the client request, and the lower one showing the flow of the reply back towards the client [10,11].

One can use DTMCs to calculate the visit counts to each of the states. As explained in [13] the expected total number of visits per job to any transient

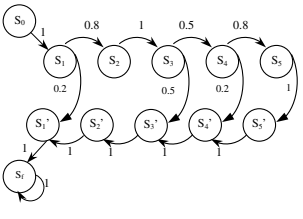


Fig. 3. Representing the tiered architecture using a DTMC

state, starting from an initial state can be calculated using the transition probability matrix of the DTMC. Hence by modeling the system as a DTMC we can find out the average number of times V_i, a component C_i will be visited for a each job. The average visit counts characterize the average usage per job of the components. If the per visit CPU and disk I/O requirements for component C_i are cpu_i and $disk_i$ respectively, the total CPU and disk I/O demands per job for this component will be: $tcpu_i = V_i \times cpu_i$ and $tdisk_i = V_i \times disk_i$ [10]. Thus, the total resource demands per job of each component incorporate the effect of the usage of the component due to the system's software architecture and the problem now reduces to deploying a set of components with known total resource demands onto a set of machines with known device speed ratings. However, the problem is still different from TAP in that unlike tasks, there are two types of demands (CPU and disk I/O) associated with each component. For a discussion on estimation of average component resource demands, refer [10,11].

Note that using DTMCs is not critical to this approach. If one already has the visit counts to different components then they can be used directly while calculating the total CPU processing and disk I/O requirements per job for the components. Next, we present some of the possible heuristics for solving this problem and compare them.

5 Heuristics for Deploying Components

A brute force approach, for deploying components so as to find the deployment with the best performance, would consider all possible deployments for each component. This approach is obviously computationally too expensive. Hence heuristics are necessary. An approach for deploying components is to pick one component at a time for deployment on any of the machines (as was done for assigning tasks to processors in the simple approximation algorithm for TAP [3]). In this case, deciding which machine a particular component should be deployed on, will be based on the resource utilizations of the machines due to the components that have already been deployed and the resource demands of this component.

Thus the effectiveness of the final deployment will clearly depend on the order in which components are considered for deployment as well as on how the deployment is actually done. Therefore, we divide the problem of deploying components for maximizing performance into two sub problems. (A) Deciding the best deployment when components are presented in a given order, and (B) Selecting the order in which the components should be presented for deployment. We consider different heuristics and compare them based on how many times do the suggested orderings result in the best deployment or close to the best deployment, finally selecting the heuristic that performs the best.

5.1 Deciding Deployment for a Given Component Ordering

We first consider the subproblem of deploying a given ordering of components onto available hardware. By an 'ordering', we mean a sequence in which

components are considered for deployment on the machines. Note that in the absence of backtracking, the order in which components will be deployed will affect the performance. We propose two heuristics for deploying the components one by one onto the available machines, with a goal to reach the best deployment. These heuristics use the speed rating of the hardware, and the component resource demands per job to decide where a component should be deployed. We label the heuristics as D1 and D2.

Both these heuristics deploy the given ordering of software components one by one. The heuristics process each component only once (there is no backtracking). For each component, both the heuristics first do a 'mock-deploy' of the software component, in which the component is deployed on each machine one by one, and machine specific information in recorded. The heuristics finally decide where to deploy the component based on this information. Note that after deployment of a component, the total CPU and disk execution times of the corresponding machine will change. For a machine M_j with speed ratings r_{cj} and r_{dj} and with the set $L(j)$ of software components deployed onto it, the aggregate average CPU execution and disk I/O execution times are given by [10]: $tmcpu_j = \sum_{i \in L(j)} (tcpu_i / r_{cj})$ and $tmdisk_j = \sum_{i \in L(j)} (tdisk_i / r_{dj})$.

Before explaining the heuristics, we also introduce some terms. We define individual *component makespan* $make_C$ as the value of its CPU or disk demands per job, whichever is greater. We define *machine makespan*, $make_M$ as its total CPU or disk execution times (due to the deployed components), whichever is greater. Similarly the *system makespan*, $make_S$ is defined as the maximum value of $make_M$ from among all the machines in the system at the time of evaluation. Specifically if $tcpu_i$ and $tdisk_i$ are the average resource demands per job for a component i, and $tmcpu_j$ and $tmdisk_j$ are the average CPU execution and disk I/O times for a machine j , then: $make_{Ci} = MAX\{tcpu_i,\ tdisk_i\}$, $make_{Mj} = MAX\{tmcpu_j,\ tmdisk_j\}$, $make_S = MAX\{Set\ of\ all\ make_{Mj}s\}$. A system with a lower value of $make_M$ is always desirable and will have a better performance in general.

The Heuristics. The first heuristic that we call D1, records the (resulting) value $make_M$ for each machine, if a component were to be deployed on that machine. Then it picks the machine with the least value of $make_M$ for deploying that component. In case of the second heuristic, D2, the value recorded at each deployment step is the resulting sum of CPU and Disk times for each machine, if a particular component were to be deployed on that machine. The machine with the least value of this sum is picked for deploying that component. These heuristics are given in Figure 4.

Experimental Evaluation. For the purpose of evaluation of these heuristics, a number of machine-software component sets are needed. We call each such set as a system configuration. Each such configuration is specified by the number of machines, the number of software components to deploy, a set of CPU and disk speed ratings for each machine, and the resource demands per job for each

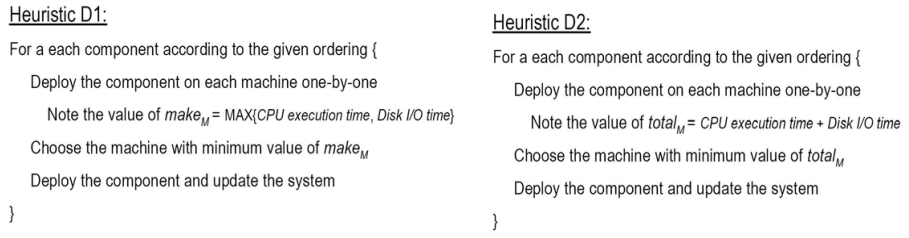


Fig. 4. The Heuristics D1 and D2

component. As the order in which the components were given to be deployed, has a significant impact on the 'goodness' of the deployment, we decided to evaluate both the heuristics for all possible orderings of components for each configuration. For a configuration with n components, this set of all possible orderings is of size $Factorial(n)$ or $n!$.

We used perl scripts to generate a large set of 1000 random configurations. The number of machines was fixed to be 3 and the number of components as 6 and the device ratings and component demands were randomly generated for each configuration. The values of device speed ratings were either 1 or 2, and the component per job resource demands were between 1 and 50 ms in each dimension. Note that for simplicity, we did not create and solve the DTMC models for each of the configurations. Instead we randomly generated the per job resource demands for each component. One can however also generate these random configurations by assigning random transition probabilities among the components and then solving the DTMC model to find the per job resource requirements for the components.

We conducted an experiment wherein each of these configurations was taken one at a time and then both the heuristics were used to deploy each of the 720 possible orderings of the 6 components. This exhaustive analysis, though time consuming, allowed us to point out if a particular heuristic performs better or worse than the other. On trying out all the orderings of each configuration, we recorded the values of system makespan for the deployment that resulted from each ordering both for D1 and well as D2. For comparing D1 and D2 on the basis of the average percentage of orderings they reach close to the (best) deployment giving the best performance for each of the 1000 configurations, we created another set of scripts that exhaustively tried out all possible deployments of the components onto the machines. Note that the number of all possible deployments for m machines and n components is given by m^n, as there are m ways of deploying each of the n components. In this case, it resulted in 3^6=729 possible deployments for each configuration. The value of the least possible system makespan, $make_{S_{best}}$, (which will result in the best performance) was recorded for each of the configurations, and then compared with corresponding system makespan values for deployments due to D1 and D2. We define the deviation of a deployment from best deployment as the difference between the values of system makespan due to the deployment ($make_{S_{dep}}$) and the system makespan

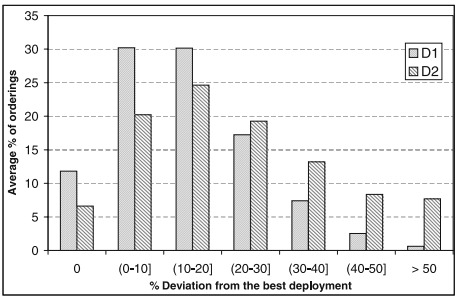


Fig. 5. Percentage deviations from the best deployment for D1 and D2 over all orderings of each configuration

due to the best deployment. The lesser this percentage deviation, the closer the particular deployment is to the best deployment. Thus it is an indicator of how good a particular deployment is. Mathematically the percentage deviation of a deployment from the best deployment is given by:

$$\%dev.\ from\ best\ deployment = \frac{make_{S_{dep}} - make_{S_{best}}}{make_{S_{best}}} \times 100$$

We then found how close the deployments due to D1 and D2 were from the best deployments, across all orderings of each of the 1000 configurations. The Figure 5 shows the average percentage of orderings across all configurations, resulting in deployments having the different percentage deviations from the best deployment, for both D1 and D2. One can see that D1 performs better than D2 and outputs the best deployments almost twice the number of times (out of all possible orderings) on an average. Moreover for majority of orderings the deviations from best deployment in case of D1 are much lesser than D2. Hence, we chose D1 as the deployment heuristic for a given component ordering. However the issue of choosing the appropriate ordering of components still remains and we discuss this next.

5.2 Selecting the Ordering of Components for Deployment

From the experiment explained in the last section, we found that the order in which components are picked for deployment using the chosen heuristic affects 'goodness' of the deployment to a large extent. The best deployment results only for some of the orderings, and so it is important to select the orderings carefully. Here we discuss some heuristics for choosing the orderings of software components for deployment. For finding out the effectiveness of the ordering heuristics, we used the large set of system configurations generated in the previous experiment. The proposed orderings were used along with the deployment heuristic D1 to find the deployments for each of the 1000 configurations and the value of system makespan for those deployments were noted. We examined if the suggested

heuristic deployments resulted in the best makespan. If not then the difference between the best and the corresponding makespans was found to determine how close the performance of the deployment suggested by these heuristics lies to that of the best deployment.

The Heuristics. We formulated four heuristics for ordering components based on their average resource demands per job. The first two heuristics are based on ordering the components with their maximum resource demands in a sorted order. In other words we first ascertain the value of $make_C$ for each component and then sort them according to these. The first heuristic which we shall call O1, does this sorting in the ascending order and the second heuristic, O2, was the reverse, i.e., the maximum demands sorted in descending order. In essence O1, aims at deploying components with smaller demands first, while the use of O2 deploys the components with large demands first.

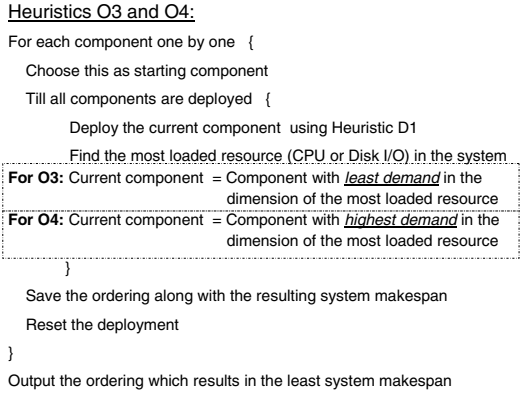


Fig. 6. The Heuristics O3 and O4

While O1 and O2 are static in nature, the next two heuristics are dynamic in nature in that they use the information from the intermediate steps in the deployment process to pick up components and thus result in a on-the-fly ordering of components. Both these heuristics which we shall call O3 and O4, choose the orderings while trying a mock deploy of the components. These heuristics pick up the next component to deploy based upon the resource having the highest system-wide makespan. In case of O3, the next component to be deployed is chosen such that it has the minimum demand per job in the dimension of the current highest makespan resource. For O4, the next component is the one having the highest demand per job in the dimension of the current highest makespan resource. So if during the deployment the current highest makespan resource is the CPU at any machine, O3 will choose the component with the least total CPU demands and O4 will choose one with the highest total CPU demand per job, from among the remaining components.

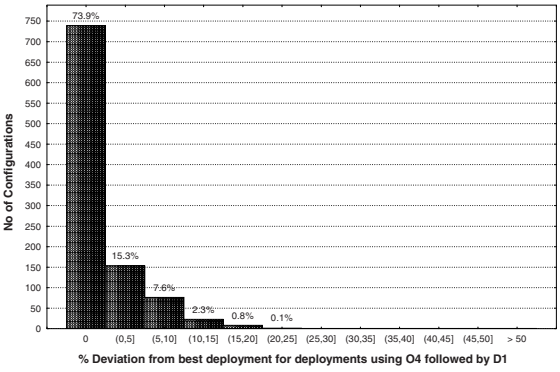


Fig. 7. Deviation from best deployment for deployments using O4 followed by D1

However the problem of choosing the first component remains. So we decided to let O3 and O4 start with each of the components as the first component and proceeded with the heuristic thereon. Thus each O3 and O4 will internally generate a set of 'n' orderings (each starting with a different component) and then choose the best out of these. The details of these heuristics are shown in Figure 6.

Experimental Evaluation. As mentioned before, we used same set of randomly generated configurations as before and then found out how close the deployments (using D1) due to the orderings suggested by the different ordering heuristics, lie with respect to the best deployment. This is captured using the percentage deviation of the different deployments from the best possible deployment. The evaluation results showed that while both O1 and O2 were not good at outputting the best ordering, O2 was better than O1 in general. While orderings due to O1 achieved the best deployment for only 3.5% of the configurations, for O2, this value was at about 40 %. It was seen that while the deployments resulting from the orderings by O1 had high deviations from the best deployment for the bulk of the configurations, for O2 the opposite was true. On an average, for the configurations where these heuristics did not achieve the best deployment, the deployments due to orderings suggested by O1 and O2 have a mean deviation of about 20% and 10% respectively from the best deployment.

Next we evaluated O3 and O4. It was found that for the orderings suggested by O3, the best deployment was reached only about 15% of the configurations - much worse than the simpler O2. The mean deviation from the best deployment for the configurations, where the deployment due to the orderings given by O3 that did not reach the best deployment, was about 11.2%.

On the other hand O4 was a clear winner among all the ordering heuristics. The orderings suggested by the heuristic result in the best deployment in nearly 74% of the configurations (739 out of 1000). Figure 7 shows that even for the configurations, where it fails to reach the best deployment, the deviation from

the best deployment is less than 10% in more than 96% cases. The mean deviation from the best deployment for O4 (followed by D1), for the configurations where it failed to reach the best deployment was only about 5.3%. These results suggest that O4 is indeed a good choice for a heuristic for ordering components. Thus we suggest the use of the heuristic combination that we shall call O4D1, for deploying software components onto available hardware for maximizing performance.

6 Applying the Heuristics

As described in the last section, based on the results, we select O4 and D1 as the deployment heuristics so as to maximize performance. In this section we further discuss the performance of this heuristic combination, which we call O4D1, for deploying systems with different architectural configurations. We consider two different software architectures. The first one being a 5-tiered architecture (which we call Arch1), as shown in Figure 1 and the second one, a 5 component general software architecture (which we call Arch2), a DTMC representation of which is shown in Figure 8. Note that this example system has been taken from [12], where it was used in context of a performance and reliability study. This DTMC model assumes one state per software component. We assume that the per visit resource demands of the components for both Arch1 and Arch2 as shown in Figure 1 (with Comp1 corresponding to C1, and so on). Moreover the available hardware for these architectures is again same as described in Section 2. For simplicity, we assume that the connectors between the machines are sufficiently fast and the effect of network I/O is factored in the CPU and disk I/O demands of the components. We used O4D1 to then output the deployment for both of these. The outputted deployments for Arch1 and Arch2 are shown in Table 1.

Simultaneously, we also generated all possible deployments for Arch1 as well as Arch2. In this case the number of all possible deployments was $3^5 = 243$. The maximum possible or limiting throughput that the system can deliver was

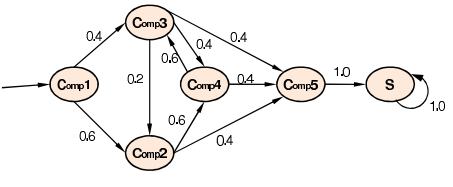


Fig. 8. DTMC model for the architecture Arch2

Table 1. Suggested Deployments for the two Architectures using O4D1

Architecture	On M1	On M2	On M3
Arch1	{C1}	{C2}	{C3, C4, C5}
Arch2	{Comp1}	{Comp2, Comp4}	{Comp3, Comp5}

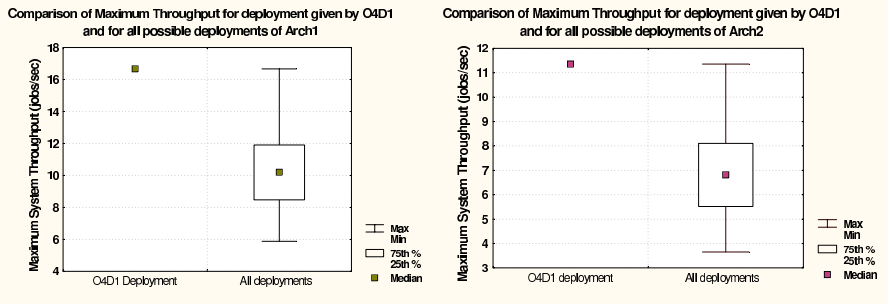


Fig. 9. The performance of the heuristic deployment for Arch1 and Arch2

recorded for both the architectures, for the deployment given by O4D1, as well as for all possible deployments. We show these throughput values for the two architecture configurations using median box-plots as shown in Figure 9.

These plots show that for both the architectures, O4D1 indeed outputs a deployment that maximizes performance. Moreover, if a random deployment is chosen for these configurations, the performance on an average will be pretty bad as compared to that of the O4D1 suggested deployment. Another observation is that with everything else remaining the same, the architecture of system has a big impact on the performance and hence on the best possible deployment, thus reinforcing the need for factoring in the effect of software architecture.

7 Conclusion

Component deployment is a very important factor affecting the performance on any software system. For a given software architecture and hardware resources, there are a huge number of possible component deployments, with different deployments resulting in different overall performance. Thus finding the deployment that results in the best performance becomes important. However the prevalent performance analysis approaches overlook this problem and thus it is left to the software designer to try to select the best deployment manually from among the exponential possibilities.

In this paper we formulated and studied the problem of deploying software components onto available hardware such that the system performance is maximized. We stated the problem formally in terms of the systems's software architecture, component resource requirements, and the hardware specifications. We then explored various heuristic solutions for this problem. Our approach is based on first representing the software architecture of a given system using a DTMC model, the solution of which along with other hardware and software specifications gives us the necessary input parameters for the heuristics for finding the best deployment. We divided the problem into two subproblems, the first one concerned with deploying a given sequence or ordering of components

so as to maximize performance, and the second one for choosing the appropriate component ordering for deployment. We explored different heuristics for both of these.

We conducted a set of experiments, wherein 1000 randomly generated system configurations were created and the heuristics were compared based on the goodness of the deployment they propose. Based on these experiments, we found out the heuristic combination that performed the best, O4D1, chooses the order of components by using information from the deployment process itself. The component deployment is then found, so that at each deployment step, the system makespan is as small as possible. This combination yielded very good results, outputting the best deployment for about 74% of the 1000 randomly generated system configurations and reached close to the best in more than 96% cases.

Our heuristic-based approach can be used at the time of deploying a new systems, as well as finding new deployments for existing systems wherein either the hardware or the software components have changed or the changes are in the system's software architecture itself. Moreover this approach can also be used in an online fashion by autonomous systems to automatically adapt the deployment to any addition, deletion or change that take place in the system as it operates. Note that we have not explored asynchronous event-based systems here and we assume that the component-based systems which need to be deployed are synchronous in nature. We have not incorporated situations with dependencies between components in this study and also have not considered the performance impact of middleware, containers or virtual machines here. Though we consider the effect of network I/O in the CPU and disk usage of the components, we assumed that network delays are negligible in this study. Different deployments would result in different amounts of network transfer delays, which if significant, would affect overall performance and the approach may be extended in future to incorporate this. Costs may also be incorporated, turning this into an optimization problem. Other approaches such as genetic algorithms [4] could also possibly be used to search through the deployments and find the best ones. In real-life systems, certain deployments may not be feasible, and heuristics may be adapted so as to avoid those deployments. Component deployment may also affect other attributes like reliability and availability, and another avenue for future work is to automatically find a deployment that optimizes these different quality attributes.

References

1. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. Software Eng. 30(5), 295–310 (2004)
2. Goseva-Popstojanova, K., Trivedi, K.S.: Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45(2-3), 179–204 (2001)
3. Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal 45, 1563–1581 (1966)

4. Harman, M.: The current state and future of search based software engineering. In: FOSE 2007: 2007 Future of Software Engineering, Washington, DC, USA, pp. 342–357. IEEE Computer Society, Los Alamitos (2007)
5. Jalote, P.: An Integrated Approach to Software Engineering, 3rd edn. Springer, New York (2006)
6. Lenstra, J.K., Shmoys, D.B., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming 46(3), 259–271 (1990)
7. Matevska-Meyer, J., Hasselbring, W., Reussner, R.H.: Software architecture description supporting component deployment and system runtime reconfiguration. In: Proceedings of the Ninth International Workshop on Component-Oriented Programming (WCOP), at ECOOP 2004, pp. 14–18 (2004)
8. Menascé, D.A., Casalicchio, E.: A framework for resource allocation in grid computing. In: DeGroot, D., Harrison, P.G., Wijshoff, H.A.G., Segall, Z. (eds.) MASCOTS, pp. 259–267. IEEE Computer Society, Los Alamitos (2004)
9. Mikic-Rakic, M., Medvidovic, N.: Architecture-level support for software component deployment in resource constrained environments. In: Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD), London, UK, pp. 31–50. Springer, Heidelberg (2002)
10. Sharma, V.S.: Performance and Reliability Analysis of Software Architectures. Doctoral Thesis, Department of Computer Science and Engineering, IIT Kanpur, Kanpur, India (2007)
11. Sharma, V.S., Jalote, P., Trivedi, K.S.: A performance engineering tool for tiered software systems. In: Proceedings of the 30th IEEE Annual International Computer Software and Applications Conference (COMPSAC), pp. 63–70. IEEE Computer Society, Los Alamitos (2006)
12. Sharma, V.S., Trivedi, K.S.: Reliability and performance of component based software systems with restarts, retries, reboots and repairs. In: Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006), pp. 299–310. IEEE Computer Society, Los Alamitos (2006)
13. Trivedi, K.S.: Probability and Statistics with Reliability, Queuing, and Computer Science Applications. John Wiley and Sons, New York (2001)
14. Williams, L.G., Smith, C.U.: Performance evaluation of software architectures. In: WOSP 1998: Proceedings of the 1st international workshop on Software and performance, pp. 164–177. ACM Press, New York (1998)

Performance Prediction for Black-Box Components Using Reengineered Parametric Behaviour Models

Michael Kuperberg, Klaus Krogmann, and Ralf Reussner

Chair for Software Design and Quality, University of Karlsruhe, Germany
{mkuper,krogmann,reussner}@ipd.uka.de

Abstract. In component-based software engineering, the response time of an entire application is often predicted from the execution durations of individual component services. However, these execution durations are specific for an execution platform (i.e. its resources such as CPU) and for a usage profile. Reusing an existing component on different execution platforms up to now required repeated measurements of the concerned components for each relevant combination of execution platform and usage profile, leading to high effort. This paper presents a novel integrated approach that overcomes these limitations by reconstructing behaviour models with platform-independent resource demands of bytecode components. The reconstructed models are parameterised over input parameter values. Using platform-specific results of bytecode benchmarking, our approach is able to translate the platform-independent resource demands into predictions for execution durations on a certain platform. We validate our approach by predicting the performance of a file sharing application.

1 Introduction

To meet user requirements, software must be created with consideration of both functional and extra-functional properties. For extra-functional properties such as performance (i.e., response time and throughput), early analysis and prediction reduce the risks of late and expensive redesign or refactoring in case the extra-functional requirements are not met. Performance of component-based applications is predicted on the basis of performance of underlying components.

The performance of component-based applications depends on several factors [2]:

a) the *architecture* of the software system, i.e. the static "component assembly"
b) the *implementation* of the components that comprise the software system
c) the *runtime usage context* of the application (values of input parameters etc.) and
d) the *execution platform* (hardware, operating system, virtual machine, etc.)

Conventional performance prediction methodologies do not consider all four factors separately [4,24] or limit themselves to real-time/embedded scenarios [7]. To make the influence of these factors on performance explicit (or even quantifiable), these approaches would need to re-benchmark each component, or even the entire application each time one of the four factors changes. Instead, separating these factors is beneficial for efficient performance prediction in the following scenarios:

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 48–63, 2008.

- **Redeployment** of an application to an execution platform with different characteristics, i.e. into a new *deployment context*.
- **Sizing** of suitable execution platform to fulfill changed performance targets for an existing software system, for example due to changes in the *usage context* (i.e., number of concurrent users, increased user activity, different input).
- **Reuse** of a single component in another architecture or **architectural changes** in an existing software system, i.e. changes in the *assembly context* of a component.

In this paper, we present a novel integrated approach that makes these factors explicit and quantifiable.

Our first contribution is a validated reverse engineering approach that uses machine learning (genetic programming) on runtime monitoring data for creating *platform-independent* behaviour models of black-box components. These models are parameterised over usage context and deployment context.

Our second contribution is the performance prediction for these behaviour models, which predicts *platform-specific* execution durations on the basis of bytecode benchmarking results, allowing performance prediction for components and also entire component-based applications. Re-benchmarking an application for all relevant combinations of usage and deployment contexts is thus not necessary anymore.

We validate our approach by reconstructing a performance prediction model for a file sharing application and subsequently predict the execution duration of the application, depending on usage context and deployment context. To the best of our knowledge, this is the first validated bytecode-based performance prediction approach. We describe how our approach maintains the black-box property of components by working without their source code and without needing the full inner details of their algorithms and implementations.

The paper is structured as follows: in Section 2, we describe related work. In Section 3, an overview of our approach is given and its implementation is described. Using a case study of a file-sharing application, we evaluate our approach in Section 4. The limitations and assumptions of the presented approach and its implementation are provided in Section 5, before the paper concludes in Section 6.

2 Related Work

This paper is related to reverse engineering of performance models, bytecode-based performance prediction, and search-based software engineering [12].

Reverse engineering of performance models using traces is performed by Hrischuk et al. [14] in the scope of "Trace-Based Load Characterisation (TLC)". In practice, such traces are difficult to obtain and require costly graph transformation before use. The target model of TLC is not component-based.

Using trace data to determine the "effective" architecture of a software system is done by Israr et al. in [16]. Using pattern matching, this approach can differentiate between asynchronous, blocking synchronous, and forwarding communication. Similar to our approach, Israr et al. support components and have no explicit control flow, yet they do not support inter-component data flow and do not support internal parallelism in component execution as opposed to the approach presented in this paper. As in TLC, Israr et al. use Layered Queueing Networks (LQNs) as the target performance model.

Regression splines are used by Courtois et al. in [9] to recognise input parameter dependencies in code. Their iterative approach requires no source code analysis and handles multiple dimensions of input, as does the approach described by us. However, the output of the approach in [9] are polynomial functions that approximate the behaviour of code, but which are not helpful in capturing discontinuities in component behaviour. The approach is fully automated, but assumes fixed external dependencies of software modules and fixed hardware.

Search based approaches such as simulated annealing, genetic algorithms, and genetic programming have been widely used in software engineering [12]. However, these approaches have not been applied to reverse engineering, but to problems like finding concept boundaries, software modularization, or testing.

Daikon by Ernst et al. [11] focusses on detection of invariants from running programs, while our approach aims at detecting parametric propagation and parametric dependencies of runtime behaviour w.r.t performance abstractions. Analysis is in both approaches supported by genetic algorithms.

Performance prediction on the basis of bytecode benchmarking has been proposed by several researchers [13,23,25], but no working approach has been presented and no libraries or tools are available. Validation has been attempted in [25], but it was restricted to very few Java API methods, and the actual bytecode instructions were neither analysed nor benchmarked. In [18], bytecode-based performance prediction that explicitly distinguishes between method invocations and other bytecode instructions has been proposed.

Obtaining execution counts of bytecode instructions is needed for bytecode-based performance prediction, and has been addressed by researchers (e.g. [5], [19]) as well as in commercial tools (e.g. in profilers, such as Intel VTune [10]). ByCounter [19] counts bytecode instructions and method invocations *individually* and it is portable, light-weight, and transparent to the application. ByCounter works for black-box components and its Java implementation will be used in this paper.

Execution durations of individual bytecode instructions have been studied independently from performance prediction by Lambert and Brown in [20], however, their approach to *instruction timing* was applied only to a subset of the Java instruction set, and has not been validated for predicting the performance of a real application. In the Java Resource Accounting Framework [6], performance of all bytecodes is assumed to be equal and parameters of individual instructions (incl. names of invoked methods) are ignored, which is not realistic. Hu et al. derive worst-case execution time of Java bytecode in [15], but their work is limited to real-time JVMs.

Cost analysis of bytecode-based programs is presented by Albert et al. in [1], but neither bytecode benchmarks not actual realistic performance values can be obtained, since the performance is assumed to be equal for all bytecode instructions.

3 Reverse Engineering and Performance Prediction

An overview of our approach is summarised in Fig. 1, and Sections 3.1-3.6 provide detailed descriptions of its steps. Our approach consists of two parts, separated in Fig. 1 by the dashed line. The first (upper, light) part **A** produces behavioural performance

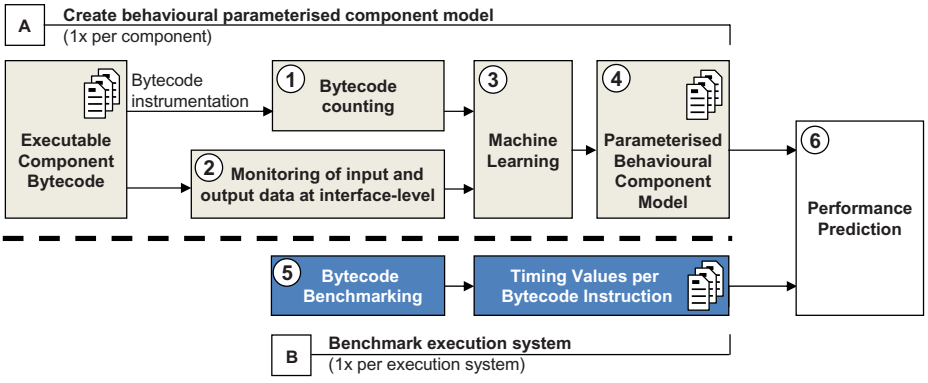


Fig. 1. Overview on the approach

models for black-box components. These models are platform-independent because they do not contain platform-specific timing values. Such timing values are produced by the second part of our approach (cf. (lower, darker) part **B** of Fig. 1), which uses bytecode benchmarking as described in Section 3.5.

Part **A** of our approach works on components and for component-based applications, for which only binary code and no source code may be available. In Step 1, the considered component is executed in a testbed (e.g. derived from running representative applications) and executed bytecode instructions and method invocations are counted. In Step 2 (which can be executed concurrently with Step 1), inputs and outputs of the considered component are monitored at the interface level.

Machine learning in Step 3 then (i) estimates the parametric dependencies between input data and the number of executions for each bytecode instruction and (ii) finds data and control flow dependencies between provided and required service, which is important for components since output of one service will be the input of another one.

Step 4 uses results from machine learning in Step 3 and constructs a behavioural component model which is parameterised over usage context (input data), external services and also over the execution platform. Such a model includes how input data is populated through an architecture, a specification how often external services are called with which parameters, and how an execution platform is utilised.

Part **A** of our approach is executed once per component-based application.

In Step 5 (part **B**), bytecode instructions are benchmarked on the target execution platform to gain timing values for individual bytecode instructions (e.g. "`IADD` takes 2.3 ns"). These timing values are specific for the used target platform, and the benchmarking step is totally independent of the previous steps of our approach. Hence, benchmarking must be executed only once per each execution platform for which the performance prediction is to be made.

The results of part **A** and part **B** are inputs for the performance prediction (Step 6), which combines performance model and execution platform performance measures to predict the actual (platform-specific) execution duration of an application executed on that platform.

The separation of application performance model and execution platform performance model allows to estimate the performance of an application on an execution platform without actually deploying the application on that platform, which means that in practice, one can avoid buying expensive hardware (given a hardware vendor providing benchmarking results) or also avoid costly setup and configuration of a complex software application.

3.1 Counting of Bytecode Instructions and Method Invocations

To obtain runtime counts of executed bytecode instructions (cf. Fig. 1, Step 1), we use the ByCounter tool, which is described in detail in [19] and works by instrumenting the bytecode of an executable component. We count all bytecode instructions individually, and also count method invocations in bytecode, such as calls to API methods.

The instrumentation with the required counting infrastructure is *transparent*, i.e. the functional properties of the application are not affected by counting and all method signatures and class structures are preserved. Also, instrumentation runs fully automated, and the source code of the application is not needed.

At runtime, the inserted counting infrastructure runs in parallel with the execution of the actual component business logic, and does not interfere with it. The instrumentation-caused counting overhead of ByCounter is acceptable and is lower than the overhead of conventional Java profilers. As said before, the instruction timings (i.e. execution durations of individual instruction types) are not provided by the counting step, but by bytecode benchmarking in Step 5.

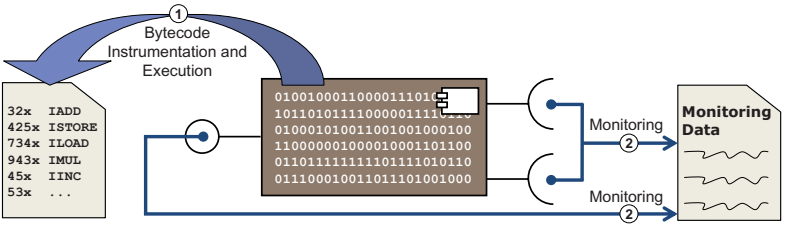


Fig. 2. Data extraction from executed black-box components

The counting results (cf. Fig. 2, Step 1) are counts for each bytecode instruction and found method signature, and they are specific for the given usage context. The counting is repeated with different usage contexts of the considered component service, but on the same execution platform. Counting results are saved individually for each usage context and later serve as data on which machine learning is run (cf. Fig. 1, Step 3).

3.2 Data Gathering from Running Code

To capture the parametric dependencies between the application input and output, our approach monitors at the level of component interfaces (cf. Fig. 1, Step 2). We gather runtime information about component behaviour by executing the component in a testbed or executing the entire application (cf. Fig. 2, Step 2).

To obtain representative data, the execution of the monitored component services must be repeated for a set of representative inputs to the application (recent overview on test data generation can be found in [21]). The datasets obtained from monitoring serve as the input for the machine learning (Fig. 1, Step 3) to learn the parametric dependencies between input and output.

For each component service call (provided or required), our tool monitors the input parameter values of each component service call and the properties of the data that is returned by that service call. Monitored data properties are:

- for primitive types (i.e. `int`, `float` etc.): their actual values
- for all *one*-dimensional arrays (e.g. `int[]`, `String[]`), `Collection`, or `Map` types: the number of their elements
- for one-dimensional arrays of primitive type (e.g. `int[]`), also aggregated data, such as number of occurrences of values in an array (e.g. the number of '0's and '1's in an `int[]`)
- for a *multi*-dimensional array (e.g. `String[][]`): its size, plus results of individual recording of each included array (as described above)

For each *provided* service, we additionally monitor which required services are called by it how often and with which parameters. The described data monitoring and recording can be applied to component interfaces without a-priori knowledge about their semantics, and without inspecting the internals of black-box components. Supporting and monitoring complex or self-defined types (e.g. objects, structs) requires domain expert knowledge to identify important properties of these data types. Still, generic data types are used very often, and our approach can handle these cases automatically.

3.3 Machine Learning for Recognition of Parametric Dependencies

Our approach utilises machine learning for estimating the bytecode counts on the basis of input data and for recovering functional dependencies in the monitored data. We use the Java Genetic Algorithm Package JGAP [22] to support machine learning (a general introduction for genetic programming, a special case of genetic algorithms, can be found in [17]). For our approach, we combine genes representing mathematical functions to express more complex dependencies. Simple approaches like linear regression could be applied as well, but cannot handle non-continuous functions or produce little readable approximations by polynomials.

For every gathered input data point (e.g. size of an input array, or value of a primitive type) a gene representing that parameter in the resulting model is introduced. In addition to default JGAP genes (e.g. mathematical operations for power, multiplication, addition, constants), we introduced new genes to support non-continuous behaviour (e.g. jumps caused by "if-then-else") as JGAP allows defining of additional genes.

Learning Counts of Bytecode Instructions and Method Invocations
Genetic programming tries to find the best estimation of functions of bytecode counts over input data. If an algorithm uses less `ILOAD` instructions for a 1 KB input file than for a 100 KB file, the dependency between input file size and the number of `ILOAD` instructions would be learned.

Our approach applies genetic programming for each used bytecode instruction. A simple example of the resulting estimation for the `ILOAD` instruction is $IF(filesize > 1024)\ THEN\ (filesize \cdot 1.4 + 300)\ ELSE\ (24000))$. For bytecode instructions and method invocation counts, learning such functions produces more helpful results as mere average counts, because non-linear dependencies can be described appropriately, and also because these results are not specific to one execution platform.

Learning Functional Dependencies between Provided and Required Services Genetic programming is also applied for discovering functional dependencies between input and output data monitored at the component interface level. Informal examples of such dependencies are "a required service is executed for every element of an input array", "a required service is only executed if a certain threshold is passed" (data dependent control flow), or "the size of files passed to a required component service is 0.7x the size of an input file" (data flow).

To recover such dependencies from monitoring data, genetic programming builds chromosomes from its genes to express a function matching the monitored data as much as possible. The deviation between learned function and monitored data is used as "fitness function" during learning. Thereby, genetic programming is selecting appropriate input values and rejecting others, not relevant for the functional dependency. Finally, the resulting function is an approximation of a component's internal control and data flow, where each dependency is represented by an own chromosome.

3.4 Parameterised Model of Component Behaviour

The target model (named "Parameterised Behavioural Component Model" in Fig. 1) is an instance of the Palladio Component Model [3][1]. The model instance has a representation for the static structure elements (software components, composition and wiring; not described here) and a behavioural model for each provided service of a component (an example is shown in Fig. 3).

A component service's behaviour model consists of *internal actions* (i.e. algorithms) and *external calls* (to services of other components). For internal actions (cf. left box in Fig. 3), reverse-engineered annotations for each bytecode instruction specify how often that instruction is executed at runtime, depending on component service's input parameters. The parameterised counts that form these annotations are platform-independent and do not contain platform-specific timing values.

For external calls (e.g. `add` and `store` in Fig. 3), the model includes dependencies between component service input and external call parameters, with one formula per input parameter of an external call (e.g. `a = input1 * 2` in Fig. 3). Also, the number of calls to each required (external) service is annotated using parameterisation over input data (cf. "Number of loops" grey box in 3).

3.5 Benchmarking Java Bytecode Instructions and Method Invocations

For performance prediction, platform-specific timings of all bytecode instructions and all executed API methods must be obtained, since the reverse engineered model from

[1] See http://www.palladio-approach.net

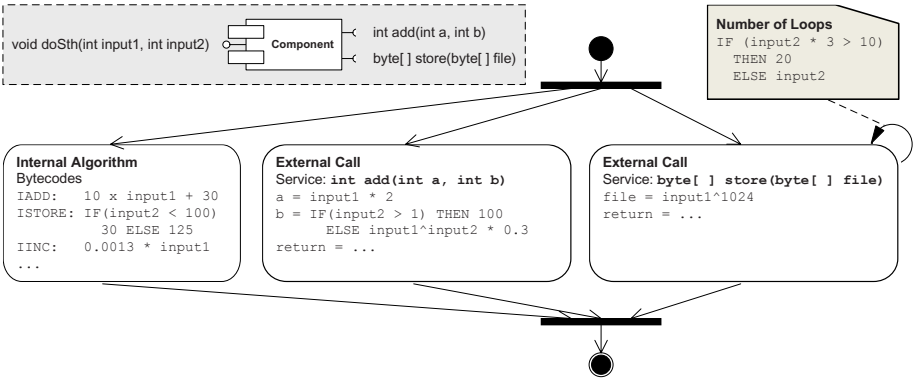


Fig. 3. Behavioural model of the provided service `void doSth(int input1, int input2)`

part **A** in Fig. 1 only contains their (estimated) counts. As timings for bytecode instructions and API methods are not provided by the execution platform and no appropriate approach exists to obtain them (cf. Section 2), we have implemented our own benchmark suite, which is an essential contribution of this paper.

We illustrate our approach by first considering the example of the Java bytecode instruction `ALOAD`. This instruction loads an object reference onto the stack of the Java Virtual Machine (JVM). To measure the duration of `ALOAD`, a naive approach would insert one `ALOAD` between two timer calls and compute the difference of their results. However, writing such a microbenchmark in Java *source code* is not possible, since there is no source code-level construct which is compiled *exactly* to `ALOAD`.

Furthermore, the resolution of the most precise Java API timer (`System.nanoTime()`) of ca. 280ns is more than two orders of magnitude larger than the duration of `ALOAD` (as shown by our microbenchmarks results). Hence, bytecode microbenchmarks must be constructed through *bytecode engineering* (rather than *source code writing*) and must consider the timer resolution.

Using bytecode engineering toolkits like ASM [8], we could construct microbenchmarks that execute a large number of `ALOAD` instructions between two timer calls. However, to fulfill the bytecode correctness requirements which are enforced by the JVM bytecode verifier, attention must be paid to pre- and postconditions. Specifically, `ALOAD` loads a reference from a register and puts it on the Java stack. However, at the end of the method execution, the stack must be empty again. The microbenchmark must take care of such stack cleanup and stack preparation explicitly.

In reality, creating pre- and postconditions for the entire Java bytecode instruction set is difficult. Often, "helper" instructions for pre-/postconditions must be inserted *between* the two timer calls. In such a case, "helper" instructions are measured together with the actually benchmarked instructions. Thus, *separate* additional "helper" mircobenchmarks must be created to be able to subtract the duration of "helper" instructions from the results of the actual microbenchmarks. Making sure all such dependencies are met and resolved is a challenging task.

Due to space restrictions, we cannot go into further details by describing the design and the implementation of our microbenchmarks. In fact, we have encapsulated the benchmarking into a toolchain that can be used without adaptation on any JVM. End users are not required to understand the toolchain unless they want to modify or to extend it. Selected results of microbenchmarks for instructions and methods will be presented in Section 4 in the context of a case study which evaluates our approach and thus also the microbenchmark results.

3.6 Performance Prediction

Step 6 performs an elementwise multiplication of all N relevant instruction/method counts c_i from step 4 with the corresponding benchmark results (execution durations) t_i from step 5. The multiplication results are summed up to produce a prediction P for execution duration: $\sum_{i=0}^{N} c_i \cdot t_i \;=:\; P$. The parametrisation over input can be carried over from c_i to the performance prediction result P, for example by computing that an algorithm implementation runs in $(n \cdot 5000 + m \cdot 3500)$ ns, depending on n and m which characterise the input to the algorithm implementation.

4 Evaluation

We evaluated our approach in a case study on the PALLADIOFILESHARE system, which is modeled after file sharing services such as RapidShare, where users upload a number of files to share them with other users. In PALLADIOFILESHARE, the uploaded files are checked w.r.t. copyright issues and whether they already are stored in PALLADIOFILESHARE. For our case study, we consider the upload scenario and how PALLADIOFILESHARE processes the uploaded files.

The static architecture of PALLADIOFILESHARE is depicted in Figure 4. The component that is subject of the evaluation is PalladioFileShare (the composite component shaded in grey), which provides the file sharing service interface and itself requires two external storage services (LARGEFILESTORAGE is optimized for handling large files and SMALLFILESTORAGE is for handling small files).

PALLADIOFILESHARE component is composed from five sub-components. The BUSINESSLOGIC is controlling file uploads by triggering services of sub-components. COMPRESSION (a Lempel-Ziv-Welch (LZW) implementation) allows to compress uploaded files, while HASHING allows to produce hashes for uploaded files. EXISTINGFILEDB is a database of all available files of the system; COPYRIGHTEDFILESDB holds a list of copyrighted files that are excluded from file sharing.

Fig. 5 shows the data dependent control flow of the BUSINESSLOGIC component which is executed for each uploaded file. First, based on a flag derived from each uploaded file, it is checked whether the file is already compressed (e.g., a JPEG file). An uncompressed file is processed by the COMPRESSION component.

Afterwards, it is checked whether the file has been uploaded before (using EXISTINGFILEDB and the hash calculated for the compressed file), since only new files are to be stored in PALLADIOFILESHARE. Then, for files not uploaded before, it is checked whether they are copyrighted using COPYRIGHTEDFILESDB. Finally, non-copyrighted

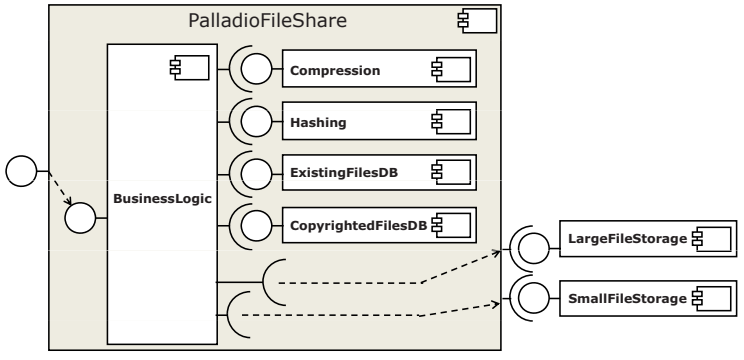


Fig. 4. Component Architecture of PALLADIOFILESHARE

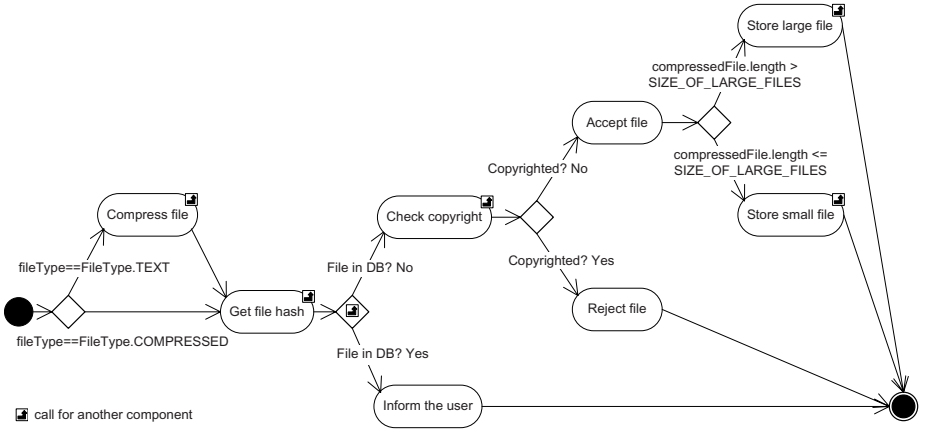


Fig. 5. Activity of BusinessLogic for each file per request (depicted here for readers convenience only; not seen by the tooling)

files are stored, either by LARGEFILESTORAGE for large files (if the file size is larger than a certain threshold) or SMALLFILESTORAGE otherwise.

4.1 Machine Learning for Recognition of Parametric Dependencies

In the case study, we monitored the behaviour of BusinessLogic in 19 test runs, each with different input data (number of uploaded files, characterisation of files (text or compressed), and file sizes). The test runs were designed to cover the application input space as far as possible. In the rest of this section, we show some interesting excerpts from the complete results.

As the names of input parameters are used hereafter, we use the signature of the file sharing service, `void uploadFiles(byte[][] inputFiles,int[] file`

`Types)`. In the signature, `inputFiles` contains the byte arrays of multiple files for upload and `fileTypes` is an array indicating corresponding types of the files, e.g. `FileType.COMPRESSED` or `FileTypes.TEXT` (i.e., uncompressed).

Data dependent control flow: Use of Compression component for multiple files
In the BusinessLogic sub-component, the number of calls of the Compress component depends on the number of uncompressed files (`FileType.TEXT`) uploaded. Genetic programming (JGAP) found the correct solution for the number of calls: *inputFiles.length - fileTypes.SUM(FileType.TEXT)*, where SUM is aggregated data from the monitoring step. The search time was less than one second.

Learning of Bytecode Instruction Counts
For estimating the functional dependencies of bytecode counts, two input variables were monitored: (i) *X1* as the size of each file and (ii) *X2* as flag showing whether the input was already compressed (*X2=1*) or not (*X2=0*).

As the behaviour of data compression algorithm strongly depends on the inner characteristics of the compressed data (and not only on its size and type), 100% precision of learned functions cannot be expected in the general case. Optimal solutions were found only for a few bytecode counts; in most cases, results of machine learning are good estimators. An optimal solution was found within about 1 sec. for bytecode instruction`ICONST_M1` : $X1 + X1 + 3.0$.

For a more complex case such as the bytecode `ICONST_0`, after evolving 15,000 generations, the following approximation (which could be simplified by a subsequent step) was found:

$0.1 + (((X1/(89.0 + 241.0 + 100.0)) * X1) + (((343.0 * X2) + (IF(267.0 >= 10.0)THEN(1.3)) + (241.0 + (X1/(241.0 + (343.0 * X2) + 100.0)) + 100.0)) * X1) + ((IF(X1 >= 0.024766982)THEN((267.0 * X2))) * X2)) + (241.0 + (200.0 + ((89.0 + (((((X1/(X1 + 241.0 + 100.0)) + 30.0 + 241.0) + X2 + 1.9) * X2) * X2) + 100.0) * ((IF(267.0 >= 10.0)THEN((1.3 * X2))) + X1 + 241.0)) + 400000.0))$

The complexity of these functions will be hidden from the user in the performance prediction toolchain.

When to use LargeFileStorage or SmallFileStorage
For answering this question, monitoring data from uploads with just one file was analysed. A set of eleven different input files (different file types, different size) was used as test data. JGAP found an optimal solution: If the file size is larger than 200,000 (bytes), a file with the same size like the file passed to the Hashing component is passed to LargeFileStorage, else nothing is stored with LargeFileStorage (an opposite dependency was found for the usage of SmallFileStorage). The search time was less than five seconds. The implementation-defined constant '200,000' was not always identified correctly, due to the limited number of input files, yet the recovered function did not contradict the monitoring data.

We tested an additional run of JGAP where the monitoring data was disturbed by calls of *uploadFiles* that did not lead to a storage write because the file already existed in the database (one out of eleven calls did not lead to a write). Such effects depending

on component state are visible at the interface level only as statistical noise that cannot be explained based on interface monitoring data. In this case the optimal solution could still be found, but within more time: less than 20 seconds (in average). In this case the confidence in the correctness ("fitness function" calculated by JGAP) of the result decreased. The average behavioural impact of uploads where no storage takes place can be captured by computing the long-term probability of such uploads independently of the uploaded files.

Estimation of the compression ratio
As the compression ratio of LZW strongly depends on the data characteristics (e.g. entropy, used encoding), no optimal solution exists to describe the compression ratio. Therefore, JGAP produces a large variety of approximations of the compression ratio. A good approximation found after 30 seconds had the following form: $0.9 * 0.5 * (X3 - (0.9 * 0.5 * (X3 - (0.9 * (0.9 * 0.5 * X3) * 1.0)))))$ where $X3$ is the size of the file input for the Compression component, which was found to be significant.

4.2 Benchmarking of Bytecode Instructions and Method Invocations

We have benchmarked bytecode instructions and methods (as described in Section 3.5) on two significantly different execution platforms to make performance prediction for the redeployment scenario (cf. Section 1). The first platform ("P1") featured a single-core Intel Pentium M 1.5 GHz CPU, 1 GB of main memory, Windows XP and Sun JDK 1.5.0_15. The second platform ("P2") was an Intel T2400 CPU (two cores at 1.83GHz each), 1.5GB of main memory and Windows XP Pro with Sun JDK 1.6.0_06.

All microbenchmarks have been repeated systematically and median of measurements has been taken for each microbenchmark. Fig. 6 is an excerpt of the results of our microbenchmark for P1 and P2. It lists execution durations of 9 bytecode instructions among those with highest runtime counts for the compression service.

Due to the lack of space, full results of our microbenchmarks cannot be presented here, but even from this small subset of results, it can be seen that individual results differ by a factor of three (`ARRAYLENGTH` and `ICONST_0`). Computationally expensive instructions like `NEWARRAY` have performance results that depend on the passed parameters (e.g. size of the array to allocate), and our benchmarking results have shown that the duration of such instructions can be several orders of magnitude larger than that of simpler instructions like `ICONST_0`.

The most important observation we made when running the microbenchmarks was that the JVM did not apply just-in-time compilation (JIT) during microbenchmark execution, despite the fact that JIT was enabled in the JVM. Hence, prediction on the basis of these benchmarking must account for the "speedup" effect of JIT optimisations that are applied during the execution of "normal" applications.

Some steps in Fig. 5 (such as "Get file hash") make heavy use of methods provided by the Java API. To benchmark such API calls and to investigate whether their execution durations have parametric dependencies on method input parameters, we have *manually* created and run microbenchmarks that vary the size of the hashed files, algorithm type etc. Due to aforementioned space limitations, we cannot describe the results

	ALOAD	ARRAYLENGTH	ANEWARRAY	BALOAD	ICONST_0	IF_ICMPLT	IINC	ILOAD	ISTORE
P1	1.95	5.47	220.42	6.98	1.41	5.08	3.10	3.21	3.45
P2	3.77	2.01	178.79	3.49	1.68	4.30	3.01	2.10	3.05

Fig. 6. Excerpt of microbenchmark results for platforms P1 and P2: instruction durations [ns]

of API microbenchmarks here. To simplify working with the Java API, we are currently working towards automating the benchmarking of Java API methods.

4.3 Performance Prediction

After counting and benchmarking have been performed, our approach predicts the execution durations of the activities in Fig. 5. From these individual execution durations, response time of the entire service will be predicted. These prediction results are platform-specific because underlying bytecode timings are platform-specific.

First, for source platform P1, we predict the duration of compressing a text file (randomly chosen) with a size of 25 KB on the basis of bytecode microbenchmarks, yielding 1605 ms. Then, we measure the duration of compressing that file on P1 (124 ms) and calculate the ratio $R := \frac{bytecode-based\ \ prediction}{measurement}$. R is a constant, algorithm-specific, multiplicative factor which quantifies the JIT speedup and also runtime effects, i.e. effects that are not captured by microbenchmarks (e.g. reuse of memory by the compression algorithm). R's value on P1 for the compression algorithm was 12.9.

Hence, R serves to *calibrate* our prediction. In our case study, R proved to be algorithm-specific, but platform-independent and also valid for any input to the considered algorithm. Using R obtained on platform P1, we have predicted the compression of the same 25 KB text file for its relocation to platform P2: 113 ms were predicted, and 121 ms were measured (note that to obtain the prediction, the compression algorithm was neither measured nor executed on P2 !). We then used the *same* calibration factor R for predicting the duration of compressing 9 additional, different files on platform P2 (these files varied in contents, size and achievable compression rate). For each of these 9 files, the prediction accuracy was within 10% (except one outlier which was still predicted with 30% accuracy).

This shows that the calibration factor R is input-agnostic. Also, R can be easily obtained in the presented relocation scenario because an instance of the application is already running on the "source" execution platform P1 (note that the prediction of performance on P1 is only needed for relocation, as the real performance on P1 is available by measuring the already deployed application).

The performance of the hashing action in Fig. 5 was predicted by benchmarking the underlying Java API calls, whereby a linear parametric dependency on the size of input data was discovered. The JIT was carried out by the JVM during benchmarking of these API calls, which means that R does not need to express the JIT speedup. For example, hashing 36747 bytes of data on P2 was predicted to 1.71 ms while 1.69 ms were measured, i.e. with $< 2\%$ error. Similar accuracy for predicting hashing duration is obtained for other file sizes and types.

The total upload process for the above 25KB text file on P2 was predicted to take 115 ms, and 123 ms were measured. Upload of 37 KB JPEG (i.e. already compressed)

file took 1.82 ms, while 1.79 ms were predicted. For all files used in our case study, the prediction of the entire upload process for one file had an average deviation of $< 15\%$.

Ultimately, our bytecode-based prediction methodology can deal with all four factors discussed in Sec. 1: *execution platform* (as demonstrated by relocation from P1 to P2), *runtime usage context* (as demonstrated by the prediction for different input files), *architecture* of the software system (as we predict for individual component services and not a monolithic application), and the *implementation* of the components (as our predictions are based on the bytecode implementation of components). From these results, we have concluded that a mix of upload files can be predicted if it is processed sequentially. However, for capturing effects such as multithreaded execution, further research is needed to study performance behaviour of concurrent requests w.r.t. bytecode execution. In the next section, we discuss the assumptions and the limitations of our approach.

5 Limitations and Assumptions

For the monitoring step, we assume that a representative workload (including input parameter values) can be provided, for example by a test driver. This workload has to be representative for both current and planned usage of the component. For running systems, this data can be obtained using runtime monitoring; otherwise, a domain expert judges which scenarios are interesting or critical, and she should select or specify the corresponding workloads ([21] provides an overview on test data generation).

To predict performance on a new (or previously unknown) execution platform, our approach does not need to run the application there, but must run the microbenchmark suite on the new platform. Hence, we assume that either this is possible for the predicting party, or that the microbenchmark results are provided by a third party (for example, by the execution platform vendor).

One of the current limitations of our approach is that it is not fully automated. For example, the parts **A** and **B** in Fig. 1 are not integrated for an automated performance prediction. Also, API calls must be measured manually to consider parametric dependencies and complicated parameter conditions; hence, only a limited number of API calls can be supported realistically.

In the data gathering step of our approach, asynchronous communication (e.g. message-based information exchange) is not supported by the used logging framework. Hence, if there is asynchronous communication inside the component under investigation, monitored results will be misleading. This limitation will be addressed in next versions of our implementation.

To support the black-box component principle (end-users do not have to deal with code), monitoring should be performed in an automated way. In general, collecting dozens of metrics for input and output data is not justified by the requirements of our approach. At the moment, we assume that all input and output data is composed from primitive types or general collection types like `List`. In more elaborate cases, a domain expert can specify important data characteristics manually to improve the monitoring data base.

In the machine learning step, heavily disturbed results (i.e. those having causes not visible at the interface-level) lead to decreased convergence speed and smaller probability of finding a good solution.

6 Conclusions

In this paper, we have presented a performance prediction approach supporting black-box software components by creating platform-independent parametric performance models. The approach requires no a-priori knowledge on the components under investigation. By explicitly considering parameters in the performance model, the approach enables prediction for different execution platforms, different usage contexts, and changing assembly contexts.

In the described approach, bytecode is monitored at runtime to count executed bytecode instructions and method calls, and also for gathering data information at component interface level to create the parametric performance model. Then, bytecode instructions and methods are benchmarked to obtain their performance values for a certain platform. The advantage of separating behaviour model from platform-specific benchmarking is that the performance model must be created only *once* for a component-based application, but can be used for predicting performance for any execution platform by using platform-specific benchmark results.

We evaluated the presented approach using a case study for the Java implementation of a file-sharing application. The evaluation shows that the approach yields accurate prediction results for (i) different execution platforms and (ii) different usage contexts. In fact, the accuracy of predicting the execution duration of the entire upload process after redeployment to a new execution platform lies within 15% for all considered usage contexts (i.e. uploaded files), and even within 5% in all but three contexts. The average accuracy is therefore also very good.

For our future work, we plan to automate the entire approach and to merge bytecode counting in Step 1 of our approach with data monitoring and recording in Step 2. The manual execution of the approach took ca. five hours for the case study. Also, we plan to automate creating microbenchmarks for methods, which currently must be created by hand and also do not cover the entire Java API. We also plan to consider parameters at bytecode level both for bytecode microbenchmarks and method microbenchmarks.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Experiments in Cost Analysis of Java Bytecode. Electr. Notes Theor. Comput. Sci. 190(1), 67–83 (2007)
2. Becker, S., Happe, J., Koziolek, H.: Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R., Szyperski, C., Weck, W. (eds.) WCOP 2006 (June 2006)
3. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-Driven Performance Prediction. Journal of Systems and Software (in press, 2008) (accepted manuscript)
4. Bertolino, A., Mirandola, R.: CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 233–248. Springer, Heidelberg (2004)
5. Binder, W., Hulaas, J.: Flexible and Efficient Measurement of Dynamic Bytecode Metrics. In: GPCE 2006, pp. 171–180. ACM, New York (2006)
6. Binder, W., Hulaas, J.: Using Bytecode Instruction Counting as Portable CPU Consumption Metric. Electr. Notes Theor. Comput. Sci. 153(2), 57–77 (2006)

7. Bondarev, E., de With, P., Chaudron, M., Musken, J.: Modelling of Input- Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In: Proceedings of the 31th EUROMICRO Conference (EUROMICRO 2005) (2005)
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. Adaptable and Extensible Component Systems (2002)
9. Courtois, M., Woodside, C.M.: Using regression splines for software performance analysis. In: WOSP 2000, Ottawa, Canada, September 2000, pp. 105–114. ACM, New York (2000)
10. Donnell, J.: Java Performance Profiling using the VTune Performance Analyzer (Retrieved 2007-01-18) (2004)
11. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
12. Harman, M.: The Current State and Future of Search Based Software Engineering. In: Future of Software Engineering, 2007. FOSE 2007, May 23-25, 2007, pp. 342–357 (2007)
13. Herder, C., Dujmovic, J.J.: Frequency Analysis and Timing of Java Bytecodes. Technical report, Computer Science Department, San Francisco State University, Technical Report SFSU-CS-TR-00.02 (2000)
14. Hrischuk, C.E., Murray Woodside, C., Rolia, J.A.: Trace-based load characterization for generating performance software models. IEEE Transactions Software Engineering 25(1), 122–135 (1999)
15. Hu, E.Y.-S., Wellings, A.J., Bernat, G.: Deriving Java Virtual Machine Timing Models for PortableWorst-Case Execution Time Analysis. In: Meersman, R., Tari, Z. (eds.) OTM-WS 2003. LNCS, vol. 2889, pp. 411–424. Springer, Heidelberg (2003)
16. Israr, T., Woodside, M., Franks, G.: Interaction tree algorithms to extract effective architecture and layered performance models from traces. Journal of Systems and Software, 5th International Workshop on Software and Performance 80(4), 474–492 (2007)
17. Koza, J.R.: Genetic Programming – On the Programming of Computers by Means of Natural Selection, 3rd edn. MIT Press, Cambridge (1993)
18. Kuperberg, M., Becker, S.: Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs. In: Reussner, R., Czyperski, C., Weck, W. (eds.) WCOP 2007 (July 2007)
19. Kuperberg, M., Krogmann, M., Reussner, R.: ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In: BYTECODE 2008 (2008)
20. Lambert, J., Power, J.F.: Platform Independent Timing of Java Virtual Machine Bytecode Instructions. In: Workshop on Quantitative Aspects of Programming Languages, Budapest, Hungary, March 29-30 (2008)
21. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
22. Meffert, K.: JGAP - Java Genetic Algorithms Package (last retrieved: 2008-03-18), http://jgap.sourceforge.net/
23. Meyerhöfer, M., Meyer-Wegener, K.: Estimating Non-functional Properties of Component-based Software Based on Resource Consumption. Electr. Notes Theor. Comput. Sci. 114, 25–45 (2005)
24. Smith, C.U., Williams, L.G.: Performance Engineering Evaluation of Object- Oriented Systems with SPEED. In: Marie, R., Plateau, B., Calzarossa, M.C., Rubino, G.J. (eds.) TOOLS 1997. LNCS, vol. 1245, Springer, Heidelberg (1997)
25. Zhang, X., Seltzer, M.: HBench:Java: an application-specific benchmarking framework for Java virtual machines. In: JAVA 2000: Proceedings of the ACM 2000 conference on Java Grande, pp. 62–70. ACM Press, New York (2000)

Validating Access Control Configurations in J2EE Applications

Lianshan Sun, Gang Huang*, and Hong Mei

Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
School of Electronics Engineering and Computer Science, Peking University, 100871, China
sunlsh@sei.pku.edu.cn, huanggang@sei.pku.edu.cn, meih@pku.edu.cn

Abstract. Access control is a means to achieve information security. When we build large-scale systems based on commercial component middleware platforms, such as those compliant to J2EE, a usual way to enforce access control is to define Access Control Configurations (ACCs) for components in a declarative manner. These ACCs can be enforced by the J2EE security service to grant or deny access requests to components. However, it is difficult for the developers to define correct ACCs according to complex and sometimes ambiguous real-world access control requirements. Faults of ACCs in large-scale J2EE applications may inevitably occur due to various reasons, for example ad hoc mistakes of the developers. This paper identifies three kinds of faults specific to ACCs of J2EE applications as incompleteness, inconsistency, and redundancy, presents validation algorithms for identifying these faults according to access control requirements, illustrates these faults and the validation algorithms with an online bank application.

Keywords: J2EE Security, Access Control, Validation.

1 Introduction

Component-Based Software Engineering is focused on developing software intensive systems from pre-fabricated and reusable components [1]. It has achieved great success in software engineering highlighted by the proliferation of commercial component middleware platforms, such as those compliant to J2EE. Although commercial middleware platforms are valuable in constructing large-scale component based systems, there are still concerns on the quality of the systems built on them [2, 3, 4]. This paper is on how to assure the quality of J2EE access control configurations.

Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied [5]. It is a widely used means to achieve information security, especially confidentiality and integrity [6]. In the J2EE, Role-Based Access Control (RBAC) model [7] is adopted and its implementation mechanism is provided as the J2EE security service [8, 9]. In RBAC, the users of a system are assigned one or more roles. Each role has a set of rights. Rights of a user are the union of sets of rights of all roles that

* Corresponding author.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 64–79, 2008.

user currently belongs to [7]. From the perspective of business, a role represents a class of users or user groups in organizations according to their function, seniority, and context in which they are active [10]. Consequently, the Access Control Requirements (ACRs) of an organization should specify what roles can do what business operations on resources under some conditions. From the perspective of technology, a role is actually a set of method permissions, each of which describes the right of performing a method of a component. By adopting J2EE security service, access control concern can be well separated from the functional ones. The developers can focus on functionalities first and then implementing ACRs via defining proper Access Control Configurations (ACCs) for components in a declarative manner, i.e. assigning method permissions to each role identified in organizations.

However, due to the gap between ACCs and ACRs, the complex structure of J2EE applications, the knowledge and intelligence limits of human being, and many other reasons, the ACCs in J2EE applications may contain various faults in terms of ACRs. For the J2EE applications, these faults may induce unexpected risks, for example information leakage, denial of service, and low performance. Therefore, it is necessary to validate ACCs against ACRs to identify possible faults.

ACCs are actually access control policies (ACPs) [5] in the J2EE applications. Current researches focus on verifying general ACPs against some formally-specified properties [11, 12, 13], testing ACPs [14], but often neglect the influence of how and where the ACPs are hooked into applications. In J2EE applications, component containers are responsible for hooking ACPs at different places. Multiple ACCs hooked at different component methods may conflict or coordinate in implementing one ACR. In that, except for the faults possibly occurred in general ACPs, ACCs in J2EE applications may contain some special faults. This paper identifies three kinds of faults specific to ACCs of J2EE applications: incompleteness, inconsistency, and redundancy. This paper presents validation algorithms to identify these faults according to ACRs automatically. By using the algorithms, analysts are relieved of the hard and challenging work of manually validating voluminous ACCs.

In the rest of this paper, we first present how the way that ACPs are hooked into J2EE applications leads to faults specific to ACCs and illustrate three kinds of faults in an online bank application, which is used as the case throughout the paper. We then introduce the validation algorithms in details and demonstrate them in the online bank application. Later, we discuss our work in a bigger picture. Finally, we introduce related work, conclude the paper and present the outlook on the future.

2 Faults Specific to ACCs in J2EE Applications

As shown in figure 1, in a modern access control system, requests to a resource are intercepted at a policy enforcement point (PEP) and forwarded to a policy decision point (PDP). The PDP then evaluates the requests against ACPs stored somewhere, and returns the evaluation results, for example “allow” or “deny” to the PEP, which will finally allow or deny the access requests to resource. In J2EE applications, the protected resources are component methods; and component container serves as PEP; J2EE security service serves as PDP, ACPs are actually ACCs defined in deployment descriptor of components.

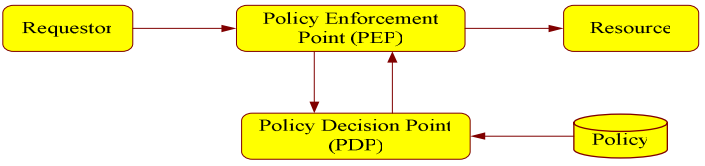


Fig. 1. Conceptual Architecture of Access Control System (reprinted and simplified from [15])

Researchers have recognized that various faults may occur in ACPs and have also provided verification, validation and testing techniques to identify these faults [11, 12, 13, 14]. For example, ACPs may violate some organization specific authorization constraints; multiple ACPs defined for protecting one resource may allow or deny some users simultaneously due to the inappropriate conflict-resolution algorithms. The previous work is focused on faults of general ACPs, but the influence of how and where ACPs are hooked into applications is always neglected.

However, faults in ACPs of J2EE applications, i.e. ACCs, may come from not only the policies themselves but also where policies are hooked into the applications, i.e. where the PEPs are placed.

J2EE applications are composed of interconnected components across multiple tiers and running in various containers, which are responsible for hooking J2EE security service to specific components or component methods [8]. In J2EE applications, ACCs are defined in deployment descriptors of components. Some ACCs tell how a component can acquire a principal on behalf of a user who is playing some roles and which roles are allowed for a component method. When no ACCs are defined for a component method, any access to it will be allowed.

In J2EE applications, users can perform a business operation by calling one or more component methods implementing it directly or indirectly via other component methods. All these component methods interconnect with each other via invocation relationship and form one or more access paths to the business operation. In that, implementing ACRs for protecting business operations on resources is to define ACCs for some component methods on access paths to block all unauthorized users while to allow all authorized users.

The developers can configure any components on the access paths on their discretion, i.e. setting PEPs to hook ACPs. For example, some applications may only define ACCs for web tier components while other applications may define ACCs for even all component methods on the access paths. On the one hand, multiple ACCs can be defined for implementing one ACR. On the other hand, multiple ACRs can be implemented by multiple ACCs.

However, discretion of the developers is not always reliable and various faults may be introduced, especially in large-scale component-based systems. In short, multiple ACCs defined for different component methods (whose containers serve as PEPs) may conflict or coordinate with each other in terms of ACRs. Defining or maintaining some ACCs may incur unexpected impact on other ACCs and consequently the security of the whole system. This means that the J2EE applications may include some specific faults in ACCs. We identify three kinds of faults as follows. They are incompleteness, inconsistency, and redundancy.

Incompleteness means that the developers may leave some access paths to a business operation without control for accidental mistakes. In this case, unauthorized users may access the business operation via the security hole.

Sometimes, multiple components on one access path to a business operation are configured for implementing an ACR. In fact, when the developers define ACCs to allow some roles to access a component method, any component methods called by the method should be accessible for the role.

Inconsistency of ACCs means that some roles may be allowed by some ACCs while denied by others to access a business operation. That is, the business operation may be not available for some legitimate users.

Sometimes, the developers deliberately define configurations for multiple components to achieve better security [16]. However, these ACCs may not always necessary for protecting one business operation, i.e. implementing an ACR.

Redundancy means that the developers may define redundant ACCs to implement an ACR. The redundancy may lead to unnecessary performance overhead and sometimes may introduce potential security risks by allowing redundant roles to access component methods. Redundant roles allowed to resources will violate the least privilege principle (no one shall have more privileges than needed for performing their duties) [17] and thus will lead to potential security risks.

In essence, these faults originate from the mismatch between resources to be protected in organization -- business operations, and resources actually protected in software applications -- component methods. To identify these faults, it is necessary for the developers to have all the knowledge of the structure of J2EE applications. When the scale of J2EE applications increases, the developers will need an automatic tool to perform the validation than to do it manually.

2.1 An Illustrative Example

To explain the possible faults of ACCs more clearly and to demonstrate our algorithms in the next section, we build a new online bank application on top of the duke's bank application, which is used by an imaginary bank--the duke's bank and is developed by Sun Microsystems™ to exemplify J2EE technologies [8]. The new online bank application is still used by the duke's bank to support new business transactions and ACRs as follows.

The duke's bank divides its customers into two roles, including *VIPCustomer* and *NormCustomer*, and supports three new business operations, including B_1 (transferring funds across banks), B_2 (transferring funds within the duke's bank), and B_3 (querying histories of transactions). There are three ACRs. First, customers in the role *VIPCustomer* can transfer funds between accounts across banks or within the duke's bank. Second, customers in the role *Normcustomer* can only transfer funds between accounts within the duke's bank. Third, all customers can query transactions of their accounts in the duke's bank.

Some components in the online bank application and the interactions between them are shown in fig. 2. To transfer funds between accounts within the duke's bank, customers in the role *VIPCustomer* and *NormCustomer* can call the component method *TxCtrl.internalTrans* via the web tier component: the *VIPInterface* and the *NormInterface* respectively.

To transfer funds between accounts across banks, customers in the role *VIPCustomer* can call the component methods *ToBank-1.Transfunds* or *ToBank-n.Transfunds* via the *VIPInterface* and consequently the *TxCtrl.extTrans.* Multiple components, such as *ToBank-1,* and *ToBank-n,* are introduced to conform to regulations of different banks. In both cases, *TxLogger.Log* is called to record the information of transactions for future retrieval.

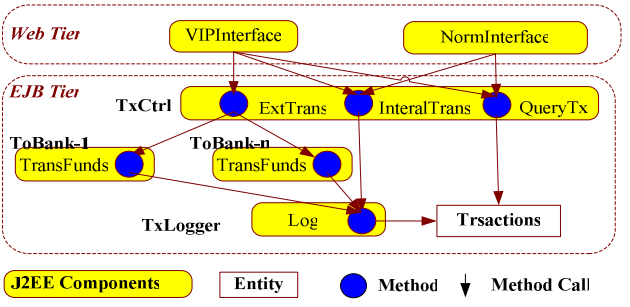


Fig. 2. Fragments of the Online Bank Application

To query histories of transactions of an account in the duke's bank, customers in both roles *VIPCustomer* and *NormCustomer* can call the component method *TxCtrl.queryTx* via two web tier components the *VIPInterface* and the *NormInterface* respectively.

Currently, J2EE applications use the configuration file *Web.xml* and *ejb-jar.xml* to specify several aspects of an application, such as the dependencies between components, transactions, and security policy. The ACCs are declared within the elements <*Security-Constraints*> in *web.xml* or the elements <*method-permission*> in *ejb-jar.xml*. A possible implementation of the three ACRs of the online bank application is as follows.

```
<!--Configurations in WEB.xml -->
1.  <security-constraint>
2.   <web-resource-collection>
3.    <web-resource-name>VIPInterface</web-resource-name>
4.    <url-pattern>/VIP/*</url-pattern>
5.   </web-resource-collection>
6.   <auth-constraint>
7.      <role-name> VIPCustomer </role-name>
8.   </auth-constraint>
9.  </security-constraint>
10. <security-constraint>
11.  <web-resource-collection>
12.   <web-resource-name>NormInterface</web-resource-name>
13.   <url-pattern>/Norm/*</url-pattern>
14.  </web-resource-collection>
15.  <auth-constraint>
16.    <role-name> VIPCustomer,NormCustomer  </role-name>
17.  </auth-constraint>
```

```
18. </security-constraint>
19. <login-config>
20.   <auth-method> BASIC </auth-method>
21. </login-config>
<!--Configurations in ejb-jar.xml -->
22. <method-permission>
23.  <role-name> VIPCustomer </role-name>
24.  <method>
25.    <ejb-name> TxCtrl </ejb-name>
26.    <method-name>*</method-name>
27.  </method>
28.  <method>
29.    <ejb-name> ToBank-1 </ejb-name>
30.    <method-name> * </method-name>
31.  </method>
32.  <method>
33.    <ejb-name> ToBank-n </ejb-name>
34.    <method-name> * </method-name>
35.  </method>
36. </method-permission>
37. <method-permission>
38.   <role-name> NormCustomer </role-name>
39.   <method>
40.     <ejb-name> TxCtrl </ejb-name>
41.     <method-name> * </method-name>
42.   </method>
43.   <method>
44.     <ejb-name> TxLogger </ejb-name>
45.     <method-name> * </method-name>
46.   </method>
47. </method-permission>
```

List 1. An Example of Access Control Configurations of the Online Bank Application

List 1 is fragments of the deployment descriptor of the online bank application. It configures roles allowed to access each component method in figure 1. Lines 1-18 allow customers in the role *VIPCustomer* to access *VIPInterface* and *NormInterface*; and lines 22-36 allow them to access all methods of the component *TxCtrl*, *ToBank-1*, and *To-Bank-n*. In this way, Customers in the role *VIPCustomer* are assigned with sufficient privileges for perform all three business operations B_1, B_2, and B_3.

Lines 10-18 allow customers in the role *NormCustomer* to access the web tier component *NormInterface*; and lines 37-47 allow them access all methods of the components *TxCtrl* and *TxLogger*. In this way, customers in the role *NormCustomer* are assigned with sufficient privileges to perform business operations B_2 and B_3.

Unfortunately, ACCs defined in List 1 do not work as expected. Some faults are buried in them. For example, the ACC defined for *TxLogger.Log* at lines 38, 43-46 allow only the role *NormCustomer* to access the component method *TxLogger.Log*. It blocks the requests from the components *ToBank-1* and *ToBank-n*, which are on behalf of the role *VIPCustomer* to transfer funds from the duke's bank to other banks. That is, ACCs in List 1 are inconsistent.

Furthermore, the ACC defined for *TxCtrl.extTrans* at lines 23-27 is redundant because any request to it issued by customers not in the role *VIPCustomer* will be denied by the ACCs defined for *ToBank-1* and *ToBank-2*. This redundant ACC leads to unnecessary performance overhead.

In addition, the ACCs defined for *TxCtrl.extTrans* at lines 38-42 allow the role *NormCustomer* to access all methods of *TxCtrl* for simplicity in defining ACCs. Although the ACCs can satisfy all three ACRs, a redundant ACC allows the role *NormCustomer* to access the method *TxCtrl.extTrans*. In that, malicious customers in the role *NormCustomer* can inject program to call the method *TxCtrl.extTrans.*

At last, lines 1-18 define ACCs for both web tier components, which are user-interfaces of all access paths to all business operations. In that, ACCs defined in List 1 are complete.

As discussed above, it is difficult for the developers to manually and completely identify and cure all faults of ACCs, or to justify their correctness even in such a simple example. We argue that in large-scale component based systems, which may include hundreds of business operations and thousands of components methods to be configured, the faults of ACCs are often inevitable and very difficult to be manually identified and cured by the developers.

3 Validation: Algorithms and Demonstration

Validation is an activity of software quality assurance [18]. In this section, we present validation algorithms for three kinds of faults specific to ACCs of J2EE applications. This section first shows the prerequisites for validating ACCs against ACRs, then introduces the algorithms for identifying different kinds of faults of ACCs, and finally demonstrates the algorithms with the online bank application.

3.1 Prerequisites for Validation

To enable the validation of the ACCs of J2EE applications against ACRs, some prerequisites are needed.

First, the ACRs as the foundation of validation should certainly be available. ACRs specify what roles can access what business operations on which resources.

In order to access the J2EE applications properly, users need to play some legitimate roles, each of which is a set of privileges that a user can possess to perform his/her duties. A role can be denoted as r. The set $R= \{r_1,r_2,\ldots r_m, m>0\}$ denotes all roles of J2EE applications and R_s denotes a subset of R. The business operations on resources allowed for a role r are actually user-visible functions, which should have been captured as functional requirements of the J2EE applications. An ACR can be specified as a pair of the allowed roles R_s and a business operation f, i.e., $acr = \{R_s, f\}$. All ACRs in a J2EE application can be denoted as $ACRs= \{acr_1,\ldots,acr_n, n>0\}$. Some requirements specification methods have already contain such requirements though they are defined in other formats [10, 19]. We believe they can be easily derived from these requirements specifications.

Second, the ACCs of J2EE applications as the objects to be validated should also be available. An ACC can be denoted as $acc = \{R_s\ m\}$, where m is the component

method that can be accessed by the roles in R_s. In this paper, one *acc* only controls the access to one method *acc.m*. A section of <*method-permission*> in the deployment descriptor of J2EE applications actually defines one or more ACCs. All ACCs in a J2EE application form a set $ACCs = \{acc_1,\ldots,acc_k, k>0\}$, which can be obtained easily from the deployment descriptor of J2EE applications. For a component method *m*, the ACC defined for it can be referred as *m.acc*. A traversal on *ACCs* of the J2EE application is enough to get the *m.acc* from *m*.

For an *acr*, a user may perform the business operation *acr.f* via multiple access paths. Any ACCs defined for methods on these paths contribute to the *acr*. To validate ACCs against the *acr*, all access paths to *acr.f* should be constructed from the architecture of J2EE applications including components interconnected with each other via invocation relationship. The architecture of J2EE applications should be available for validating ACCs. The architecture of J2EE applications can be obtained with various means and in different settings. For example, when we develop the application with only in-house components, the architecture is actually an artifact of design. When we develop the application with the Off-The-Shelf (OTS) components, dependencies between OTS component interfaces are available. By viewing the dependencies between two component interfaces as roughly the invocation relations among all methods of two interfaces, an inaccurate architecture is also available. Sometimes, this inaccurate architecture can be adjusted with information mined from source codes by algorithms of source code analysis [20] or by information reflected from the runtime system [21].

With ACRs, ACCs and the architecture of J2EE applications as inputs, the developers can start to validate ACCs against ACRs, that is, to check their completeness, consistency, and redundancy in implementing ACRs.

To validate ACCs against a given *acr*, it is necessary to find out all the ACCs implementing the *acr*, that is, to identify all methods involved in using the business operation *acr.f* because it is easy to find out the ACC defined for a component method. These methods can be classified into two categories. One category includes methods implementing *acr.f*. We call methods in this category as target methods of *acr.f*. We denote target methods of the business operation *f* as $TM(f)= \{m_1,\ldots m_p, p>0\}$. *TM (f)* can be acquired automatically when traceability links between functional requirements and component methods are available [22], or can be acquired from manual analysis by the developers when traceability links are not available.

The other category includes methods directly or indirectly calling or called by target methods of *acr.f*. To acquire methods in the second category, we define two functions. One is *path(tm)*, which will be used to compute access paths to a target method *tm* of a business operation *f*; the other is *called(tm)*, which will be used to compute component methods called by a target method *tm*. The functions *path (tm)* and *called(tm)* traverse the architecture of J2EE applications to extract access paths to *tm* and component methods called by *tm*. For an *acr*, by applying *path ()* and *called()* to each method *tm* in *TM (acr.f)*, we can easily acquire all component methods related to the business operation *acr.f*. Consequently, all ACCs related to the *acr* can be acquired by traversing the set *ACCs*.

3.2 Analyzing Completeness

For an *acr*, the business operation *acr.f* is protected completely only when each access path to it is controlled by at least one *acc*. In this sense, we can say that the ACCs for an *acr* are complete. The following algorithm traverses all access paths to *acr.f* and identifies all uncontrolled paths of them as the set *UncontrolledPath*.

```
1.  Algorithm Completeness( acr )
2.  Begin
3.    UnControlledPath = Φ ;
4.    For each m in TM(acr.f)
5.      For each path in path(m) ;
6.       For m = path.head to path.tail
7.       If (m.acc!= NIL )
8.         Go to next path; // the path is under control
9.       End
10.      UncontrolledPath += { path };
11.     End
12.   End
13. End
```

In the above algorithm, the statement executed most frequently is line 7. It computes *m.acc* from a method *m* by performing a traversal on *ACCs* of the J2EE application. We assume that the number of all component methods is *N* and the number of elements of *ACCs* is *M,* which is certainly less than *N*. We assume that the average number of component methods implementing a business operation is *C*. In addition, the average number of component methods called by one component method is two. In that, the computation complexity of our algorithms can be denoted by the formula $T(N) = O(C \times N \times log_2(N))$, where $log_2(N)$ is the average length of an access path without considering possible loop.

3.3 Analyzing Consistency

For an *acr*, in order to allow all legitimate roles to access the business operation *acr.f* properly, all ACCs defined for component methods on all access paths to target methods of *acr.f* and component methods called by them directly or indirectly should work harmoniously. On the one hand, when the developers define ACCs to allow some roles to access component methods calling directly or indirectly the target methods of *acr.f*, they actually want to allow these roles to access *acr.f*. In that, roles allowed by each of these ACCs should be a subset of $acr.R_s$. If all roles allowed by these ACCs together form a real subset of $acr.R_s$, but not exactly the $acr.R_s$, these ACCs will block some legitimate roles for *acr.f*.

On the other hand, when the developers define an *acc* for a component method called by the target methods of *acr.f*, they have to ensure $acr.R_s$ is a subset of $acc.R_s$ to allow that users in all legitimate roles can access the *acr.f* properly. In general, for the two endpoints of an invocation relationship, roles allowed for the caller should be allowed for the callee. In this way, when the caller is accessed by legitimate roles, it can in turn call its callee to respond the requests without authorization error. The following algorithm identifies all inconsistent candidates of ACCs of an *acr* as the set *InConsistent*, and all blocked legitimate roles as the set *Role-Overlooked*.

```
1.  Algorithm Consistency(acr)
2.  Begin
3.   InConsistent=Φ ; roles = Φ ;
4.   For each tm in TM(acr.f)
5.    For each path in path(tm)
6.      preacc = NIL;
7.      For m = path.head to path.tail
8.        If( m.acc == NIL )
9.          break;
10.       If m.acc.roles ⊄ acr.roles
11.         InConsistent += {m.acc };
12.       roles += m.acc.roles;
13.     End
14.     If(preacc.roles ⊄ acc.roles)
15.        InConsistent +={preacc,acc};
16.     preacc = m.acc;
17.   End
18.   If(roles∩acr.roles)≠acr.roles
19.     Role-overlooked = acr.roles-roles;
20.   For each chain in Called(tm)
21.     For m= chain.head to chain.tail
22.      preacc = NIL;
23.      If(acr.roles ⊄m.acc.roles);
24.        InConsistent += { m.acc };
25.      End
26.      If(preacc.roles ⊄ acc.roles)
27.        InConsistent+={preacc,m.acc};
28.      preacc = m.acc;
29.   End
30. End
```

Similar to the algorithm *Completeness(acr)*, the computation complexity of the algorithm *Consistency(acr)* is also $T(N) = O(C \times N \times log_2(N))$.

3.4 Analyzing Redundancy

Multiple ACCs can be defined to implement one *acr*. Sometimes, removing some of ACCs will not influence the degree to which the *acr* is satisfied. In this sense, we can say these ACCs are redundant for the *acr*. Sometimes, it is important to remove redundant ACCs because they may induce unnecessary performance overhead or even may introduce potential security risks by allowing roles to access methods not required for their duties.

According to an *acr*, only roles in $acr.R_s$ are allowed to access the business operation *acr.f* via its target methods. If two ACCs defined for a target method of *acr.f* or its callers allow same set of roles to access *acr.f*, then they are candidates of redundant ACCs for the *acr*. One of them could be removed. If the set of roles allowed by an ACC defined for a target method or its callers is not a sub set of $acr.R_s$, then some redundant roles are allowed to access *acr.f*.

Any invocations from target methods of *acr.f* to other component methods are actually on behalf of one or more roles in $acr.R_s$. In that, if an ACC defined for component

methods, which is called directly or indirectly by target methods of *acr.f* only allows $acr.R_s$, we can say that it is a candidate of redundant ACC.

The following algorithm identifies candidates of redundant ACCs for an *acr* as the set *RedundantACC*. In particular, it identifies some redundant roles allowed for *acr.f* as *RedundantRole*.

```
1.  Algorithm Redundancy(acr)
2.  Begin
3.   RedundantACC = Φ ; roles = Φ ;
4.   RedundantRole = Φ ;
5.   For each tm in TM(acr.f)
6.    For each path in path(tm)
7.      preacc = NIL;
8.      For m = path.head to path.tail
9.        If((acc=m.acc))== NIL )
10.         break;
11.       If(acc.roles<=preacc.roles)
12.           RedundantACC+={preacc};
13.       roles += acc.roles;
14.     End
15.     preacc = m.acc;
16.   End
17.   If(roles∩acr.roles)≠roles
18.     RedundantRole += roles - acr.roles;
19.   For each chain in Called(tm)
20.    For m= chain.head to chain.tail
21.     If(m.acc.roles == acr.roles);
22.         RedundantACC += {m.acc};
23.    End
24.   End
25. End
```

Similar to the algorithm *Completeness(acr)*, the computation complexity of the algorithm *Redundancy(acr)* is also $T(N) = O(C \times N \times log_2(N))$.

It is noteworthy that, a candidate of redundant ACC can be removed only when it is unnecessary for all access paths passing the component method it protects. For a redundant ACC candidate c_1 of an ACR a_1, when c_1 does not appear in the set of *RedundantACC* of other ACRs but *c1.m* is calling or called directly or indirectly by target methods of other ACRs, we can say c_1 is a fake redundant ACC for *a1*. In fact, we can only identify part of fake redundant ACCs. Sometimes, redundant ACCs may come from deliberate design decisions. For example, the developers may decide that components at each tier of J2EE applications should be configured with access control capability. When necessary, the developers need to filter out the fake redundant ACCs manually.

3.5 Validating the Online Bank Application

One *acr* of the online bank application is $\{VIPCustomer, B_1\}$. It denotes that only customers in the role *VIPCustomer* can transfer funds from the duke's bank to other banks. We can apply the above algorithms to validate whether the ACCs defined in

List 1 are complete, consistent and redundant or not against the *acr*. Target methods of B_1 include *ToBank-1.TransFunds* and *ToBank-n.TransFunds*. Access paths passing them include *{VIPInterface, TxCtrl.extTrans, ToBank-1.TransFunds}* and *{VIPInterface, TxCtrl.extTrans, ToBank-n.TransFunds }*. Component methods called by them include only *{ TxLogger.Log}*

According to the algorithm *Completeness(acr)*, List 1 defines ACCs in lines 23, and 28-35 for *ToBank-1.TransFunds* and *ToBank-n.TransFunds* to allow customers in the role *VIPCustomer*. In that, ACCs defined in List 1 are complete for protecting the business operation B_1.

According to the algorithm *Consistency(acr)*, the ACC defined in lines 1-9 allows the role *VIPCustomer* to access *VIPInterface*. In turn, roles allowed for *TxCtrl.extTrans* include *VIPCustomer* and *NormCustomer* according to ACCs defined in lines 23-27 and lines 38-42 in List 1. Obviously, roles allowed by the ACC for *TxCtrl.extTrans* are not subsets of roles allowed by ACCs for both *ToBank-1.TransFunds,* and *ToBank-n.TransFunds.* In that, these ACCs are inconsistent in implementing the *acr*. In addition, roles allowed by the ACC defined for *TxLogger.Log* in lines 43-47 include only *NormCustomer*, which is not a superset of roles allowed by the ACCs defined for both *ToBank-1.TransFunds,* and *ToBank-n.TransFunds.* An authorization error will occur when customers in the role *VIPCusomter* transfer funds across banks. In that, these ACC are inconsistent too in implementing the *acr*.

According to the algorithm *Redundancy(acr)*, the ACC defined by lines 1-9, 24-27 is redundant relative to ACCs defined by lines 28-35. Furthermore, the ACC defined by lines 39-42 allows redundant role *NormCustomer* for the component method *TxCtrl.extTrans.* However, the ACC defined by lines 1-9 will not be removed when a design decision is that each tier of J2EE applications should be controlled independently. At last, the *acc* for *TxCtrl.extTrans* is removed.

4 Discussion

The main contributions of this paper include the identification of three types of faults specific to ACCs in J2EE applications and the definition of validation algorithms for each of them. These algorithms are somewhat straightforward because we focus on the feasibility of the validation. In real settings, many concerns need to be considered and the validation engine may be much more complex than the algorithms presented. For example, validation engine needs to clarify how the identified candidates of faults can be cured and how the validation engine can acquire necessary inputs when activated at different phases of development.

In J2EE applications, when a component method m_1 with the principal acquired from its caller fails to call its downstream components due to authorization error. That is, some roles allowed by $m_1.acc$ are not allowed by the ACCs of the component methods called by m_1. This is actually an instance of inconsistency fault, which can be identified by the algorithm *Consistency()*. Recognizing the inconsistency, the developers can use the <*run-as*> element of the deployment descriptor to assign a role *r* to the component *c* with the method m_1 to allow it to invoke its downstream components properly. In this case, the developers cure the inconsistency of ACCs by employing

the <*run-as*> element. We can develop an algorithm for checking whether the inconsistent ACCs identified by the algorithm *Consistency()* have been cured by the <*run-as*> element in deployment descriptor or not. In fact, many reasons can lead to faults of ACCs and consequently various design decisions may be involved in fixing the faults identified. For example, both employing the <*run-as*> element and changing roles allowed to access methods can cure some faults of inconsistency. We argue it is possible to automatically recommend remedies for fixing these faults. However, it is very difficult, if no possible, to ensure the correctness of the remedies.

In real applications, users may be required to play multiple roles at the same time for performing some business operations. Due to the inability of J2EE deployment descriptor in specifying multiple roles that have to be activated simultaneously, the developers may deliberately define inconsistent ACCs to rule out users not activating some required roles step by step. However, in this case, security risks are inevitably introduced. A more secure solution is to define a new role which subsumes the privileges owned by all required roles. In this case, users can activate only the new role for performing the business operations.

The ACCs specified as the deployment descriptor of J2EE applications can be developed and maintained at different phases of component-based software engineering. Accordingly, our algorithms are applicable at different phases where the ACRs and ACCs are available. In particular, the architecture of J2EE applications can be acquired at different phases from different sources, such as from detailed design model at design phases, and then enriched by source code analysis [20] at deployment time and by reflected information at runtime [21]. At different phases, our algorithms can identify the faults of ACCs in terms of ACRs an automatic manner.

In general, ACRs can be specified based on instances, for example location or time of requests, or based on the roles played by users. Furthermore, the developers of J2EE applications can use customized security or J2EE security to implement ACRs. In particular, they can use J2EE security in a programmatic manner or declarative manner. Algorithms in this paper validate only the declarative ACCs conforming to role-based access control model [7] against role-based ACRs. Declarative security in J2EE applications can only control access requests across component boundaries because the container of components takes the responsibility of enforcing the ACCs. The declarative security is very suitable for the integration of reusable components, which are black box for assembler. To this end, the intra-component call is not considered in our algorithms.

At last, we argue that the faults of access control configurations we identified in J2EE applications may also occur in applications based on other component middleware platforms, which adopt and implement role-based access control model, such as .NET. We can identify the faults of ACCs in those applications in a similar way as in J2EE applications.

5 Related Work

ACCs in J2EE applications are actually the role-based ACPs enforced by the underlying access control mechanism--the J2EE security service [8, 9]. ACPs have grown from simple matrices to non-trivial specifications as sophisticated configurations in

distributed system [15, 23]. The increasing complexity of these policies correspondingly demands strong quality assurance techniques.

Many researches have been done to verify whether or not the policies conforming to some properties. For example, Jajodia et al checked whether two or more ACPs grant or deny an access simultaneously and whether all access have been granted or denied [24]. Naumovich and Centonze [25, 26] evaluated the "location consistency" between function-based ACPs and data-based ACPs. Hansen et al [12] demonstrated how to use finite model checking to conformance testing between security policies expressed in form of LTL claims and their implementation in RBAC framework. Fisler et al. [13] developed a tool called Margrave that uses multi-terminal binary decision diagrams to verify user-specified properties and perform change-impact analysis on ACPs. Martin et al [14, 27] introduced idea of mutation test into verification of ACPs to assess the quality of policy properties used to verify ACPs. In addition, some organization level authorization constraints [11] need to be enforced when specifying ACPs. For example, two roles should not be allowed to access a resource. Some researchers have provided facilities to validate and test ACPs for their violations to authorization constraints [28].

However, these researches do not consider how the places where the ACPs are hooked into the software applications influence the verification of ACPs. In that, these approaches fail to identify faults originated from the correlation of multiple ACPs hooked at different places in the software applications in implementing an ACR, i.e. protecting a business operation. As an exception, Pistoia et al [29] assumed methods allowed for roles are correct and then utilize the call graph of methods in a program, which is protected by access control policies to verify whether the roles assigned to a user is insufficient or redundant for executing the program. They construct the call graph of a program by source code analysis tool DOMO, which can identify both inter-component call and intra-component call in J2EE applications.

In contrast to verifying the properties of ACPs, our algorithms validate ACPs against high-level ACRs while considering the places where the policies are hooked in J2EE applications. Taking software architecture of J2EE applications including components interconnected via method invocation relationship as input, our validation algorithms automatically evaluate whether ACCs are complete, consistent or redundant or not for implementing an ACR. Here, the ACRs are supposed to be correct and serve as the base for reasoning and adapting the ACCs. In real settings, ACRs may include some faults. When viewing the ACRs as high-level ACPs, the developers can apply existing validation and testing techniques on ACRs, for example work in [12, 13, 24, 28]. Massacci et al [30] identify and resolve the conflicts of ACRs with functional requirements.

6 Conclusion and Future Work

This paper identifies three kinds of faults specific to ACCs in J2EE applications – an instance of component based software, including incompleteness, inconsistency and redundancy. The problematic ACCs may harm the security and performance of J2EE applications. This paper presents algorithms for validating the ACCs of J2EE applications against ACRs in an automatic manner. Our algorithms utilize the architecture of

J2EE applications to validate multiple ACCs defined for different component methods in J2EE applications. Our algorithms reduce the work required for the analysts to validate the ACCs in large-scale J2EE applications, in which hundreds of business operations need to be protected and thousands of component methods need to be configured for protecting the business operations.

In the future, we plan to integrate the validation algorithms into our architecture centric development tool suite and apply it in more real J2EE applications. Furthermore, in contrast to validating ACCs against ACRs after ACCs are produced as this paper does, we plan to generate complete, consistent and non-redundant ACCs from ACRs by a model driven approach.

Acknowledgements. This work is sponsored by the National Basic Research Program (973) of China under Grant No. 2002CB312000, the High-Tech Research and Development Program (863) of China under Grant No. 2006AA01Z156, the National Natural Science Foundation of China under Grant No. 60528006 and the Fok Ying Tung Education Foundation.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison Wesley, London (2002)
2. Lan, L., Huang, G., et al.: Architecture Based Deployment of Large-Scale Component Based Systems: The Tool and Principles. In: Heineman, G., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 123–138. Springer, Heidelberg (2005)
3. Liu, Y., Gorton, I.: Performance Prediction of J2EE Applications Using Messaging Protocols. In: Heineman, G., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 123–138. Springer, Heidelberg (2005)
4. Lau, K.K., Ukis, V.: Defining and Checking Deployment Contracts for Software Components. In: Gorton, I., Heinemann, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 1–16. Springer, Heidelberg (2006)
5. Samarati, P., di Vimercati, S.C.: Access Control: Policies, Models, and Mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 137–196. Springer, Heidelberg (2001)
6. BS799-1: Information Security Management—Part 1: Code of Practice for Information Security, British Standards Institution, London (1999)
7. Sandhu, R.S., Coyne, E.J., et al.: Role-based access control models. Computer 29(2), 38–47 (1996)
8. Sun Microsystems, The Java EE 5 Tutorial, `http://java.sun.com/javaee/5/docs/`
9. Sun Microsystems, Enterprise JavaBeans Specification v3.0, `http://java.sun.com/products/ejb/`
10. Crook, R., Ince, D.C., et al.: Modelling access policies using roles in requirements engineering. J. Information & Software Technology 45(14), 979–991 (2003)
11. Ahn, G.J.: The RCL 2000 language for specifying role-based authorization constraints, Ph.D. thesis, George Mason University, Fairfax, Virginia (1999)

12. Hansen, F., Oleshchuk, V.: Conformance Checking of RBAC Policy and Its Implementation. In: Deng, R.H., Bao, F., Pang, H., Zhou, J. (eds.) ISPEC 2005. LNCS, vol. 3439, pp. 144–155. Springer, Heidelberg (2005)
13. Fisler, K.S., Krishnamurthi, L., et al.: Verification and change-impact analysis of access-control policies. In: Proc. of ICSE 2005, pp. 196–205. ACM Press, New York (2005)
14. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proc. of WWW 2007, pp. 667–676 (2007)
15. Moses, T.: eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS (February 2003)
16. Ilechko, P., Kagan, M.: Authorization concepts and solutions for J2EE applications, http://www.ibm.com/developerworks/websphere/library/techarticles/0607_ilechko/0607_ilechko.html
17. Vimercati, S., Paraboschi, S., et al.: Access control: principles and solutions. Software - Practice and Experience 33, 397–421 (2003)
18. Adrion, W.R., Branstad, M.A., et al.: Validation, Verification, and Testing of Computer Software. ACM Computing Surveys (CSUR) 14(2), 159–192 (1982)
19. Giorgini, P., Massacci, F., et al.: Modeling Security Requirements Through Ownership, Permission and Delegation. In: Proc. of ICRE 2005, pp. 167–176. IEEE Computer Society Press, Los Alamitos (2005)
20. Grove, D., Chambers, C.: A Framework for Call Graph Construction Algorithms. ACM Trans. Program. Lang. Syst. 23(6), 685–746 (2001)
21. Huang, G., Mei, H., et al.: Runtime recovery and manipulation of software architecture of component-based systems. Autom. Softw. Eng. 13(2), 257–281 (2006)
22. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Transactions on Software Engineering 27(1), 58–93 (2001)
23. Vo, H.D., Suzuki, M.: An Approach for Specifying Access Control Policy in J2EE Applications. In: Proc. of APSEC 2007, pp. 422–429 (2007)
24. Jajodia, S., Samarati, P., et al.: A logical language for expressing authorizations. In: Proc. of 1997 IEEE Symposium on Security and Privacy, pp. 31–42 (1997)
25. Naumovich, G., Centonze, P.: Static Analysis of Role-Based Access Control in J2EE Applications. SIGSOFT Software Engineering Notes 29(5), 1–10 (2004)
26. Centonze, P., Naumovich, G., Fink, S.J., et al.: Role-Based Access Control Consistency Validation. In: Proc. of the ISSTA 2006, pp. 121–132. ACM Press, New York (2006)
27. Martin, E., Xie, T., et al.: Assessing Quality of Policy Properties in Verification of Access Control Policies. Technical Report. North Carolina State University Raleigh, NC, USA (2007)
28. Sohr, K., Ahn, G.J., et al.: Specification and Validation of Authorisation Constraints Using UML and OCL. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 64–79. Springer, Heidelberg (2005)
29. Pistoia, M., Fink, S.J., et al.: When Role Models Have Flaws: Static Validation of Enterprise Security Policies. In: Proc. of ICSE 2007, pp. 478–488. IEEE Computer Society, Los Alamitos (2007)
30. Massacci, F., Zannone, N.: Detecting Conflicts between Functional and Security Requirements with Secure Tropos: John Rusnak and the Allied Irish Bank, Technical Report DIT-06-002, University of Trento (2006)

Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms*

Pierre Parrend and Stéphane Frénot

INRIA Amazones / CITI, INSA-Lyon, F-69621, France
Tel.: +334 72 43 71 29, Fax. +334 72 43 62 27
{pierre.parrend,stephane.frenot}@insa-lyon.fr

Abstract. Java-based systems have evolved from stand-alone applications to multi-component to Service Oriented Programming (SOP) platforms. Each step of this evolution makes a set of Java vulnerabilities directly exploitable by malicious code: access to classes in multi-component platforms, and access to object in SOP, is granted to them with often no control.

This paper defines two taxonomies that characterize vulnerabilities in Java components: the vulnerability categories, and the goals of the attacks that are based on these vulnerabilities. The 'vulnerability category' taxonomy is based on three application types: stand-alone, class sharing, and SOP. Entries express the absence of proper security features at places they are required to build secure component-based systems. The 'goal' taxonomy is based on the distinction between undue access, which encompasses the traditional integrity and confidentiality security properties, and denial-of-service. It provides a matching between the vulnerability categories and their consequences. The exploitability of each vulnerability is validated through the development of a pair of malicious and vulnerable components. Experiments are conducted in the context of the OSGi Platform. Based on the vulnerability taxonomies, recommendations for writing hardened component code are issued.

1 Introduction

Java execution environments evolve from stand alone applications to component-based systems to Service Oriented Programming (SOP) Platforms [1]. Component-based systems introduce multi-application execution. Service Oriented Programming (SOP) Platforms add a strong runtime dynamicity of component linkage, thus supporting more customizable applications. While new features are added, each of these evolutions turns potential vulnerabilities into directly exploitable flaws. Access to component class represents a first important threat. It makes class vulnerabilities directly exploitable by other components. SOP broaden this threat by enabling direct access to objects provided by these

* This work is partially funded by the ANR-07-SESU-007 LISE Project.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 80–96, 2008.

components, with often no restriction. This makes object vulnerabilities exploitable. We present a classification of vulnerabilities of components, so as to help developers identify and mitigate this threat and build secure component-based systems.

Experiments are conducted on the Java/OSGi SOP Platform [13]. They aim at identifying vulnerabilities that can actually be exploited by malicious components that are installed on a system, as well as additional preconditions. Is considered as an exploitable vulnerability any feature which use leads to a behavior that break explicit or implicit security policies for the system [8].

In most cases, the condition of exploitation is that vulnerabilities must be present in the code that is made available to other components. This is what we call *Public Code*. This concept is introduced in the Parnas and Wang component model [14], cited by [4].

The following of the paper is organized as follows. Section 2 presents related works. Section 3 describes the vulnerability categories, and provides the related taxonomies. The rationale and experiment of this study is provided in Section 4. Section 5 concludes this work.

2 Related Works

As a part of the Java ecosystem, the Java language itself has been designed with a strong emphasis on security. However, as no system is entirely secure, pitfalls and behaviors exist that turn out to be actual vulnerabilities, in particular in the context of multi-component systems and Service Oriented Programming.

2.1 Attack Vectors against the Java/OSGi SOP Platform

Hackers can attack Java/OSGi applications by exploiting two main attack vectors: platform vulnerabilities, and component vulnerabilities. Platform vulnerabilities can be exploited to indirectly attack other components. The only requirement for exploiting them in a default, non secure Java/ OSGi platform implementation, is to install a bundle that calls dangerous or faulty platform code. This often implies that it is published in a known bundle repository, and sometimes that it is signed. A security analysis of the Java/OSGi Platform is given in [15]. 32 vulnerabilities are identified. They lead to Denial-of-Service (through platform crash or performance breakdown) and to undue access to code. Most vulnerabilities (18 out of 32) are bound with the JVM, such as the lack of CPU and memory isolation between components, the Runtime API, the presence of dangerous functionalities such as native code execution, thread creation, reflection. Others (14 out of 32) are bound with the OSGi Platform itself, such as bundle fragments, bundle management, and lack of control on Service-Oriented-Programming. All of these vulnerabilities lead to attacks against the Platform that can be exploited to harm other components. Other vulnerabilities are specific to given implementations of the JVM [2], or to specific embedded platforms such as the CLDC [3].

Component vulnerabilities can be exploited to directly attack other components. They are due to java language properties [7] that can be misused to achieve a malicious goal.

2.2 Known Vulnerabilities in Java/OSGi Components

After platform vulnerabilities, the second kind of vulnerabilities that can plague Java-based component systems is the presence of flaws in the components themselves. They can be exploited as soon a malicious component can be installed in a Platform where components share code with each other.

Several works provide hints related to some attacks against Java-based systems, without taking a systematic approach. Java features that can lead to vulnerabilities are presented by Long [12]: type safety limitations, public fields, inner classes, serialization, reflection, JVM Tool Interface, debugging and management tools can be exploited to abuse Java-based applications. More weaknesses are mentioned by the Last Stage of Delirium Research Group [18], such as unsafe type conversion, class loader attacks, bad implementation of system classes. Another specific attack consists in executing arbitrary code through forced type mismatch [5]. It is based on memory errors that can mainly be forced through physical access to the machine. These vulnerabilities form the first set of occurences on which our experiments are based.

The first systematic set of candidate vulnerabilities that flaw Java Extensible Component Platforms is provided by the Findbugs tools Vulnerability List [1] [6]. The *Malicious Code Vulnerability* category identifies 12 code patterns that can lead to exposition and modification of object internal data to another potentially untrusted code element, such as returning references to mutable objects or array or storing data in class variable that are not properly encapsulated.

The second systematic set of candidates vulnerabilities that flaw Java Extensible Component Platforms is provided by the 'Sun Java Security Coding Guidelines' [17]. Each guideline matches a code flaw that can be exploited by untrusted code to perform malicious actions. For instance, abuse of inheritance, faulty validation and copy of method parameters or returned objects, security checks by-passing and serialization/de-serialization of sensitive data are referenced. Sun Java Security Coding Guidelines are completed by Charlie Lai's Java Insecurity Subtleties [9]. These two lists of vulnerabilities form the second set of occurences on which our experiments are based. More are detailed in the Appendix A.1.

These references provide useful support both to train developers and for supporting vulnerability identification through static analysis. However, several criticisms can be issued. First, none of these works provides a classification that is structured or complete. Secondly, they do not provide information relative to the exploitability of these vulnerabilities: are they present but harmless, or is any installed component able to exploit them all with little to no additional effort?

[1] http://findbugs.sourceforge.net/bugDescriptions.html

3 Vulnerabilities in SOP Platforms

Vulnerabilities in Java/OSGi components pertain to three categories: Stand-Alone components, Class Sharing, and Object Sharing. The last category is made exploitable by the Service Oriented Programming (SOP) paradigm. Two taxonomies characterize at best their properties: Categories of the vulnerabilities, and goals of attacks that exploit them. These taxonomies are obtained by classifying the 39 distinct vulnerabilities that we identified through bibliographical review and through our own experience. Two examples that highlight abuse risks are given in the Appendix A.1: *Malicious Inversion of Control* and *Synchronized Code*.

3.1 Vulnerability Classes

Classes of vulnerabilities are defined according to the preconditions that must be enforced to exploit them. These preconditions are: No access to the code (*Stand-Alone component*), access to classes (*Shared Classes*), access to objects (*Shared Objects* or *SOP*). These component vulnerabilities are referenced in two vulnerability catalogs: the *Malicious Bundles* catalog [15], which identifies vulnerabilities that can be exploited through malicious components and are implied by platform features, and the *Vulnerable Bundles* catalog [16], which identifies vulnerabilities that are implied by component features, mostly based on Java language properties [7]. Following features of the Java/OSGi Platform lead to component vulnerabilities: the reflection API, SOP services, and fragments. Other entries of the *Malicious Bundles* catalog are not considered here, since they concern the implementation of the platform and the isolation mechanisms it enforces, and not the way components are coded.

So as to provide an overview of the relative importance of each vulnerability category, their cardinality is extracted.

The total number N of vulnerabilities that we identify in Java/OSGi components is: 6 vulnerabilities from the *Malicious Bundles* catalog, and all 33 vulnerabilities from the *Vulnerable Bundles* catalog.

$$N = 6 + 33 = 39$$

The number N_{SA} of vulnerability in stand alone components is 1, which matches the use of serialization. When not properly protected, it provides access to any entity that is able to read the serialized data, for instance through the network or the file system. This vulnerability may not be restricted to component platforms.

$$N_{SA} = 1$$

Vulnerabilities that pertain to the *Shared Classes* vulnerability category can be exploited provided that two conditions are met. First, victim code must be loaded by the same ClassLoader as the attack CODE, OR be shared among ClassLoaders. In the Java/OSGi case, this concerns exported packages as well as bundle fragments and their hosts. Secondly, the code must be launched by the application. In OSGi, this is for instance done through the bundle activator, or when methods are called. These vulnerabilities occur mainly when static

fields exists in the code, and when reflection, inheritance and fragments can be exploited.

$$N_{CL} = 18$$

Some vulnerabilities require the execution of a given method, which can be achieved either through static access or through SOP, depending on the implementation. This is the case *e.g.* for synchronization problems. They can therefore not be classified in one or the other category, though for simplicity one can consider them to be SOP vulnerabilities, because they are much more likely to be exploitable in this case.

$$N_S = 2$$

The vulnerabilities that pertain to the *Object Sharing* category can be executed provided that a malicious component can be installed, and that access to objects is granted. This is typically the case in SOP Platforms. For instance, in OSGi, it is possible to access all objects that are registered as services. The number N_{SOP} of vulnerabilities in the *Object Sharing* category is:

$$N_{SOP} = 18$$

Figure 1 provides an overview of the vulnerability categories in a SOP Platforms.

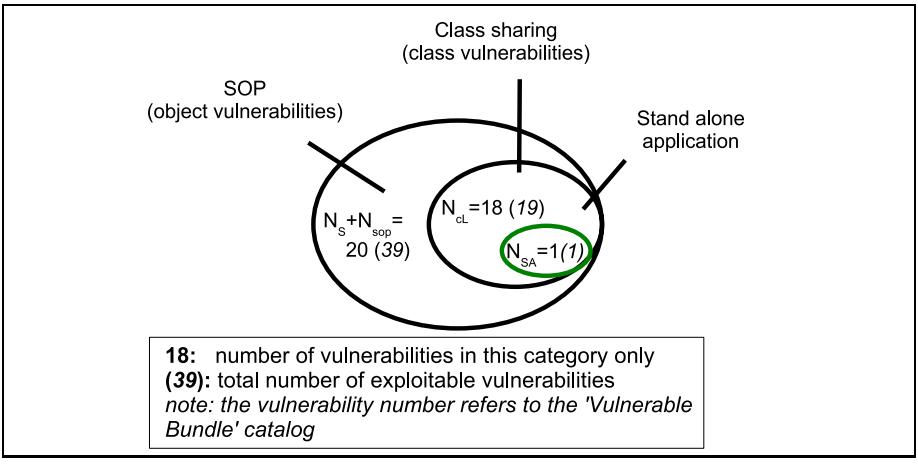


Fig. 1. Vulnerability types in a SOP Platform

3.2 Vulnerability Implementations

Vulnerability lists are usually given without regard to their actual likeliness. The following taxonomy provides for each system configuration the set of vulnerabilities that can be exploited without further effort. Each vulnerability category extends the others: Stand Alone Application vulnerabilities can also be exploited in case of Class Sharing system, and Class Sharing vulnerabilities can be exploited in the context of SOP.

Table 1 present the taxonomy for vulnerability categories according to the vulnerability category they pertain to.

Table 1. Taxonomy: Implementations of the Component Vulnerabilities in Java/OSGi SOP Platform

Attack Vector	**Implementation**			**Occurences**
Component Interactions	Stand Alone App.	Serialization		1
	Class Sharing	Exposed Internal Representation	Mutable element in static variable	2
			Reflection	3
			Fragments	2
			No suitable control	2
		Avoidable Calls to the Security Manager	At instanciation	4
			In method call	5
	Class Sharing or SOP	Synchronization		2
	SOP	Exposed Internal Representation	Returns reference to mutable element	2
			No suitable control	4
		Flaws in Parameter Validation	Unchecked parameter	3
			Checked parameter without copy	1
			Checked and copied parameter	4
			Non final parameter	2
		Invalid Workflow		1

Stand Alone Applications. Stand alone applications do not enable to run third party code. The only code-level vulnerability that can be exploited in this case is the access to internal data that is made available through serialization. This flaw can be prevented by avoiding serialization, or by properly protecting, for instance through cryptography, the serialized data.

Other vulnerabilities may of course also exist, but they are related with the application behavior itself, not with the code properties, and are therefore not of interest here.

Class Sharing. Platforms that support Class Sharing are typically component-based systems. Each component can make classes available, and have dependencies to others. In the OSGi Platform, for instance, this feature is supported by the Module Layer. Class Sharing makes two main category of vulnerabilities open for exploits: *Exposed Internal Representation* and *Avoidable Calls to the Security Manager*. In some specific cases, the *Synchronization* vulnerability can also be exploited.

Exposed Internal Representation enables malicious components to access data inside victim components. In the Class Sharing case, it enables to

execute code that should remain hidden, and to access static class members[2]. Four sub-categories exist: *Mutable element in static variable*, *Reflection*, *Fragments* and *No suitable control*.

- *Mutable element in static variable* vulnerabilities consist in giving access to third party components to fields that are both static and final, but which content can nonetheless be modified. This occurs when the fields either contain arrays, or mutable classes. Mutable classes are any classes that store data that the client object can modify. For instance, implementations of the `Set` and `Collection` interfaces are mutable. The only way to prevent this vulnerability to be exploited is to ban such constructs from public variables of public code classes.
- The *Reflection* sub-category consists in exploiting the Reflection API to access and exploit the content of the victim component. It encompasses code observation, component data modification when this data is static, and launching hidden method . The protection against these vulnerabilities are of two types. First, clean encapsulation can prevent unwanted access, since reflection does not allow to access fields and execute methods when visibility modifiers (public, protected, default, and private) forbid it. Secondly, Java Permissions can be set to prevent untrusted components from using the Reflection API.
- *Fragments* vulnerabilities exploit the OSGi-specific fragments. Fragments are used to provide configuration data and code to OSGi bundles, *e.g.* for supporting context specific behaviors such as internationalization. Fragment code is executed in the same ClassLoader as its Host bundle. This enable them to have full access to the code, and to share this access with other components by exporting it. Three implementations exist for this vulnerability category. First, a fragment can access the classes inside its host bundle. It can call classes that do not pertain to public code. Next, the *split package* feature enable to gain access to package protected classes, fields and methods, if the fragment contains a package with the same name as the targeted package in the host. Lastly, private inner classes, which are made package protected at compilation, are thus available from the fragment. Protection against fragments consists in setting `BundlePermission:HOST` and `BundlePermission:FRAGMENT` to trusted components only.
- *No suitable control* vulnerabilities enable to influence the behavior of the application through class access. In particular, shutdown hooks[3] can be exploited to keep a handle on an object after all references have been destroyed in the application. This enables in particular the execution of code after components have been uninstalled. The protection against the shutdown hook attack can be obtained by preventing untrusted components to set such hooks, *e.g.* through Java Permissions.

[2] Class members are fields and methods.

[3] Shutdown hooks are methods that are executed during the shutdown process of the virtual machine. They can be set at any moment.

Avoidable Calls to the Security Manager enables malicious components to by-pass security checks that occur in the code. These vulnerabilities are either exploited by overriding the code that contains the check, or by taking advantage of methods that are executed in spite of the presence of a Security Manager (or any similar check).

Two types of vulnerabilities are identified: avoidable checks 'at instantiation', which allows to create protected objects, and avoidable checks 'in method call', which allows to perform protected actions.

- The avoidable checks *at instantiation* category consists either in not using a constructor that contains security checks to create objects, or in overriding it in a sub-class. Object creation without constructor can be achieved either through the `clone()` method, or through de-serialization, when these two mechanisms are not protected. The protection consists in performing the same security checks in all constructors, in the `clone()` method if the class is `cloneable`, and in the `readObject()` method if the class is `serializable`. Avoiding security checks through overriding simply consist in re-writing the methods that contain the checks. This is possible either if a constructor exists that does not contain checks, or if the checks are performed in other methods. Consequently, these methods should always be final to prevent exploitation.
- The avoidable checks *in method call* category consists in performing actions that should be prevented by the security policy. The simplest way to achieve this is to override a method that contains a security check by a self-defined one. Executing methods of objects which creation has aborted due to security reasons is also possible: the `finalize()` method is always executed, even through the constructor could not be properly executed. Calls on the object, which is in a such case often only partially initialized, can typically reveal internal data. The protection here is to perform security checks at the very beginning of the creator method (constructor or other), to prevent data to be set before the security check. The last vulnerability that avoids security checks consists in executing sensible operations on behalf on untrusted components. This is done through `doPrivileged()` calls. A specific case can occur with security checks that depends only on the local ClassLoader, such as `java.lang.Class.forName` and `java.lang.Class.newInstance`. The protection against these two vulnerabilities is to never execute sensitive operations on behalf of others.

Synchronization vulnerabilities threaten Java/OSGi Platforms with freezing: if a `synchronized` method call does not return, all subsequent calls keep waiting for the lock to be released. Exploiting these vulnerabilities requires either that the `synchronized` call freezes by itself, or that the malicious component is able to interfere with its execution, for instance by providing a malicious service on which the victim method relies. So as to make attack through Shared Classes possible, these methods must be launched through a static method call, either directly (the `synchronized` method is also static) or indirectly (the `synchronized`

method is called by a static method). Attack is triggered when this malicious service freezes and thus blocks the `synchronized` call. Two implementations exist: either a full method is synchronized, or a code block inside a method. This vulnerability occurs without regard to the location of the `synchronized` keyword inside the component: they are not restricted to public code. The ways to prevent them is to ban `synchronized` code from components, or to ensure that only trusted and non-freezing components are called by `synchronized` statements.

Service Oriented Programming (SOP). Service Oriented Programming Platforms support the dynamic registration and discovery of local services, *i.e.* objects that are characterized by the interface they implement. In the OSGi Platform, for instance, this feature is supported by the Service Layer. Service Oriented Programming provides full access to the service objects, which means that both read and write access is granted. The vulnerabilities that plague SOP are the following: **Exposed Internal Representation**, **Flaws in Parameter Validation**, and **Invalid Workflow**. Moreover, the exploitation of **Synchronization** vulnerabilities is much easier, since synchronized methods can be targeted without requiring a static access.

Exposed Internal Representation enables, as in the Class Sharing case, malicious components to access data inside victim components. In the SOP case, these vulnerabilities enable malicious code to access and thus modify data that should be kept internal to the object. Two vulnerability categories exist: *Returns reference to mutable element* and *No suitable control*.

- The *Returns reference to mutable element* category occurs when a method returns these very mutable elements. If a proper copy is not performed before giving a reference of a mutable object to a third party component, this latter is able to modify it. Malicious or accidental conflicts can then occur between the modifications that take place inside the vulnerable component, and the modifications that are performed by the caller. The protection consists in copying the mutable element before returning it. This can only be achieved if the considered mutable element does not itself contain mutable elements. Otherwise, the copy process would be overly complex and error prone.
- The *No suitable control* category in the *Object Sharing* vulnerability class enables information leak from one component to another. It encompasses the absence of wrapper (no encapsulation), an excessive visibility for the members[4] or classifiers[5]. The protection against ill-coded public classes vulnerabilities consists in a proper encapsulation of all variables. Another vulnerability is the leak of configuration, system, or application sensitive data through exceptions. Exception handing should therefore either be performed internal to the component, or only provide generic data that contain at most references to user input to keep the message informative without revealing the internal component state.

[4] Class members are fields and methods.
[5] Classifiers are classes and interfaces.

Flaws in Parameter Validation enables malicious or ill-coded components to call methods from other ones while passing objects as parameter that are either not supported or lead to unexpected code behavior.

Four sub-categories exist: 'Unchecked parameter', 'Checked parameter without copy', 'Checked and copied parameter', and 'Non final parameter'.

- The *Unchecked parameter* category occurs when the method parameters are not checked before use. It contains three vulnerabilities: accidentally unsupported values that cause the program to behave in an erratic manner, malicious Java code, and malicious native code. In this latter case, the caller can forge and provide arbitrary malicious code. In particular, parameters that are defined as interfaces or as non final classes are vulnerable. The protection against such abuses consists in checking both the value and the actual type of the parameters. Public class methods should only accept parameters which types are final classes, so as to prevent malicious inheritance. Lastly, no native code should be executed on behalf of other components.
- The *Checked parameter without copy* vulnerability consists in performing the validation of the parameter, but without previously copying it to a local variable. If the object is modified in the caller component after the validation occurs, it can take arbitrary values, including those which are rejected by the validation process. The absence of parameter copy makes parameter validation useless because of TOCTOU (Time of Check to Time of Use) attacks. The suitable protection consists of course in copying the parameter object before its validation.
- The *Checked and copied parameter* category highlights the restriction of the parameter copy process: unless an object is serializable and thus explicitly states which fields are `transient` and are thus not required during copy, copying it is not necessarily straightforward. Two types of vulnerabilities exist. The first one is the presence of fake clone methods or copy constructor, which are provided by the malicious parameter itself: a copy statement is present in the code, but does not perform as expected. The protection against this problem is to use trustworthy copy methods only, such as those provided by the Java API, or manual copy. The second type of vulnerabilities is related to the manual copy process, which can be uncomplete. This occurs either when some states are omitted during the copy process, or when the given object contains references to other objects. This later problem implies that parameter objects should have a limited depth of mutable objects so as to prevent copy faults and omissions.
- The *Non-final parameter* category consists in exploiting the extensibility of classes or the possibility of providing self-defined implementation of interfaces to execute arbitrary code. This can also lead to more complex scenario: a malicious parameter can be used to trigger execution of code in the caller bundle, possibly passing back data from the victim bundle. This actually builds a case of malicious inversion of control (see Appendix A.1). As we already mentioned, the protection against this vulnerability is to allow only basic and final types as method parameters in public classes. A copy

mechanism designed to avoid cited flaws can also prevent this vulnerability category from being exploited, as the object passed as parameter is no longer used during the method execution.

Invalid Workflow (SOP) vulnerabilities are bound with invalid configuration of the service dependencies. In Java/OSGi platforms, services are discovered and retrieved through the `BundleContext`, which plays the role of local service repository. Service lookup is performed according to a given Java interface, with possibly additional provider-set properties. Consequently, very little control is enforced, in particular when components from several mutually untrusted providers coexist in a SOP platform. This lack of control have two main consequences. First, there is no guarantee that found services actually provide a valid implementation of the advertised interface. They could either provide arbitrary code, or gather data that is passed to them as parameters. Secondly, there is no guarantee that the service call does not abort. Such abortion can be generated either directly, for instance by systematically throwing exceptions, or indirectly, for instance by creating loops between services that lead to `StackOverflowError`s. To date, most SOP frameworks assume that provided services are benevolent. The identified risks show that a full SOP security framework should be designed if this should not be the case. This is a requirement for future work.

3.3 Goals of the Attacks That Exploit These Vulnerabilities

The goals of the attacks that exploit vulnerabilities in Java/OSGi component interactions are described below. The main goals are *Undue Access* and *Denial of Service*. Undue access is either *Access to internal Data* or *By-pass Security Checks*. Denial of Service (DoS) is restricted to method unavailability, because it is achieved through method calls on the public code. More serious DoS attacks can be performed in Java/OSGi platforms by attacking the platform directly [15].

The taxonomy of the goals of the attacks that can be performed by taking advantage of vulnerabilities in Java/OSGi component interactions is shown in Table 2, along with related vulnerability categories.

Undue Access - Access to internal Data. Hackers can gain *Access to internal Data* through the *Exposed Internal Representation* and the *Fragments* vulnerabilities.

Table 2. Taxonomy: Goals of the Attacks that exploit Vulnerabilities in Java/OSGi Component Interactions

Attack Goal	**Sub-goal**	**Interaction Category**
Undue Access	Access to internal Data	Class Sharing and SOP - Exposed Internal Representation Class Sharing - Fragments
	By-pass Security Check	Class Sharing - Avoidable Calls to the Security Manager SOP - Flaws in Parameter Validation
DoS	Method unavailability	Class Sharing and SOP - Synchronization SOP - Invalid Workflow

The first vulnerability provides access to internal data of the component that provides ill-coded Shared Classes of Shared Objects. The second one provides access to all the code of the target component, but without access to the actual objects. Attacks are performed by malicious client components.

Undue Access - By-pass Security Check. Hackers can *By-pass Security Check* through the *Avoidable Calls to the security manager* and the *Flaws in parameter validation* vulnerabilities. The first vulnerability enables to execute code that is not properly protected by security checks. It is specific to the Shared Classes vulnerability category. The second one enables to pass unvalid or malicious code as method parameters. It is specific to the Shared Object vulnerability category. Attacks that exploit both weaknesses are performed by malicious client components.

Denial of Service - Method Unavailability. Hackers can force *Method Unavailability* through the *Synchronization* and the *Invalid SOP Workflow* vulnerabilities. Both vulnerabilities enable to block the normal execution of programs, by freezing them of by forcing them to abort. In most cases they are bound with the Shared Object vulnerability category, but synchronization can also be exploited through Shared Classes. Attacks that exploit both weaknesses are performed by malicious servant components, *i.e.* malicious components which are dependencies of the victim code.

4 Experiments

A vulnerability is any feature that forces a program to behave so that it breaks the implicit or explicit security policy of the considered system [8]. They are generated by errors in the program development or by assumptions that are not valid in the execution context. In the case of vulnerabilities in Java/OSGi component interactions, the second case holds: the Java language has not been designed to support the execution of mutually untrusted components in the same virtual machine.

4.1 Rationale

The rationale for identifying, validating and classifying these vulnerabilities is the following. First, we gather knowledge about Java behaviors that are considered as the expression of vulnerabilities. Sources are the computer science literature as well as our own experience. Secondly, the Java Language Specification is analyzed to identify further vulnerabilities, and to check that no language construct has been neglected [7]. Next, the suspected vulnerabilities are validated through proof of concept implementation of the attack scenarios. Lastly, taxonomies are created to classify both vulnerability type - their implementation - and the goal of the attacks based on the experiment results.

This rationale is strongly inspired by similar studies that focus on Operating System vulnerabilities, such as those by Landwehr [10] and Lindqvist [11] for Unix.

4.2 Implementation of Malicious and Vulnerable SOP Components

Each identified vulnerability must be validated by implementing it so as to confirm that it actually breaks security requirements.

The experiment environment is the Java/OSGi Platform. Tests are conducted on the Sun JVM 1.6, with the Apache Felix[6] open source implementation of the OSGi Platform. Felix 1.0.0 is compliant with the OSGi Release 4 Specifications.

An implementation of a vulnerability validates this vulnerability if the malicious component actually performs an operation on the vulnerable one that breaks the implicit or explicit security policy, *i.e.* that either is able to perform more operations that calling provided methods, or enforces a denial of service.

For each of the 39 vulnerability occurrences that we identify, a malicious/ vulnerable component pair is implemented. Providing an implementation for each attack has a twofold goal. It enables to validate the feasibility of the attack, and provide a sound basis for our documentation effort. And it makes sample code available for subsequent effort toward automated vulnerability identification.

5 Conclusions and Perspectives

Based on the presented experiments and classifications of vulnerabilities in Java/ OSGi component interactions, following recommendations can be emitted to component developers. Security constraints should be enforced at two level: the component level, *i.e.* the application architecture, and the Public Code level, *i.e.* the code that components make available to others.

Components should:

- only have dependencies on components they trust,
- never used synchronized statements that rely on third party code,
- provide a hardened public code implementation following given recommendations.

Shared Classes should:

- provide only final static non-mutable fields,
- set security manager calls during creation in all required places, at the beginning of the method: all constructors, `clone()` method if the class is cloneable, `readObject(ObjectInputStream)` if serializable,
- have security checks in final methods only,

Shared Objects (*e.g.* SOP Services) should:

- only have basic types and serializable final types as parameter,
- perform copy and validation of parameters before using them,
- perform data copy before returning a given object in a method. This object should also be either a basic type or serializable,

[6] http://felix.apache.org

– not use Exception that carry any configuration information, and not serialize data unless a specific security mechanism is available,
– never execute sensitive operations on behalf of other components.

The contribution of this paper is twofold. First, taxonomies that describes the categories of exploitable vulnerabilities and their goals for Java systems are defined. The three main system types are stand alone applications, multi-component systems, and Service Oriented Programming (SOP) Platforms, which each make a specific set of vulnerabilities directly exploitable. Secondly, recommendations are issued to help software developers build more secure code.These recommendations can be used for training, or to enrich the flaw sets that are identified by static analysis tools. Our approach is validated through a systematic implementation of each vulnerability through a proof-of-concept malicious / vulnerable pair of OSGi bundles. Experiments show that given vulnerabilities are actually directly exposed to malicious components in standard platforms. The only condition is that the malicious component can be installed and executed to perform its abuses.

The perspective of this work is first to disseminate the knowledge gathered through the development of plug-ins for static analysis tools such as FindBugs or PMD.

Further requirements are also identified. A security framework should be defined and developed to enforce security at the SOP level. Current tools, such as SCR for the OSGi Platform, do not take security into account. This mechanism should be made mandatory and support for instance dynamic proxies that would prevent the exploitation of identified vulnerabilities by isolating service implementation and service client. Such a feature could prove to provide a big improvement in the quest after the dynamic discovery of unknown components from the environment, while ensuring that the system security is not at risk.

Aknowledgement

This work has been made possible by valuable discussions with Emmanuel Coquery (LIRIS, Lyon 1 University) and Nicolas Geoffray (LIP6, Paris VI University).

References

1. Bieber, G., Carpenter, J.: Introduction to service-oriented programming (rev 2.1). OpenWings Whitepaper (April 2001)
2. Cotroneo, D., Orlando, S., Russo, S.: Failures classification and analysis of the java virtual machine. In: 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006) (2006)
3. Debbabi, M., Saleh, M., Talhi, C., Zhioua, S.: Security evaluation of j2me cldc embedded java platform. Journal of Object Technology 5(2), 125–154 (2005)
4. Dolbec, J., Shepard, T.: A component based software reliability model. In: CASCON 1995: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, p. 19. IBM Press (1995)

5. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: Symposium on Security and Privacy (2003)
6. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: ACM SIGPLAN Notices, vol. 39, p. 92–106 (2004); COLUMN: OOPSLA onward
7. Steele, G., Bracha, G., Gosling, J., Joy, B.: Java Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
8. Krsul, I.V.: Software Vulnerability Analysis. PhD thesis, Purdue University (May 1998)
9. Lai, C.: Java insecurity: Accounting for subtleties that can compromise code. IEEE Software 25(1), 13–19 (2008)
10. Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S.: A taxonomy of computer program security flaws, with examples. In: ACM Computing Surveys, September 1994, vol. 26, pp. 211–254 (1994)
11. Lindqvist, U., Jonsson, E.: How to systematically classify computer security intrusions. In: IEEE Symposium on Security and Privacy, pp. 154–163 (May 1997)
12. Long, F.: Software vulnerabilities in java. Technical Report CMU/SEI-2005-TN-044, Carnegie Mellon University (October 2005)
13. OSGI Alliance. Osgi service platform, core specification release 4.1. Draft, 05 2007
14. Parnas, D.L., Wang, Y.: The trace assertion method of module interface specification. Technical Report 89-261, Dept. of Computing and Information Science, Queen's Univ. at Kingston, Ontario, Canada (October 1989)
15. Parrend, P., Frenot, S.: Java components vulnerabilities - an experimental classification targeted at the osgi platform. Research Report RR-6231, INRIA, 06 (2007)
16. Parrend, P., Frenot, S.: More vulnerabilities in the java/osgi platform: A focus on bundle interactions. Technical report, INRIA (to be released, 2008)
17. Sun Microsystems Inc. Secure coding guidelines for the java programming language, version 2.0. Sun Whitepaper (2007), `http://java.sun.com/security/seccodeguide.html`
18. The Last Stage of Delirium. Research Group. Java and java virtual machine. security vulnerabilities and their exploitation techniques. In: Black Hat Briefings (2002)

A Appendix

The Appendix presents additional informations related to vulnerabilities in Java/OSGi component interactions. Subsection A.1 gives a detailed documentation for two vulnerabilities that exist in component-based applications: *Malicious Inversion of Control through overridden Parameters*, and *Synchronized Code*.

A.1 New Attacks Exploiting Interactions between Java Components

We now present two behaviors that enable malicious components to exploit weak ones in order to achieve security breaks inside component-based applications: *Malicious Inversion of Control through overridden Parameters*, and *Synchronized Code*. The first vulnerability enables an attack that performs undue access to code. The second one enables an attack that performs denial of service. To the best of our knowledge, these behaviors of Java components have not yet been identified and documented as vulnerabilities.

The *Malicious Inversion of Control through overridden Parameters* vulnerability occurs when public code expose methods with non-final parameters. This is the case for all parameters that are defined as interfaces, and most classes with the exception of basic type wrappers (`Integer`, *etc*) and `String`. Abuse occurs when called methods are overwritten, and trigger actions that are not supposed to take place such as spying the behavior of the servant bundle or getting undue access to internal data. An example of an attack that exploits this vulnerability is given in Figure 2 as an UML Component Diagram.

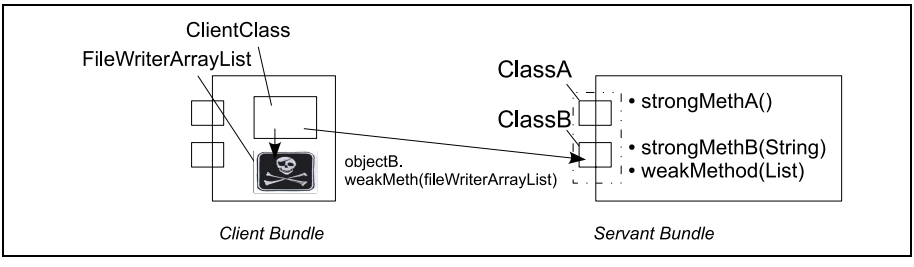


Fig. 2. An Example Scenario of malicious Inversion of Control: Component Diagram

The weak method, named `weakMethod(List)`, is provided by the class `ClassB` of the servant bundle. In our example, it simply manipulates the `List` parameter. The attack is performed as follows. First, the client bundle defines a malicious `FileWriter- ArrayList`, whose `iterator()` method is overwritten and triggers action that it should not. In our case, this is a single text print for demonstration. The client bundle creates a `FileWriterArrayList` object, and passes it as parameter to the `ClassB.weakMethod(List)` method. When code in `ClassB.weakMethod(List)` is executed, malicious code is executed seamlessly. Again, the example does not go further than the demonstration, but shows how a naive servant can execute unrequired code from its caller.

This vulnerability has one main consequence: public code that is intended to be executed by not fully trusted code should never provide methods with non final parameters.

The *Synchronized Code* vulnerability occurs when code in a public class is tagged as `synchronized`, which means that one single client bundle can access it at a time. Synchronization is used in particular to protect transactions or access to system resources. Abuse occurs when the synchronized method is forced to hang, which causes all subsequent calls to the method to freeze. An example of an attack that exploits this vulnerability is given in Figure 3 as an UML Sequence Diagram.

The synchronized method, `setData()`, is provided by the `Data` class. This service relies on another one, `DataStorage`. A default valid scenario is executed by Alice, which is a benevolent component that stores data every 20 seconds. The attack is performed as follows. First, the `DataStorage` service must be replaced

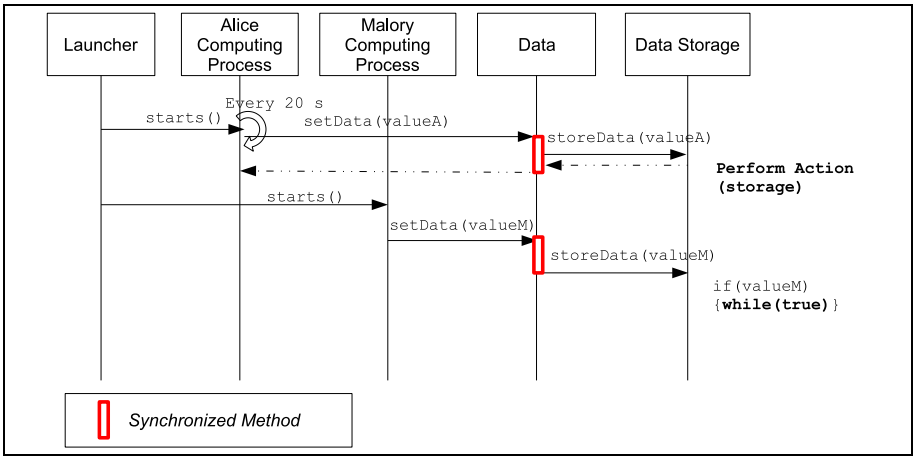


Fig. 3. An Example Scenario for an Attack against a Synchronized Method: Sequence Diagram

by a malicious one, which hangs under certain circumstances (here, a specific `valueM` value of the transmitted data is the signal for hanging). This substitution can be replaced by a Denial-Of-Service Attack against a valid implementation of the `DataStorage` service. The Mallory component is the accomplice of the malicious `DataStorage` service, and therefore knows how to trigger its freezing (transmit data with 'valueM' value). It performs the malicious call to the `Data` service, which in turn calls the `DataStorage` service, which hangs. As a consequence, Alice as well as any other client of the `Data` service will hang.

The *Synchronized Code* vulnerability exists under two flavours: Synchronized method and Synchronized code block. This vulnerability has two consequences. First, access to synchronized methods MUST be granted to trusted components only. Secondly, services on which synchronized methods rely MUST be guaranteed to be trustworthy.

Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems

Chiyoung Seo[1], Sam Malek[2], and Nenad Medvidovic[1]

[1] Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781, U.S.A.
{cseo,neno}@usc.edu
[2] Department of Computer Science, George Mason University, Fairfax, VA 22030-4444, U.S.A.
smalek@gmu.edu

Abstract. Efficiency with respect to energy consumption has increasingly been recognized as an important quality attribute for distributed software systems in embedded and pervasive environments. In this paper we present a framework for estimating the energy consumption of distributed software systems implemented in Java. Our primary objective in devising the framework is to enable an engineer to make informed decisions when adapting a system's architecture, such that the energy consumption on hardware devices with a finite battery life is reduced, and the lifetime of the system's key software services increases. Our framework explicitly takes a component-based perspective, which renders it well suited for a large class of today's distributed, embedded, and pervasive applications. The framework allows the engineer to estimate the distributed system's energy consumption at system construction-time and refine it at runtime. In a large number of distributed application scenarios, the framework showed very good precision on the whole, giving results that were within 5% (and often less) of the actual energy consumption incurred by executing the software. Our work to date has also highlighted the framework's practical applications and a number of possible enhancements.

Keywords: Distributed systems, energy consumption, Java, component-based software.

1 Introduction

Modern software systems are predominantly distributed, embedded, and pervasive. They increasingly execute on heterogeneous platforms, many of which are characterized by limited resources. One of the key resources, especially in long-lived systems, is battery power. Unlike the traditional desktop platforms, which have uninterrupted, reliable power sources, a newly emerging class of computing platforms have finite battery lives. For example, a space exploration system comprises satellites, probes, rovers, gateways, sensors, and so on. Many of these are "single use" devices that are not rechargeable. In such a setting, minimizing the system's power consumption, and thus increasing its lifetime, becomes an important quality-of-service concern.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 97–113, 2008.

The simple observation guiding our research is that if we could estimate the energy cost of a given software system in terms of its constituent components ahead of its actual deployment, or at least early on during its runtime, we would be able to take appropriate, possibly automated, actions to prolong the system's life span: unloading unnecessary or expendable components, redeploying highly energy-intensive components to more capacious hosts, collocating frequently communicating components, and so on.

To this end, we have developed a framework that estimates the energy consumption of a distributed Java-based software system at the level of its components. We chose Java because of its intended use in network-based applications, its popularity, and very importantly, its reliance on a virtual machine, which justifies some simplifying assumptions possibly not afforded by other mainstream languages. We have evaluated our framework for precision on a number of distributed Java applications, by comparing its estimates against actual electrical current measurements. In all of our experiments the framework has been able to estimate the power consumed by a distributed Java system to within 5% of the actual consumption.

One novel contribution of our estimation framework is its component-based perspective. To facilitate component-level energy cost estimates, we suggest a computational energy cost model for a software component. We integrate this model with the component's communication cost model, which is based on the experimental results from previous studies. This integrated model results in highly accurate estimates of a component's overall energy cost. Furthermore, unlike most previous power estimation tools for embedded applications, we explicitly consider and model the energy overhead of a host's OS and an application's runtime platform (e.g., JVM) incurred in facilitating and managing the execution of software components. This further enhances the accuracy of our framework in estimating a distributed software system's energy consumption. Another contribution of our work is its ability to adjust energy consumption estimates at runtime efficiently and automatically, based on monitoring the changes in a small number of easily tracked system parameters (e.g., size of data exchanged over the network, inputs to a component's interfaces, invocation frequency of each interface, etc.).

In the remainder of this paper we first present the related research in the energy estimation and measurement areas (Section 2). We then introduce our energy estimation framework (Section 3) and detail how it is applied to component-based Java systems (Section 4). This is followed by our evaluation strategy (Section 5) and results (Section 6). The paper concludes with a discussion of planned applications of this research (Section 7).

2 Related Work

Several studies have profiled the energy consumption of Java Virtual Machine (JVM) implementations. Farkas et al. [3] have measured the energy consumption of the Itsy Pocket Computer and the JVM running on it. They have discussed different JVMs' design trade-offs and measured their impact on the JVM's energy consumption. Lafond et al. [11] have showed that the energy required for memory accesses usually accounts for 70% of the total energy consumed by the JVM. However, none of these studies suggest a model that can be used for estimating the energy consumption of a distributed Java-based system.

There have been several tools that estimate the energy consumption of embedded operating systems (OSs) or applications. Tan et al. [19] have investigated the energy behaviors of two widely used embedded OSs, μC/OS [10] and Linux, and suggested their quantitative macro-models, which can be used as OS energy estimators. Sinha et al. [16] have suggested a web-based tool, *JouleTrack*, for estimating the energy cost of an embedded software running on StrongARM SA-1100 and Hitachi SH-4 microprocessors. While they certainly informed our work, we were unable to use these tools directly in our targeted Java domain because none of them provide generic energy consumption models, but instead have focused on individual applications running on specific OSs and platforms.

Recently, researchers have attempted to characterize the energy consumption of the Transmission Control Protocol (TCP) [15]. Singh et al. [15] measured the energy consumption of variants of TCP (i.e., Reno, Newreno, SACK, and ECN-ELFN) in ad-hoc networks, and showed that ECN-EFLN has a lower energy cost than the others. These studies also show that, since TCP employs a complicated mechanism for congestion control and error recovery, modeling its exact energy consumption remains an open problem. While we plan to incorporate into our framework the future advancements in this area, as detailed in the next section we currently rely on the User Datagram Protocol (UDP), which does not provide any support for congestion control, retransmission, error recovery, and so on.

Several studies [4,21] have measured the energy consumption of wireless network interfaces on handheld devices that use UDP for communication. They have shown that the energy usage by a device due to exchanging data over the network is directly linear to the size of data. We use these experimental results as a basis for defining a component's communication energy cost.

Finally, this research builds on our previous works [12,13], where we have outlined the architecture of the framework [12], and the overall energy estimation process [13]. In this paper, we provide a comprehensive and detailed description of the framework, runtime refinement of its estimates, its practical applications, and an extensive evaluation of its accuracy in the context of several applications.

3 Energy Consumption Framework

We model a distributed software system's energy consumption at the level of its components. A component is a unit of computation and state. In a Java-based application, a component may comprise a single class or a cluster of related classes. The energy cost of a software component consists of its *computational* and *communication* energy costs. The computational cost is mainly due to CPU processing, memory access, I/O operations, and so forth, while the communication cost is mainly due to the data exchanged over the network. In addition to these two, there is an additional energy cost incurred by an OS and an application's runtime platform (e.g., JVM) in the process of managing the execution of user-level applications. We refer to this cost as *infrastructure energy overhead*. In this section, we present our approach to modeling each of these three energy cost factors. We conclude the section by summarizing the assumptions that underlie our work.

3.1 Computational Energy Cost

In order to preserve a software component's abstraction boundaries, we determine its computational cost at the level of its public interfaces. A component's interface corresponds to a service it provides to other components. While there are many ways of implementing an interface and binding it to its caller (e.g., RMI, event exchange), in the most prevalent case an interface corresponds to a method. In Section 3.2 we discuss other forms of interface implementation and binding (e.g., data serialization over sockets).

As an example, Figure 1 shows a component c_1 on host H_1, c_1's provided interfaces, and the invocation of them by remote components. Given the energy cost *iCompEC* resulting from invoking an interface I_i, and the total number b_i of invocations for the interface I_i, we can calculate the overall energy cost of a component c_1 with *n* interfaces (in *Joule*) as follows:

$$cCompEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCompEC(I_i, j) \tag{1}$$

In this equation, $iCompEC(I_i,j)$, the computational energy cost due to the j_{th} invocation of I_i, may depend on the input parameter values of I_i and differ for each invocation.

In Java, the effect of invoking an interface can be expressed in terms of the execution of JVM's 256 Java bytecode types, and its native methods. Bytecodes are platform-independent codes interpreted by JVM's interpreter, while native methods are library functions (e.g., `java.io.FileInputStream`'s `read( )` method) provided by JVM. Native methods are usually implemented in C and compiled into dynamic link libraries, which are automatically installed with JVM. JVM also provides a mechanism for synchronizing threads via an internal implementation of a *monitor*.

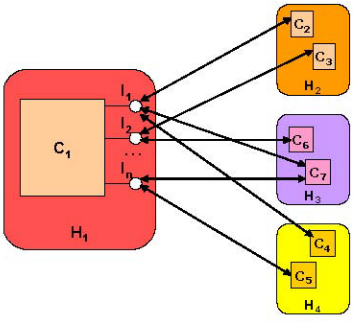


Fig. 1. Interactions among distributed components

Each Java statement maps to a specific sequence of bytecodes, native methods, and/or monitor operations. Based on the 256 bytecodes, *m* native methods, and monitor operations that are available on a given JVM, we can estimate the energy cost $iCompEC(I_i,j)$ of invoking an interface as follows:

$$iCompEC(I_i, j) = \left(\sum_{k=1}^{256} bNum_{k,j} \times bEC_k \right) + \left(\sum_{l=1}^{m} fNum_{l,j} \times fEC_l \right) + mNum_j \times mEC \tag{2}$$

where $bNum_{k,j}$ and $fNum_{l,j}$ are the numbers of each type of bytecode and native method, and $mNum_j$ is the number of monitor operations executed during the j_{th} invocation of I_i. bEC_k, fEC_l, and mEC represent the energy consumption of executing a given type of bytecode, a given type of native method, and a single monitor operation, respectively. These values must be measured before Equation 2 can be used. Unless two platforms have the same hardware setup, JVMs, and OSs, their respective values for bEC_k, fEC_l, and mEC will likely be different. We will explain how these values can be obtained in Section 5.

3.2 Communication Energy Cost

Two components may reside in the same address space and thus communicate locally, or in different address spaces and communicate remotely. When components are part of the same JVM process but running in independent threads, the communication among the threads is generally achieved via native method calls (e.g., `java.lang.Object`'s `notify ()` method). A component's reliance on native methods has already been accounted for in calculating its computational cost from Equation 2. When components run as separate JVM processes on the same host, Java sockets are usually used for their communication. Given that JVMs generally use native methods (e.g., `java.net.SocketInputStream`'s `read ()`) for socket communication, this is also captured by a component's computational cost.

In remote communication, the transmission of messages via network interfaces consumes significant energy. Given the communication energy cost *iCommEC* due to invoking an interface I_i, and the total number bi of invocations for that interface, we can calculate the overall communication energy consumption of a component c_1 with n interfaces (expressed in Joule) as follows:

$$cCommEC(c_1) = \sum_{i=1}^{n} \sum_{j=1}^{b_i} iCommEC(I_i, j) \quad . \tag{3}$$

In this equation, $iCommEC(I_i,j)$, the energy cost incurred by the j_{th} invocation of I_i, depends on the amount of data transmitted or received during the invocation and may be different for each invocation. Below we explain how we have modeled $iCommEC(I_i,j)$.

We focus on modeling the energy consumption due to the remote communication based on UDP. Since UDP is a light-weight protocol (e.g., it provides no congestion control, retransmission, and error recovery mechanisms), it is becoming increasingly prevalent in resource-constrained pervasive domains [2,20]. Previous research [4,21] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data. Based on this, we quantify the communication energy consumption due to the j_{th} invocation of component c_1's interface I_i on host H_1 by component c_2 on host H_2 as follows:

$$iCommEC(I_i, j) = \left(tEvtSize_{c_1,c_2} \times tEC_{H_1} + tS_{H_1}\right) + \left(rEvtSize_{c_1,c_2} \times rEC_{H_1} + rS_{H_1}\right) \tag{4}$$

Parameters *tEvtSize* and *rEvtSize* are the sizes (e.g., *KB*) of transmitted and received messages on host H_1 during the j_{th} invocation of I_i. The remaining parameters are host-specific. tEC_{H1} and rEC_{H1} are the energy costs (*Joule/byte*) on host H_1 while it transmits and receives a unit of data, respectively. tS_{H1} and rS_{H1} represent constant energy overheads associated with device state changes and channel acquisition [4].

In Equation 4, the energy values of *tEC*, *rEC*, *tS*, *rS* are constant and platform-specific.[1] The system parameters that need to be monitored on each host are only the sizes of messages exchanged (*tEvtSize* and *rEvtSize*, which include the overhead of network protocol headers). Note that transmission or receipt failures between the sender and receiver hosts do not affect our estimates: UDP does not do any processing to recover from such failures, while our framework uses the actual amount of data transmitted and received in calculating the communication energy estimates.

[1] We will elaborate on how these parameters are determined for an actual host in Section 6.2.

3.3 Infrastructure Energy Consumption

Once the computational and communication costs of a component have been calculated based on its interfaces, its overall energy consumption is determined as follows:

$$overallEC(c) = cCompEC(c) + cCommEC(c) \quad (5)$$

However, in addition to the computational and communication energy costs, there are additional energy costs for executing a Java component incurred by JVM's garbage collection and implicit OS routines. During garbage collection, all threads except the Garbage Collection (GC) thread within the JVM process are suspended temporarily, and the GC thread takes over the execution control. We estimate the energy consumption resulting from garbage collection by determining the average energy consumption rate *gEC* of the GC thread (*Joule/second*) and monitoring the total time *tGC* the thread is active (*second*). In Section 5 we describe how to measure the GC thread's execution time and its average energy consumption rate.

Since a JVM runs as a separate user-level process in an OS, it is necessary to consider the energy overhead of OS routine calls for facilitating and managing the execution of JVM processes. There are two types of OS routines:

1. explicit OS routines (i.e., system calls), which are initiated by user-level applications (e.g., accessing files, or displaying text and images on the screen); and
2. implicit OS routines, which are initiated by the OS (e.g., context switching, paging, and process scheduling).

Java applications initiate explicit OS routine calls via JVM's native methods. Therefore, Equation 2 already accounts for the energy cost due to the invocation of explicit OS routines. However, we have not accounted for the energy overhead of executing implicit OS routines. Previous research has shown that process scheduling, context switching, and paging are the main consumers of energy due to implicit OS routine calls [19]. By considering these additional energy costs, we can estimate the overall infrastructure energy overhead of a JVM process *p* as follows:

$$ifEC(p) = \left(tGC_p \times gEC\right) + \left(csNum_p \times csEC\right) + \left(pfNum_p \times pfEC\right) + \left(prNum_p \times prEC\right) \quad (6)$$

Recall that *gEC* is the average energy consumption rate of the GC thread, while *tGCp* is the time that the GC thread is active during the execution of process p. $csNum_p$, $pfNum_p$, and $prNum_p$ are, respectively, the numbers of context switches, page faults, and page reclaims that have occurred during the execution of process p. *csEC*, *pfEC*, and *prEC* are, respectively, the energy consumption of processing a context switch, a page fault, and a page reclaim. We should note that *csEC* includes the energy consumption of process scheduling as well as a context switch. This is due to the fact that in most embedded OSs a context switch is always preceded by process scheduling [19].

Since there is a singleton GC thread per JVM process, and implicit OS routines operate at the granularity of processes, we estimate the infrastructure energy overhead of a distributed software system in terms of its JVM processes. In turn, this helps us to estimate the system's energy consumption with higher accuracy. Unless two platforms have the same hardware configurations, JVMs, and OSs, the energy values of *gEC*, *csEC*, *pfEC*, and *prEC* on one platform may not be the same as those on the other platform. We will describe how these values can be obtained for an actual host in Section 5.

Once we have estimated the energy consumption of all the components, as well as the infrastructure energy overhead, we can estimate the system's overall energy consumption as follows:

$$systemEC = \sum_{i=1}^{cNum} overallEC(c_i) + \sum_{j=1}^{pNum} ifEC(p_j) \quad (7)$$

where *cNum* and *pNum* are, respectively, the numbers of components and JVM processes in the distributed software system.

3.4 Assumptions

In formulating the framework introduced in this section, we have made several assumptions. First, we assume that the configuration of all eventual target hosts is known in advance. This allows system engineers to closely approximate (or use the actual) execution environments in profiling the energy consumption of applications prior to their deployment and execution. As alluded above, and as will be further discussed in Sections 4 and 5, several elements of our approach (e.g., profiling the energy usage of a bytecode, assessing infrastructure energy costs) rely on the ability to obtain accurate energy measurements "off line".

Second, we assume that interpreter-based JVMs, such as Sun Microsystems' KVM [9] and JamVM [5], are used. These JVMs have been developed for resource-constrained platforms, and require much less memory than "just-in-time" (JIT) compilation-based JVMs. If a JIT-based JVM is used, the energy cost for translating a bytecode into native code "on the fly" would need to be added into Equation 2 since the JIT compilation itself happens while a Java application is being executed. We are currently investigating how our framework can be extended to JIT-based JVMs.

Third, we assume that the systems to which our framework is applicable will be implemented in "core" Java. In other words, apart from the JVM, we currently do not take into account the effects on energy consumption of any other middleware platform. While this does not prevent our framework from being applied on a very large number of existing Java applications, clearly in the future we will have to extend this work to include other middleware platforms.

Finally, we assume that the target network environment is a (W)LAN that consists of dedicated routers (e.g., wireless access points) and stationary or mobile hosts. This is representative of a majority of systems that rely on wireless connectivity and battery power today. In the case of mobile hosts, we assume that each host associates itself with an access point within its direct communication range and communicates with other hosts via dedicated access points. In this setting, there could be a hand-off overhead when mobile hosts move and change their associated access points. However, it is not the software system that causes this type of energy overhead, but rather the movement of the host (or user). Therefore, we currently do not consider these types of overhead in our framework. Note that in order to expand this work to a wireless ad-hoc network environment, we also need to consider the energy overhead of routing event messages by each host. This type of energy overhead can be accounted for by extending the infrastructure aspect of our framework. We plan to investigate this issue as part of our future work.

4 Energy Consumption Estimation

In this section, we discuss the framework's application for estimating a software system's energy cost at both during system construction-time and runtime.

4.1 Construction-Time Estimation

For construction-time estimation, we first need to characterize the computational energy cost of each component on its candidate hosts. To this end, we have identified three different types of component interfaces:

I. An interface (e.g., a date component's `setCurrentTime`) that requires the same amount of computation regardless of its input parameters.
II. An interface (e.g., a data compression component's `compress`) whose input size is proportional to the amount of computation required.
III. An interface (e.g., DBMS engine's `query`) whose input parameters have no direct relationship to the amount of computation required.

For a type I interface, we need to profile the number of bytecodes, native methods, and monitor operations only once for an arbitrary input. We can then calculate its energy consumption from Equation 2.

For type II interfaces, we generate a set of random inputs, profile the number of bytecodes, native methods, and monitor operations for each input, and then calculate its energy cost from Equation 2. However, the set of generated inputs does not show the complete energy behavior of a type II interface. To characterize the energy behavior of a type II interface for any arbitrary input, we employ multiple regression [1], a method of estimating the expected value of an output variable given the values of a set of related input variables. By running multiple regression on a sample set of input variables' values (i.e., each generated input for a type II interface) and the corresponding output value (the calculated energy cost), it is possible to construct an equation that estimates the relationship between the input variables and the output value.

Interfaces of type III present a challenge as there is no direct relationship between an interface's input and the amount of computation required, yet a lot of interface implementations fall in this category. For type III interfaces with a set of finite execution paths, we use symbolic execution [8], a program analysis technique that allows using symbolic values for input parameters to explore program execution paths. We leverage previous research [7], which has suggested a generalized symbolic execution approach for generating test inputs covering all the execution paths, and use these inputs for invoking a type III interface. We then profile the number of bytecodes, native methods, and monitor operations for each input, estimate its energy cost from Equation 2, and finally calculate the interface's average energy cost by dividing the total energy cost by the number of generated inputs.

The above approach works only for interfaces with finite execution paths, and is infeasible for interfaces whose implementations have infinite execution paths, such as a DBMS engine. We use an approximation for such interfaces: we automatically invoke the interface with a large set of random inputs, calculate the energy cost of the interface for each input via Equation 2, and finally calculate the average energy consumption of the interface by dividing the total consumption by the number of random inputs. This approach will clearly not always give a representative estimate of the

interface's actual energy consumption. Closer approximations can be obtained if an interface's expected runtime context is known (e.g., expected inputs, their frequencies, possible system states, and so on). As we will detail in Sections 4.2, we can refine our estimates for type III interfaces by monitoring the actual amount of computation required at runtime.

To estimate the communication energy cost of an interface, we rely on domain knowledge (e.g., the known types of input parameters and return values) to predict the average size of messages exchanged due to an interface's invocation. Using this data we approximate the communication energy cost of interface invocation via Equation 4. Finally, based on the computational and communication energy costs of interfaces, we estimate the overall energy cost of a component on its candidate host(s) using Equations 1, 3, and 5.

Before estimating the entire distributed system's energy cost, we also need to determine the infrastructure's energy overhead, which depends on the deployment of the software (e.g., the number of components executing simultaneously on each host). Unless the deployment of the system's components on its hosts is fixed a priori, the component-level energy estimates can help us determine an initial deployment that satisfies the system's energy requirements (e.g., to avoid overloading an energy-constrained device). Once an initial deployment is determined, from Equation 6 we estimate the infrastructure's energy cost. We do so by executing all the components on their target hosts simultaneously, with the same sets of inputs that were used in characterizing the energy cost of each individual component. Finally, we determine the system's overall energy cost via Equation 7.

4.2 Runtime Estimation

Many systems for which energy consumption is a significant concern are long-lived, dynamically adaptable, and mobile. An effective energy cost framework should account for changes in the runtime environment, or due to the system's adaptations. Below we discuss our approach to refining the construction-time estimates after the initial deployment.

The amount of computation associated with a type I interface is constant regardless of its input parameters. If the sizes of the inputs to a type II interface significantly differ from construction-time estimates, new estimates can be calculated efficiently and accurately from its energy equation generated by multiple regression. Recall from Section 4.1 that for type III interfaces our construction-time estimates may be inaccurate as we may not be able to predict the frequency of invocation or the frequency of the execution paths taken (e.g., the exception handling code). Therefore, to refine a type III interface's construction-time estimates, the actual amount of runtime computation (i.e., number of bytecodes, native methods, and monitor operations) must be monitored. In Section 5.4 we present an efficient way of monitoring these parameters.

For the communication cost of each component, by monitoring the sizes of messages exchanged over network links, their effects on each interface's communication cost can be determined, and a component's energy cost can be updated automatically.

Finally, the fact that the frequency at which interfaces are invoked may vary significantly from what was predicted at construction-time, and the fact that the system may be adapted at runtime, may result in inaccurate construction-time infrastructure

energy estimates. Therefore, the GC thread execution time and the number of implicit OS routines invoked at runtime must also be monitored. We discuss the overhead of this monitoring in detail in Section 5.4. Based on the refined estimates of each interface's computational and communication costs, and of the infrastructure's energy overhead, we can improve (possibly automatically) our construction-time estimates of distributed systems at runtime.

5 Evaluation Strategy

This section describes our evaluation environment, the tools on which we have relied, and the energy measurement and monitoring approaches we have used.

5.1 Experimental Setup

To evaluate the accuracy of our estimates, we need to know the actual energy consumption of a software component or system. To this end, we used a digital multimeter, which measures the factors influencing the energy consumption of a device: voltage and current. Since the input voltage is fixed in our experiments, the energy consumption can be measured based on the current variations going from the energy source to the device.

Figure 2 shows our experimental environment setup that included a Compaq iPAQ 3800 handheld device running Linux and Kaffe 1.1.5 JVM [6], with an external 5V DC power supply, a 206MHz Intel StrongARM processor, 64MB memory, and 11Mbps 802.11b compatible wireless PCMCIA card. We also used an HP 3458-a digital multimeter. For measuring the current drawn by the iPAQ, we connected it to the multimeter, which was configured to take current samples at a high frequency. A data collection computer controlled the multimeter and read the current samples from it.

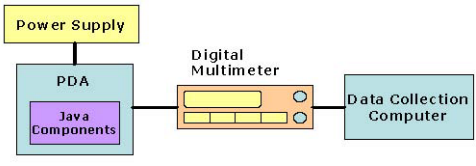


Fig. 2. Experimental setup

5.2 Selecting Java Components

We have selected a large number of Java components with various characteristics for evaluating our framework. They can be categorized as follows: 1) *Computation-intensive components* that require a large number of CPU operations. (e.g., encryption/decryption, data compression); 2) *Memory-intensive components* that require large segments of memory. (e.g., Database components); 3) *Communication-intensive components* that interact frequently with other components over a network (e.g., FTP component).

For illustration, Table 1 shows a cross-section of the Java components used in our evaluation. These components vary in size and complexity (`HSQLDB` is the largest, with more than 50,000 SLOC, while Jess is somewhat smaller, with approximately 40,000 SLOC). The source code of `Jess`, `HSQLDB`, and `IDEA` components can be found at Source Forge [18], while the source code of the other components from Table 1 was obtained from Source Bank [17].

5.3 Measurement

Prior to the deployment, we first need to measure the energy cost on a target platform of each bytecode, native method, monitor operation, and implicit OS routine, as well as the average consumption rate during garbage collection. For each bytecode we generate a Java class file that executes that bytecode 1000 times. We also create a skeleton Java class with no functionality, which is used to measure the energy consumption overhead of executing a class file. We use the setup discussed in Section 5.1 for measuring the actual energy cost of executing both class files. We then subtract the energy overhead *E1* of running the skeleton class file from the energy cost *E2* of the class file with the profiled bytecode. By dividing the result by 1000, we get the average energy consumption of executing the bytecode. Similarly, for measuring the energy consumption of each native method, we generate a class file invoking the native method and measure its actual energy consumption *E3*. Note that when JVM executes this class file, several bytecodes are also executed. Therefore, to get the energy cost of a native method, we subtract (*E1* + energy cost of the bytecodes) from *E3*. For a monitor operation, we generate a class file invoking a method that should be synchronized among multiple threads, and measure its energy consumption *E4*. Since several bytecodes are also executed during the invocation, we can get the energy cost of a monitor operation by subtracting (*E1* + energy cost of the bytecodes) from *E4*.

Table 1. A cross-section of Java components used in evaluation

Component	Description
SHA, MD5, IDEA	Components that encrypt or decrypt messages by using SHA, MD5, and IDEA algorithms
Median filter	Component that creates a new image by applying a median filter
LZW	Data compression/decompression component implementing the LZW algorithm
Sort	Quicksort component
Jess	Java Expert Shell System based on NASA's CLIPS expert shell system
DB	HSQLDB, a Java-based database engine
Shortest Path	Component that finds the shortest path tree with the source location as root
AVL, Linked list	Data structure components that implement an AVL tree and a linked list

To measure the energy cost of implicit OS routines, we employ the approach suggested by Tan et al. [19], which captures the energy consumption behavior of embedded operating systems. This allows us to determine the energy cost of major implicit OS routine calls, such as context switching, paging, and process scheduling. Due to space constraints we cannot provide the details of this approach; we point the interested readers to [19]. Finally, for getting the average energy consumption rate of the GC thread, we execute over a given period of time a simple Java class file that creates a large number of "dummy" objects, and measure the average energy consumption rate during the garbage collection.

5.4 Monitoring

Since we need to monitor the numbers of bytecodes, native methods, monitor operations, and implicit OS routines, as well as the GC thread execution time, we instrumented the Kaffe 1.1.5 JVM to provide the required monitoring data. Since the monitoring activity itself also consumes energy, we had to ensure that our monitoring mechanism is as light-weight as possible. To this end, we modified Kaffe's source

code by adding: 1) an integer array of size 256 for counting the number of times each bytecode type is executed; 2) integer counters for recording the number of times the different native methods are invoked; and 3) an integer counter for recording the number of monitor operations executed.

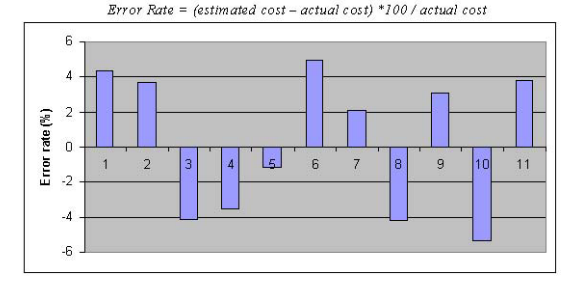
As mentioned earlier, this monitoring is only used for type III interfaces. We also added a timer to Kaffe's GC module to keep track of its total execution time. This timer has a small overhead equivalent to two system calls (for getting the times at the beginning and at the end of the GC thread's execution). For the number of implicit OS routines, we simply used the facilities provided by the OS. Since both Linux and Windows by default store the number of implicit OS routines executed in each process's Process Control Block, we did not introduce any additional overhead. We have measured the energy overhead due to these monitoring activities for the worst case (i.e., type III interfaces). The average energy overhead compared with the energy cost without any monitoring was 3.8%. Note that this overhead is transient: engineers can choose to monitor systems during specific time periods only (e.g., whenever any changes occur or are anticipated in the system or its usage).

6 Evaluation Results

In this section, we present the results of evaluating our framework.

6.1 Computational Energy Cost

To validate our computational energy model, we compare the values calculated from Equation 2 with actual energy costs. All actual energy costs have been calculated by subtracting the infrastructure energy overhead (Equation 6) from the energy consumption measured by the digital multimeter. As an illustration, Figure 3 shows the results of one series of executions for the components of Table 1. In this case, for each component we executed each of its interfaces 20 times with different input parameter values, and averaged the discrepancies between the estimated and actual costs (referred to as "error rate" below). The results show that our estimates fall within 5% of the actual energy costs. These results are also corroborated by additional experiments performed on these as well as a large number of other Java components [17,18].



1	2	3	4	5	6	7	8	9	10	11
SHA	MD5	IDEA	Median Filter	LZW	Sort	Jess	DB	Shortest path	AVL	Linked List

Fig. 3. Error rates for the components in Table 1

In addition to executing components of Table 1 in isolation, we have run these components simultaneously in different sample applications. Recall that, since each component is running in a separate JVM process, the energy overhead due to implicit OS routines is higher when multiple components are running simultaneously than

when each is running in isolation. Figure 4 shows the error rates of our computational energy model as the number of simultaneously running components increases. The experimental results show that, despite the increased infrastructure overhead, our estimates usually fall within 4% of the actual energy costs.

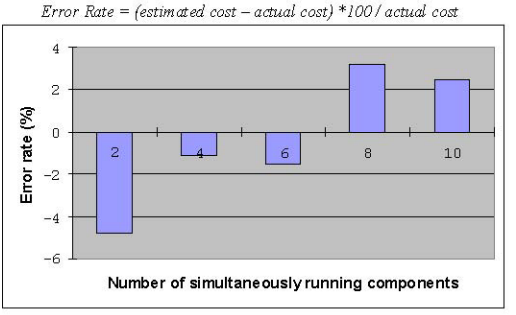
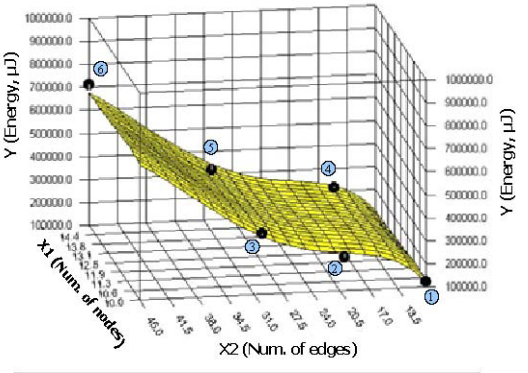


Fig. 4. Error rates with respect to the number of simultaneously running components

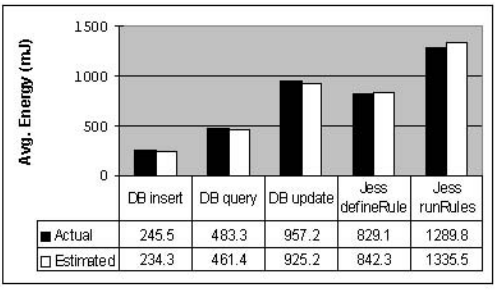
As discussed in Section 4.1, multiple regression can be used for characterizing the energy cost of invoking type II interfaces. For this we used a tool called DataFit. In measurements we conducted on close to 50 different type II interfaces, our estimates of their energy cost have been within 5% of the actual energy costs. As an illustration, Figure 5 shows the graph generated by DataFit for the `find` interface of the `Shortest Path` component, using 20 sets of sample values for `find`'s input parameters (x_1 and x_2), and the resulting energy costs (y) estimated by Equation 3. Several actual energy costs are shown for illustration as the discrete points on the graph.



	Point 1	Point 2	Point 3	Point 4	Point 5	Point 6
Actual (µJ)	122688	245395	364891	215760	316622	710709
Estimated (µJ)	119767	261905	351300	224201	341570	677319

Fig. 5. Multiple regression for the *find* interface of the Shortest Path Component

For estimating the energy consumption of type III interfaces, as discussed previously we generated a set of random inputs, estimated the energy cost of invoking each interface with the inputs using Equation 3, and calculated its average energy consumption. Figure 6 compares the average energy consumption of each interface for the DB and Jess components calculated using our framework with the interface's actual average energy consumption. The results show that our estimates are within 5% of the actual average energy costs. Recall that these design-time energy estimates can be refined at runtime by monitoring the numbers



	DB insert	DB query	DB update	Jess defineRule	Jess runRules
Actual	245.5	483.3	957.2	829.1	1289.8
Estimated	234.3	461.4	925.2	842.3	1335.5

Fig. 6. Accuracy of the framework for type III interface of *DB* and *Jess* components

of bytecodes, native methods, and monitor operations executed. For example, for a scenario that will be detailed in Section 6.3, we refined the construction-time energy estimate for the DB query interface at runtime, reducing the error rate to under 2.5%.

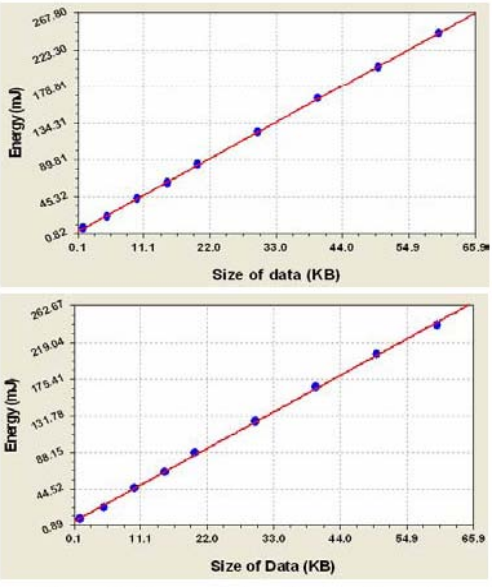


Fig. 7. Transmission (top) and receipt (bottom) energy estimation on an iPAQ

6.2 Communication Energy Cost

For evaluating the communication energy cost, we use a wireless router for the iPAQ to communicate with an IBM ThinkPad X22 laptop via a UDP socket implementation over a dedicated wireless network. Recall from Section 3.2 that several parameters (*tEC*, *rEC*, *tS*, and *rS*) from Equation 4 are host-specific. To quantify these parameters for the iPAQ, we created two Java programs that exchange messages via UDP sockets, and executed them on the iPAQ and the laptop. We then used the digital multimeter to measure the actual energy cost E on the iPAQ as a result of transmitting and receiving a sample set of messages of various sizes to/from the laptop. Since several bytecodes and native methods (e.g., `java.net.SocketInputStream`'s `read()` method) are executed during the program execution on the iPAQ, we subtract their energy costs from E to get the energy consumption of a wireless interface card on the iPAQ. Based on these results, we used multiple regression to find equations that capture the relationship between the input (size of the transmitted or received data x) and the output (actual energy consumption y of a wireless interface card on the iPAQ):

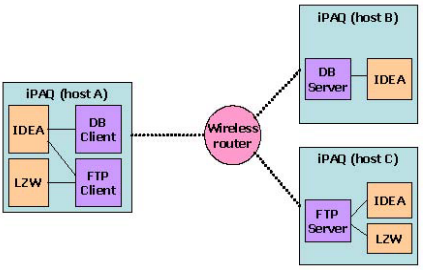


Fig. 8. A distributed Java-based system comprising three hosts

$$y_t(mJ) = 4.0131 * x_t(KB) + 3.1958 \quad (8)$$

$$y_r(mJ) = 3.9735 * x_r(KB) + 5.3229 \quad (9)$$

We used the generated equations to quantify the host-specific parameters in Equation 4. For example, the size of transmitted data x_t in Equation 8 represents tEvtSize in Equation 4. The constant energy cost of 3.1958 represents the parameter tS in Equation 4, which is independent of the size of transmitted data. The variables tEC is captured by the constant factor 4.0131. Figure 7 shows two graphs plotted for Equations 9 and 10, which represent the framework's estimates. As shown, the estimates,

which are depicted by the discrete points are within 3% of the actual energy costs.

6.3 Overall Energy Cost

We have evaluated our framework over a large number of applications. Figure 8 shows one example such application deployed across three iPAQ hosts. TheseiPAQ devices communicate with each other via a wireless router. Each software component interacts with the other components via a UDP socket. A line between two components (e.g., `IDEA` and `FTP Client` on host A) represents an interaction path between them. The `FTP Client` and `Server` components used in our evaluation are UDP-based implementations of a general purpose FTP. We have used several execution scenarios in this particular system. For example, `DB Client` component on host A may invoke the query interface (i.e., type III interface) of the remote DB Serveron host B; in response, DB Server calculates the results of the query, and then invokes `IDEA`'s `encrypt` interface (i.e., type II interface) and returns the encrypted results to `DB Client`; finally, `DB Client` invokes the decrypt interface (i.e., type II interface) of its collocated `IDEA` component to get the results.

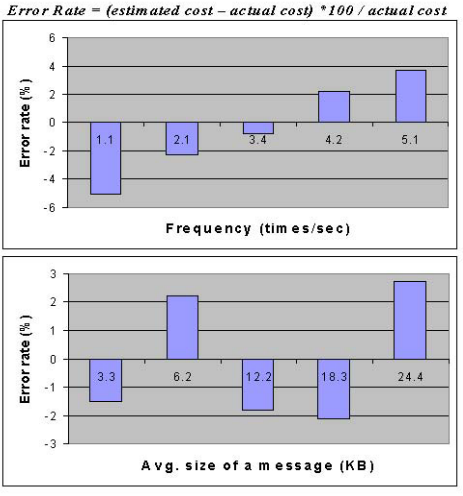


Fig. 9. The framework's error rates with respect to the interaction frequency (top) and The average size of a message (bottom)

We executed the above software system by varying the frequencies and sizes of exchanged messages, measured the system's overall energy cost, and compared it with our framework's runtime estimates. As shown in Figure 9, our estimates fall within 5% of the actual costs regardless of interaction frequencies and the average size of messages. In addition, we have evaluated our framework for a large number of additional distributed applications, increasing the numbers of components and hosts [14], and had similar results as shown in Figure 10.

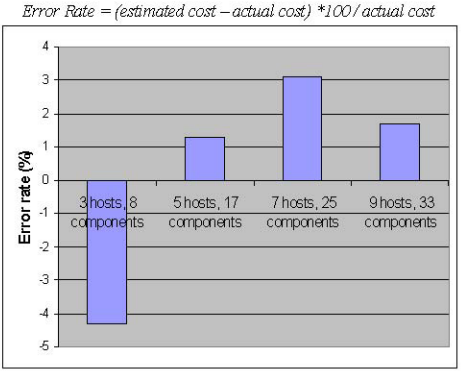


Fig. 10. Error rates of the framework with respect to the numbers of hosts and comps

7 Conclusion

We have presented a framework for estimating the energy cost of Java-based software systems. Our primary objective in devising the framework has been to enable an

engineer to make informed decisions, such that the system's energy consumption is reduced and the lifetime of the system's critical services increases. In a large number of distributed application scenarios the framework has shown very good precision, giving results that have been within 5% (and often less) of the actually measured power losses incurred by executing the software. We consider the development and evaluation of the framework to be a critical first step in pursuing several avenues of further work. As part of our future work, we plan to investigate the applications of the framework to various types of architectural decisions that could improve a system's energy usage, such as off-loading of software components, adapting components, modifying communication protocols, and so on.

Acknowledgements

This material is based upon work supported partially by the National Science Foundation under Grant Numbers 0312780 and 0820222. Effort also partially supported by the Bosch Research and Technology Center.

References

1. Allison, P.D.: Multiple regression. Pine Forge Press (1999)
2. Drytkiewicz, W., et al.: Prest: a REST-based protocol for pervasive systems. In: IEEE International Conference on Mobile Adhoc and Sensor Systems (2004)
3. Farkas, K.I., et al.: Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In: ACM SIGMETRICS, New York (2000)
4. Feeney, L.M., et al.: Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In: IEEE INFOCOM, Anchorage, AL (2001)
5. JamVM 1.3.2 (2006), `http://jamvm.sourceforge.net/`
6. Kaffe 1.1.5 (2005), `http://www.kaffe.org/`
7. Khurshid, S., et al.: Generalized Symbolic Execution for Model Checking and Testing. In: Int'l Conf. on Tools and Algorithms for Construction and Analysis of Systems, Warsaw, Poland (April 2003)
8. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976)
9. KVM (2005), `http://java.sun.com/products/cldc/wp/`
10. Labrosse, J.J.: MicroC/OS-II: The Real-Time Kernel. CMP Books (2002)
11. Lafond, S., et al.: An Energy Consumption Model for An Embedded Java Virtual Machine. In: Virtual Machine Research and Technology Symposium (2006)
12. Seo, C., et al.: An Energy Consumption Framework for Distributed Java-Based Systems. In: Int'l Conf. on Automated Software Engineering, Atlanta, Georgia (November 2007)
13. Seo, C., et al.: Estimating the Energy Consumption in Pervasive Java-Based Systems. In: Int'l Conf. on Pervasive Computing and Communication, Hong Kong (March 2008)
14. Seo, C.: Prediction of Energy Consumption Behavior in Component-Based Distributed Systems. Ph.D. Dissertation, University of Southern California (April 2008)
15. Singh, H., et al.: Energy Consumption of TCP in Ad Hoc Networks. Wireless Networks 10(5) (2004)

16. Sinha, A., et al.: JouleTrack - A Web Based Tool for Software Energy Profiling. In: Design Automation Conference (2001)
17. SourceBank, http://archive.devx.com/sourcebank/
18. sourceForge.net, http://sourceforge.net/
19. Tan, T.K., et al.: Energy macromodeling of embedded operating systems. ACM Trans. on Embedded Comp. Systems (2005)
20. UPnP Device Architecture (2007), http://www.upnp.org/
21. Xu, R., et al.: Impact of Data Compression on Energy Consumption of Wireless-Networked Handheld Devices. In: Int'l Conf. on Distributed Computing Systems, Rhode Island (2003)

Synthesis of Connectors from Scenario-Based Interaction Specifications

Farhad Arbab and Sun Meng*

CWI, Kruislaan 413, Amsterdam, The Netherlands
{Farhad.Arbab,Meng.Sun}@cwi.nl

Abstract. The idea of synthesizing state-based models from scenario-based interaction specifications has received much attention in recent years. The synthesis approach not only helps to significantly reduce the effort of system construction, but it also provides a bridge over the gap between requirements and implementation of systems. However, the existing synthesis techniques only focus on generating (global or local) state machines from scenario-based specifications, while the coordination among the behavior alternatives of services/components in the systems is not considered. In this paper we propose a novel synthesis technique, which can be used to generate constraint automata specification for connectors from scenario specifications. Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we define an algebraic framework for building constraint automata by exploiting the algebraic structure of UML sequence diagrams.

Keywords: Connector, Reo, Constraint Automata, Scenario-based Specification, UML, Synthesis.

1 Introduction

Service-oriented computing (SOC) [19] has now become the prominent paradigm for distributed computing, creating opportunities for service providers and application developers to use services as fundamental elements in their application development processes. Services consist of autonomous, platform-independent computational entities which can be described, published, categorized, discovered, and dynamically assembled for developing complex and evolvable applications that may run on large-scale distributed platforms. Such systems, which typically are heterogeneous and geographically distributed, usually exploit communication infrastructures whose topologies frequently vary and allow their components to connect to or detach from the systems at any moment. It is well-known that most service-oriented applications need a collaborative behavior among services/components, and this implies complex coordination. Therefore, it is crucial to derive a correct coordination model which specifies a precise order and causality of the service actions (for example: a purchase should happen after a payment). Consider the slogan proposed in [9]:

$$application = computation + coordination$$

* Corresponding author.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 114–129, 2008.

This separation is effective from the software engineering perspective: keeping the coordination issues separate from computation issues brings interaction to a higher level of abstraction and thus simplifies the programming task.

Compositional coordination models and languages provide a formalization of the "glue code" that interconnects the constituent components/services and organizes the communication and cooperation among them in a distributed environment. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. As an example, Reo [2,7], which is a channel-based exogenous coordination model, offers a powerful glue language for implementation of coordinating connectors based on a calculus of mobile channels, wherein complex coordinators, called connectors, are compositionally built out of simpler ones. Reo has been successfully applied in service-oriented computing [18,21]. Further details about Reo and its semantics can be found in [2,6,7].

Due to the increasing size and complexity of service oriented applications, construction of their coordination model remains a difficult task that requires considerable expertise. To solve this problem, the synthesis of component connectors from a given automaton specification has been investigated in [3]. One of the serious pragmatic limitations of the approach in [3] is the complexity of building the automaton specification of interactions in the first place. In this paper, we discuss the problem on *synthesizing connectors from scenario-based specifications* with Reo as our target implementation language. The input to this problem is a scenario specification and we aim to output a connector (Reo circuit) that correctly implements the coordination given in the specification. The synthesis of connectors from scenario-based specifications not only helps to significantly reduce the effort of coordination model construction, but it also links the approaches geared toward requirement analysis with those geared toward reasoning about system design at the architectural level.

Scenario-Based Specification languages have become increasingly popular over the last decade, through the widespread adoption of UML [22]. Scenarios describe how system components (in the broadest sense) and users interact in order to provide the system level functionality. Each scenario corresponds to a single temporal sequence of interactions among system components and provides a partial system description. Scenarios are close to users' understanding and they are often employed to refine use cases and provide an abstract view of the system behavior. Several notations have been proposed for the description of scenario-based models. The UML sequence diagrams (SDs) [22], message sequence charts (MSCs) [13], and Live Sequence Charts (LSCs) [10] are some of the most popular notations. In this paper we focus on scenarios represented as UML sequence diagrams.

Although there exist a number of works on formalizing and synthesizing from scenario-based specifications [11,12,17,23], most of them only focus on generating (global or local) state machines from scenario-based specifications. To the best of our knowledge, our work here is the first attempt to synthesize exogenous coordination models from scenario-based specifications. In [3] the problem of synthesizing Reo circuits from given automata specifications is discussed. Our work goes one step further toward bridging the gap between low-level implementations and abstract specifications by generating constraint automata (CA) specifications for coordination from

scenario-based specifications. Constraint automata were introduced in [7] as a formalism to capture the operational semantics of Reo, based on timed data streams which also constitute the foundation of the coalgebraic semantics of Reo [6]. The choice of Reo as the coordination language (and therefore constraint automata as its operational specification) is motivated by the fact that (1) it allows exogenous coordination and arbitrary user defined primitives, and (2) it is unique among other models in allowing arbitrary combination of synchrony and asynchrony in coordination protocols. This, for instance, facilitates multi-party transactions through Reo's inherent propagation of synchrony and exclusion constraints.

Inspired by the way UML2.0 SDs can be algebraically composed, we define an algebraic framework for composing constraint automata. Then we provide a family of constraint automata for basic sequence diagrams and show how to transform scenarios given as a composition of sequence diagrams into a composition of constraint automata. Beyond offering a systematic and semantically well founded synthesis approach, another benefit of our method lies in its flexibility: Due to the compositionality of the framework, modifying or replacing a given scenario has a limited effect on the synthesis process, thus fostering better traceability between the requirement and the dynamic architectural design of the system.

The remainder of this paper is organized as follows: Section 2 discusses the difference between endogenous and exogenous view of scenarios and show the motivation of our work. Section 3 briefly presents the relevant features of UML Sequence Diagrams. In Section 4 we present the algebraic framework for the synthesis of constraint automata from UML Sequence Diagrams. We discuss our work in Section 5, and compare it with related work in Section 6. Section 7 concludes the paper.

2 Motivation

The idea of using scenario descriptions, such as UML SDs, to generate operational models and/or executable code, of course, is not new. We briefly describe some related work in this area in Section 6. All such work share a common view of how scenarios are used to generate their operational models or code. For instance, consider a use case scenario in a simple bank ATM machine example, that involves a user, an ATM machine, and a number of remote processes, including a PIN verifier. The scenario describes that after feeding his card into the ATM machine, (1) the ATM machine asks the user to enter his PIN; (2) the ATM machine sends the user ID and his PIN to the PIN verifier; (3) the PIN verifier verifies the PIN to determine the validity of the access request; (4) the PIN verifier sends its (Allow/Deny/Confiscate) response back to the ATM machine; and (5) depending on the content of the response, the ATM machine proceeds to either allow or deny user access, or confiscates his card. The common view of the transformation of this scenario to executable code yields an ATM process and a PIN verifier process that directly communicate with each other: the ATM process contains a `send` $\langle ID, PIN \rangle$ `to PINverifier` instruction somewhere in its code, implementing step 2; and the PIN verifier process contains a `send response to ATM` instruction somewhere in its code, implementing step 4, above. These direct communication instructions implement the coordination protocol described in the scenario in an *endogenous* form.

Endogenous models implement/express a protocol only implicitly, through fragments of code in disparate entities that are hardwired to specifically realize that protocol. Suppose now that in a later version of this system, it is decided that the messages sent to the PIN verifier process must also be sent to some monitoring process, or instead of the PIN verifier to another more sophisticated process (e.g., one that tells the ATM machine to confiscate the user's card if the number of successive wrong PIN access attempts by the same card ID through all involved ATM machines within, say, a 24-hour sliding window, exceeds a threshold). Such changes to the protocol can easily be reflected in the SD specifications. However, implementing them requires invasive changes to a variety of independent software units that comprise the participating processes; worse, these changes may necessitate other less obvious changes that affect other software units and processes that are not directly involved in the modified portion of the protocol. Thus, small, "local" changes to a protocol can propagate through large spans of software units, touching them in ways that may invalidate their previously verified properties. Not only such invasive modifications are generally undesirable, in many cases they are impractical or even impossible, e.g., when they involve legacy code or third party providers.

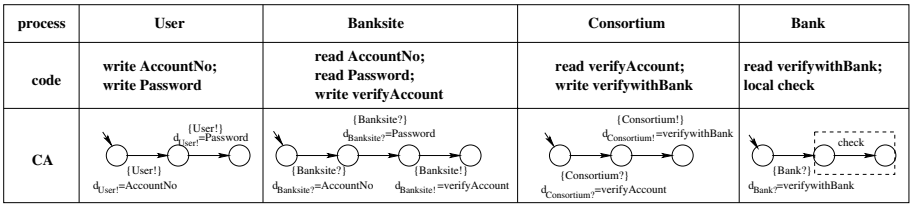


Fig. 1. Behavior Skeleton of processes in EnterPwd Scenario

Alternatively, the *exogenous* view of a scenario (e.g., the `EnterPwd` SD in the top right corner of Figure 2) imposes a purely local interpretation on each inter-process communication, implementing it as a pure I/O operation on each side, that allows processes to communicate anonymously, through the exchange of untargeted passive data. For instance, Figure 1 shows the behavior skeleton of the four processes involved in the `EnterPwd` SD, mentioned above. Observe that these processes are not hardwired to directly communicate with each other. Replacing exchanges of targeted messages with simple I/O localizes the range of their impact. This makes processes engaged in a protocol oblivious to changes in the protocol and their peers that do not directly impact their own behavior. Having expunged all communication/coordination concerns out of the parties involved, exogenous models relegate the task of conducting the required coordination to a (centralized or distributed) coordinator glue code that establishes the necessary communication links among the parties and engages them in the specified protocol. Reo is a good example of an exogenous coordination language that can be used to develop such glue code.

The scheme that we advocate in this paper for generating operational models and/or executable code from UML sequence diagrams uses the exogenous view in its interpretation of these scenario specifications. To our knowledge, this approach is novel and no other work has considered scenario specifications for exogenous coordination.

Our approach starts with the UML SD specification of a scenario and consists of the following steps: (1) generate a constraint automaton for the behavior skeleton of every process involved in the scenario *in its own ideal environment*, which assumes that all the other processes and the communication protocol are implemented correctly; (2) generate a constraint automaton for the coordination glue code that implements the protocol specified by the scenario; and (3) compose the individual processes and the glue code together to obtain a complete system that behaves as specified. The first step involves producing constraint automata representations for behavior skeletons such as in Figure 1 from individual process life-lines. This is trivial and we do not elaborate on it further. Instead, we focus in this paper on the other two steps.

Our approach is compositional and embodies the advantages inherent in the exogenous models of coordination: coordinated processes are strictly isolated from the dependencies on their environment and the details of the protocol that do not involve their individual behavior. This leads to more reusable processes, and an independent, explicit coordination protocol that can in turn be reused to coordinate these or similar processes. The ECT tools [1] can automatically generate executable code to produce a centralized implementation of the resulting composed coordination protocol. Alternatively, the resulting protocol can be used to synthesize a Reo circuit, as described in [3], that yields a distributed implementation of the coordinator.

3 UML Sequence Diagrams

UML sequence diagrams are one of the UML diagrams used to model the dynamic behavior of systems. SDs focus on the message interchange among a number of lifelines. A SD describes an interaction by focusing on the sequence of messages exchanged during a system run. See Figure 2 as an example of sequence diagrams which describe the interactions in the login phase of an on-line banking scenario. A UML SD is represented as a rectangular frame labeled by the keyword **sd** followed by the interaction name. The vertical lines in the SD represent lifelines for the individual participants in the interaction. Interactions among participants are shown as arrows called messages between lifelines.

A message defines a particular communication between two lifelines of an interaction. It can be either asynchronous (represented by an open arrow head) or synchronous (represented by a filled arrow head). Two special kinds of messages are *lost* and *found* messages, which are described by a small black circle at the arrow end (starting end respectively) of the message. Note that what we are interested in is the coordination among components/services, so we will consider only a subset of the UML2.0 SDs. For example, the internal behavior or action within the lifelines in SDs (like the *check* action in Figure 2) will not be considered in the synthesis process. Therefore, the synthesized result in our approach considers only the interaction aspect as opposed to a global state machine in which both behaviors of components and interactions among components are completely intertwined.

Before describing UML SDs in more detail, we first define the following notations.

- E is a set of events.
- L is a set of lifelines.

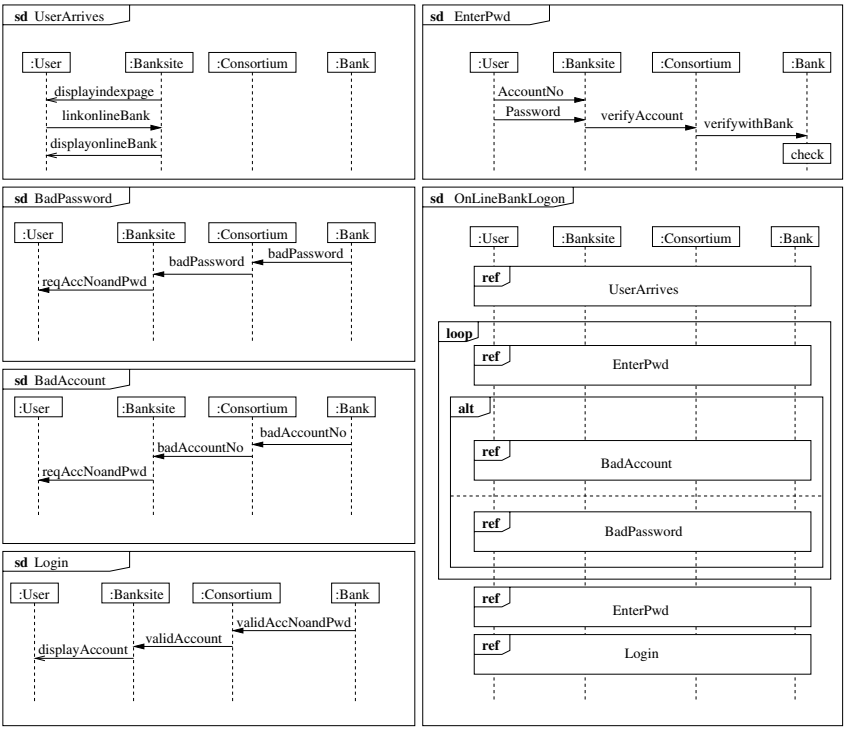


Fig. 2. Sequence Diagrams for the On-line Banking Example

- Let p, q range over L, and Σ be the set of communication actions executed by the participants in L. Such actions have the following forms, where p or q in an action can be replaced by $\bullet$ for lost or found messages:
  1. $\langle p! \rightarrow q, m\rangle$ - p sending asynchronous message m to q,
  2. $\langle p? \leftarrow q, m\rangle$ - p receiving asynchronous message m from q,
  3. $\langle p! \blacktriangleright q, m\rangle$ - p sending synchronous message m to q, and
  4. $\langle p? \blacktriangleleft q, m\rangle$ - p receiving synchronous message m from q.
- A lifeline introduces a totally ordered set of events $E' \subseteq E$.
- $\lambda : E \rightarrow \Sigma$ is a labeling function that assigns communication actions to the events on all lifelines.

Definition 1. *A basic Sequence Diagram is one of the following:*

- *asynchronous message:* $\langle\langle p! \rightarrow q, m\rangle, \langle q? \leftarrow p, m\rangle\rangle$
- *synchronous message:* $\langle\langle p! \blacktriangleright q, m\rangle, \langle q? \blacktriangleleft p, m\rangle\rangle$
- *lost message:* $\langle p! \rightarrow \bullet, m\rangle$
- *found message:* $\langle p? \leftarrow \bullet, m\rangle$

In the UML2.0 and later versions, SDs are enhanced by important control flow features. Thus, sequence diagrams can be composed by means of operators like **alt**, **par**, **strict**,

seq and **loop** to obtain more complex interactions. The operator **alt** designates that the combined SD represents a choice of behavior alternatives, where at most one of the operand SDs will be chosen. The operator **par** designates that the combined SD represents a parallel merge between the behaviors of the operand SDs. The operator **strict** designates that the combined SD represents a strict sequencing of the behaviors of the operands. The operator **seq** designates that the SD represents a weak sequencing between the behaviors of the operands (all events in the first operand situated on one lifeline must be executed before events of the second operand situated on the same lifeline, but the events on different lifelines from different operands may come in any order). The **loop** operator specifies an iteration of an interaction. As an example, the sequence diagram *OnLineBankLogon* in Figure 2 can be described as:

$$\begin{aligned} & OnLineBankLogon \\ =& \mathbf{strict}(UserArrives, \mathbf{loop}(\mathbf{strict}(Enterpwd, \\ & \mathbf{alt}(BadAccount, BadPassword))), Enterpwd, Login) \end{aligned}$$

where the SDs being used (like *UserArrives*) are further composed out of basic SDs.

4 Generating Constraint Automata from Sequence Diagrams

We can describe the method for generating connectors from scenario specifications as consisting of two main phases. The first phase concerns the generation of constraint automata from UML SDs. The second phase involves translation from constraint automata to Reo circuits. In this paper we focus on the first phase and refer to [3] for a detailed description of the second phase.

We propose an approach that generates constraint automata from scenarios via an algebraic framework that allows to switch from an algebraic composition of SDs to an algebraic composition of constraint automata. First, we extend the definition of constraint automata with the notion of final states, yielding the notion of *final state extended constraint automaton* (FSECA) for representing finite behavior. The general principle of our approach is to take each basic SD and produce a final state extended constraint automaton so that there exists a finite sequence of transitions corresponding to the scenario. Then we can use the composition operators to compose the automata from basic SDs. Finally, the final states in the resulting FSECA can be merged into its initial state to get the constraint automata model when loops on the scenarios are possible.

In the sequel, we assume a finite set $\mathcal{N}$ of nodes (ports), and $Data$ as a fixed, non-empty set of data that can be sent and received via channels. A data assignment denotes a function $\delta : N \rightarrow Data$ where $N \subseteq \mathcal{N}$. We use $DA(N)$ for the set of all data assignments for the node-set N. A symbolic representation of data assignments is given by using data constraints which consist of propositional formulas built from the atoms "$d_A \in P$", "$d_A = d_B$" or "$d_A = d$" plus standard Boolean connectors, where $A, B \in \mathcal{N}$, d_A is a symbol for the observed data item at node A and $d \in Data$, $P \subseteq Data$. We write $DC(N)$ to denote the set of data constraints that at most refer to the observed data items d_A at nodes $A \in N$, and DC for $DC(\mathcal{N})$. Logical implication induces a partial order $\leq$ on DC: $g \leq g'$ iff $g \Rightarrow g'$.

Definition 2. *A final state extended constraint automaton (FSECA) over the data domain Data is a tuple* $\mathscr{A} = (S, s_0, S_F, \mathscr{N}, \longrightarrow)$ *where* S *is a set of states, also called configurations,* $s_0 \in S$ *is its initial state,* $S_F \subseteq S$ *is a set of final states,* $\mathscr{N}$ *is a finite set of nodes,* $\longrightarrow \subseteq \bigcup_{N \subseteq \mathscr{N}} S \times \{N\} \times DC(N) \times S$*, called the transition relation.*

Note that the notion of final state has a meaning similar to its corresponding notion in classical automata, but here it also plays an addition role during composition of CA: it is used as a kind of merging state for some of the operators. In graphical representation of FSECA as in Figure 3, final states are represented by double circles.

The generation of a constraint automaton from a scenario specification is based on the generation of FSECA from basic SDs. For simplicity, we assume that for every participant p in the communication, the connector has at most one input port (denoted by $p!$) and one output port (denoted by $p?$) connected to p. This assumption can lead to a more abstract constraint automaton, while the approach can be easily extended to the case where one participant p is connected to multiple nodes in the connector.

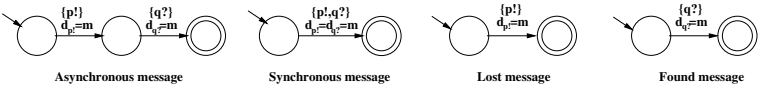


Fig. 3. Final State Extended Constraint Automata for Basic Sequence Diagrams

For the four basic Sequence Diagrams in Definition 1, the corresponding FSECA are given as in Figure 3.

In the following we define the operators $\mathbf{alt}_A$, $\mathbf{par}_A$, $\mathbf{strict}_A$, $\mathbf{seq}_A$ and $\mathbf{loop}_A$ for composition of FSECA. Final states introduced in Definition 2 will be necessary for formalizing these operators. A special FSECA used in the definitions, called the *empty FSECA*, denoted by $\mathscr{A}_\emptyset$, contains only a single state which is both an initial and a final state, and has no transitions. A FSECA is a loop if it is not empty and its initial state is a final state.

Let $\mathscr{A}_1 = (S_1, s_0^1, S_F^1, \mathscr{N}_1, \longrightarrow_1)$ and $\mathscr{A}_2 = (S_2, s_0^2, S_F^2, \mathscr{N}_2, \longrightarrow_2)$ be two final state extended constraint automata.

Definition 3. *The FSECA resulting from the alternative composition of* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *describes a choice between the behaviors of its operands.* $\mathbf{alt}_A(\mathscr{A}_1, \mathscr{A}_2) = (S, s_0, S_F, \mathscr{N}, \longrightarrow)$ *where*

- $S = \begin{cases} S_1 & \mathscr{A}_2 = \mathscr{A}_\emptyset \wedge \mathscr{A}_1 \neq \mathscr{A}_\emptyset \\ S_2 & \mathscr{A}_1 = \mathscr{A}_\emptyset \wedge \mathscr{A}_2 \neq \mathscr{A}_\emptyset \\ \{s_0\} & \mathscr{A}_1 = \mathscr{A}_\emptyset \wedge \mathscr{A}_2 = \mathscr{A}_\emptyset \\ S_1 \cup S_2 \cup \{s_0\} & s_0^i \in S_F^i \wedge \mathscr{A}_i \neq \mathscr{A}_\emptyset \text{ for } i = 1, 2 \\ S_1 \cup S_2 \setminus \{s_0^2\} & s_0^i \notin S_F^i \text{ for } i = 1, 2 \\ S_1 \cup S_2 & \text{otherwise} \end{cases}$
- $s_0 = \begin{cases} s & s \text{ is a new state} \wedge s_0^i \in S_F^i \wedge \mathscr{A}_i \neq \mathscr{A}_\emptyset \text{ for } i = 1, 2 \\ s_0^2 & s_0^2 \notin S_F^2 \wedge (s_0^1 \in S_F^1 \vee \mathscr{A}_1 = \mathscr{A}_\emptyset) \\ s_0^1 & \text{otherwise} \end{cases}$

- $S_F = (S_F^1 \cup S_F^2) \cap S$
- $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$
- $\longrightarrow = (\longrightarrow_1 \cap \bigcup_{N \subseteq \mathcal{N}} S \times \{N\} \times DC(N) \times S) \cup (\longrightarrow_2 \cap \bigcup_{N \subseteq \mathcal{N}} S \times \{N\} \times DC(N) \times S) \cup \{(s_0, N, g, s) \mid (s_0^i, N, g, s) \in \longrightarrow_i \text{ for } i = 1, 2\}$

In this definition and Definition 5, we have the identity element in the algebra, i.e., the empty FSECA, to make the operations more complete. However, in most cases for UML SDs, the operands are not empty, and the definitions can be simplified by removing the cases corresponding to empty operands. If one of the operands in an alternative composition is empty, then the resulting FSECA is the same as the non-empty one. The resulting FSECA is empty when both operands are empty. If both operands are non-empty, then the transitions in both automata will be kept in the resulting one, and the initial states will be merged together. If both of the initial states s_0^1 and s_0^2 in the two operands are final states, then a new initial state s_0 will be added in the resulting automaton, and the transitions with s_0^1 or s_0^2 as their source state will be replaced by transitions with the source state s_0. Otherwise, one of the two initial states will be kept as the initial state in the new automaton.

Definition 4. *The FSECA resulting from the parallel composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ describes a parallel merge of the behaviors of its operands.* $\mathbf{par}_A(\mathscr{A}_1, \mathscr{A}_2) = (S, s_0, S_F, \mathcal{N}, \longrightarrow)$ *where* $S = S_1 \times S_2$, $s_0 = \langle s_0^1, s_0^2 \rangle$, $S_F = S_F^1 \times S_F^2$, $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$, *and the transition relation* $\longrightarrow$ *in* $\mathbf{par}_A(\mathscr{A}_1, \mathscr{A}_2)$ *is defined by the interleaving rules:*

- *If* $s_1 \xrightarrow{N,g}_1 s_1'$, *then* $\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle s_1', s_2 \rangle$.
- *If* $s_2 \xrightarrow{N,g}_2 s_2'$, *then* $\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle s_1, s_2' \rangle$.

According to the UML specification, the **par** operator represents a parallel merge of the behaviors of its operands. The behavior of the two operands can be interleaved in any order as long as the ordering imposed within each operand as such is preserved.

Definition 5. *The FSECA resulting from the strict sequential composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ describes the behavior of the first operand followed by the behavior of the second one.* $\mathbf{strict}_A(\mathscr{A}_1, \mathscr{A}_2) = (S, s_0, S_F, \mathcal{N}, \longrightarrow)$ *where*

- $S = \begin{cases} S_1 \cup S_2 \setminus \{s_0^2\} & s_0^2 \notin S_F^2 \vee \mathscr{A}_2 = \mathscr{A}_\emptyset \\ S_2 & \mathscr{A}_1 = \mathscr{A}_\emptyset \\ S_1 \cup S_2 & \textit{otherwise} \end{cases}$
- $s_0 = \begin{cases} s_0^1 & \mathscr{A}_1 \neq \mathscr{A}_\emptyset \\ s_0^2 & \textit{otherwise} \end{cases}$
- $S_F = \begin{cases} S_F^1 \cup S_F^2 & s_0^2 \in S_F^2 \\ S_F^2 & \textit{otherwise} \end{cases}$
- $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$
- $\longrightarrow = \longrightarrow_1 \cup (\longrightarrow_2 \cap \bigcup_{N \subseteq \mathcal{N}} S \times \{N\} \times DC(N) \times S) \cup \{(f, N, g, s) \mid (s_0^2, N, g, s) \in \longrightarrow_2 \wedge f \in S_F^1\}$

If the first operand in a strict sequential composition is empty, the resulting automaton is the same as the second operand. If the initial state s_0^2 is not a final state in the second operand, it will be removed from the state space, and every transition from s_0^2 will be replaced by transitions from the final states in the first operand. That means, the behavior of the second operand will start after the behavior of the first operand finishes.

Before we define the weak sequential composition $\mathbf{seq}_A$, we first introduce another operation on FSECA: ordered composition $\boxtimes$. For a lifeline l in a SD, we introduce an order automaton $O_l = (S_l, s_0^l, s_F^l, \mathscr{M}, \longrightarrow_l)$ where the node set $\mathscr{M}$ consists of all the nodes connected to the lifeline, and all its transitions form a sequence $s_0^l \xrightarrow{M_1,g_1}_l s_1^l \cdots \xrightarrow{M_n,g_n}_l s_n^l = s_F^l$ [1]. For a FSECA $\mathscr{A} = (S, s_0, S_F, \mathscr{N}, \longrightarrow_{\mathscr{A}})$ and an order automaton $O = (T, t_0, T_F, \mathscr{M}, \longrightarrow_O)$, their phased composition $\mathscr{A} \boxtimes O$ is defined as

$$\mathscr{A} \boxtimes O = (S_{\mathscr{A}\boxtimes O}, \langle s_0, t_0\rangle, F, \mathscr{N}, \longrightarrow)$$

which is constructed as follows:

1. $S_{\mathscr{A}\boxtimes O} = \{\langle s_0, t_0\rangle\}$;
2. for $\langle s, t\rangle \in S_{\mathscr{A}\boxtimes O}$ and $s \xrightarrow{N_1,g_1}_{\mathscr{A}} s'$ a transition of $\mathscr{A}$, do
   - if $N_1 \cap \mathscr{M} = \emptyset$ then $\langle s, t\rangle \xrightarrow{N_1,g_1} \langle s', t\rangle$ is a transition of $\mathscr{A} \boxtimes O$, and $\langle s', t\rangle$ is a state of $\mathscr{A} \boxtimes O$.
   - if there exists a $t \xrightarrow{N_2,g_2}_O t'$ in O, where $N_1 \cap N_2 \neq \emptyset$ and $g_1 \wedge g_2$ is satisfiable, then $\langle s, t\rangle \xrightarrow{N_1,g_1} \langle s', t'\rangle$ is a transition of $\mathscr{A} \boxtimes O$, and $\langle s', t'\rangle$ is a state of $\mathscr{A} \boxtimes O$.
   - if for any transition $t \xrightarrow{N_2,g_2}_O t'$ with source state t in O, either $N_1 \cap N_2 = \emptyset \wedge N_1 \cap \mathscr{M} \neq \emptyset$ or $N_1 \cap N_2 \neq \emptyset \wedge g_1 \wedge g_2 = \mathbf{false}$, then there is no transition from $\langle s, t\rangle$ labelled by N_1, g_1.

   until no new states can be added to the state space $S_{\mathscr{A}\boxtimes O}$.

Note that the ordered composition is asymmetric, which is different from the product operation on constraint automata. This is due to the different roles played by the two operands. In this context, the automaton $\mathscr{A}$ describes all possible communication behavior, while the automaton O provides some requirements on the protocol by restricting the order of events. However, this operation is still compositional. That means, for any $\mathscr{A}$ and O_1, O_2, $(\mathscr{A} \boxtimes O_1) \boxtimes O_2 = (\mathscr{A} \boxtimes O_2) \boxtimes O_1$. (Detailed proofs for the compositionality is omitted here.) Therefore, we can easily extend the $\boxtimes$ operation to the case where $\mathscr{N}_1 \cap \mathscr{N}_2$ contains nodes of more than one lifeline and use $\mathscr{A} \boxtimes \{O_i\}_{1 \le i \le k}$ as an abbreviation for $(\cdots((\mathscr{A} \boxtimes O_1) \boxtimes O_2) \boxtimes \cdots \boxtimes O_k)$.

Definition 6. *The FSECA resulting from the weak sequential composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ can be reduced to a parallel merge when the operands contain disjoint sets of nodes. When the operands work on the same nodes, the weak sequential composition reduces to strict sequential composition.*

[1] Note that the order automaton can be derived from the order of events on a lifeline. If the order of events is not specified (for example, events in a coregion in UML SD), then it can be any possible order.

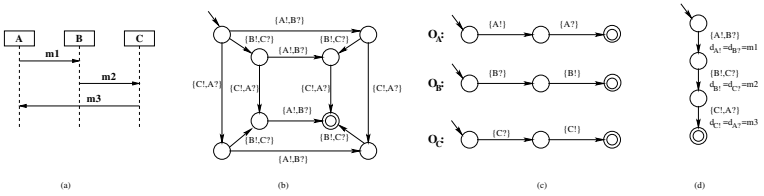


Fig. 4. Weak Sequential Composition

$$\mathbf{seq}_A(\mathscr{A}_1, \mathscr{A}_2) = \begin{cases} \mathbf{par}_A(\mathscr{A}_1, \mathscr{A}_2) & \mathscr{N}_1 \cap \mathscr{N}_2 = \emptyset \\ \mathbf{par}_A(\mathscr{A}_1, \mathscr{A}_2) \boxtimes \{O_{l_i}\} & \mathscr{N}_1 \cap \mathscr{N}_2 = \{l_i\} \end{cases}$$

where the transitions in O_{l_i} correspond to the events happening on lifeline l_i.

Example 1. As an example, consider the sequence diagram given in Figure 4(a). For each of the messages $m1, m2, m3$, we have a basic SD, and the sequence diagram can be defined as a weak sequential composition of the three basic SDs. We can first construct the parallel merge of the FSECA for the basic SDs. The resulting FSECA $\mathscr{A}$ is as shown in Figure 4(b) [2]. The order automaton for the corresponding lifelines are given in Figure 4(c). Then, according to the definition of $\boxtimes$, we can get the resulting FSECA $\mathscr{A} \boxtimes \{O_A, O_B, O_C\}$ for the whole SD as shown in Figure 4(d).

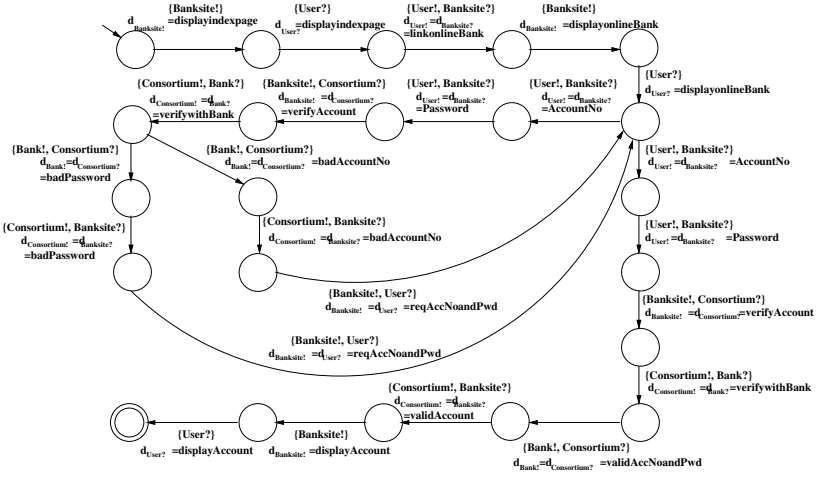


Fig. 5. Final State Extended Constraint Automaton for the OnLineBankLogon Scenario

Definition 7. *The FSECA resulting from a loop describes the iteration of the behavior of its operand.* $\mathbf{loop}_A(\mathscr{A}_1) = (S, s_0, S_F, \mathscr{N}, \longrightarrow)$ *where* $S = (S_1 \setminus S_F^1) \cup \{s_0^1\}$, $s_0 = s_0^1$, $S_F = \{s_0^1\}$, $\mathscr{N} = \mathscr{N}_1$, *and* $\longrightarrow = (\longrightarrow_1 \cap \bigcup_{N \subseteq \mathscr{N}} S \times \{N\} \times DC(N) \times S) \cup \{(s, N, g, s_0^1) \mid (s, N, g, f) \in \longrightarrow_1 \wedge f \in S_F^1\}$.

[2] For simplicity, here we omit the data constraints on all transitions, which can be derived easily from the SD.

After building a collection of FSECA for our basic sequence diagrams, the extension of the method to general SDs seems quite straight-forward. Let S be a SD constructed by composing a set of basic SDs $\{B_1, \cdots, B_k\}$ with the operators **alt**, **par**, **strict**, **seq** and **loop**. For each B_i, a FSECA $\mathscr{A}_i$ can be built using its corresponding automaton in Figure 3. Then the FSECA $\mathscr{A}_S$ for the SD S can be constructed by replacing every B_i with $\mathscr{A}_i$ and using the operators $\mathbf{alt}_A$, $\mathbf{par}_A$, $\mathbf{strict}_A$, $\mathbf{seq}_A$ and $\mathbf{loop}_A$ to replace **alt**, **par**, **strict**, **seq** and **loop**, respectively. As an example, Figure 5 shows the FSECA for the OnLineBankLogon scenario generated by composing the FSECA of its basic SDs.

5 Discussion

Every UML SD from which a FSECA is synthesized captures a single scenario, with all of its relevant actors accounted for. However, because of the algebraic operators in UML 2.0, it is possible to deal with the more interesting situations where systems may have many possible scenarios, and these scenarios are combined into a specification. The generation of constraint automata from scenario specifications provides a certain flexibility in the synthesis process. When we modify the scenario specification (for example, adding, removing or changing a sequence diagram), part of the previous synthesis result can be reused. So far, our synthesis approach focuses on generating the coordination model among components/services, instead of components or the whole system. The intuition is that, when we develop distributed applications, especially service-oriented applications running in wide area large-scale distributed environments, coordinating connectors play an important role for gluing the often pre-existing components/services together. Instead of synthesizing an implementation for separate components, we synthesize the connector that can be used to glue together and coordinate existing components/services in a distributed processing environment.

Although scenarios also describe the interactions among components, they cannot be used directly as a coordination language. Initially one would expect the synthesized state-based models (and thus the connectors) to have exactly the same set of behaviors as those depicted in the scenario specifications. However, this is not always the case. Scenarios can combine in unexpected ways and certain system behavior, not present in the scenario specifications, may appear in possible system implementations. To solve this problem, [27] proposed an approach to detect scenarios that appear in a synthesized model (implied scenarios) and then to enhance the set of requirements to include these implied scenarios. However, as pointed out in [29], the detection of an implied scenario can be done only through simulation, and some unspecified runs can be missed. Thus the number of implied scenarios can be infinite and the set of requirements may never converge toward a stable set of scenarios.

Therefore, synthesis from scenarios can be taken as an entry point toward more operational models such as CA. Since we can consider each sequence diagram as a sample interaction behavior of an application, it is clear that each interaction behavior described by a SD must match at least one run of an implementation. The constraint automata generated by synthesis can then be used to generate Reo circuits, which provide adequate communication mechanisms (synchrony, asynchrony, broadcast communication, etc.).

Then the scenarios can be used as tests to check if the Reo circuit provides appropriate communication with respect to the initial requirements.

In [3], the generation of Reo circuits from constraint automata also follows a compositional approach: for each basic expression $\langle N, g \rangle$, a Reo circuit $R_{\langle N,g \rangle}$ can be built and a family of operators can be used to combine them. Tools are also available for generating a constraint automaton from a Reo circuit. Both Reo circuits and constraint automata can serve as the coordinating glue code that composes black-box components into complex systems. The constraint automaton incarnation of a Reo circuit constitutes a centralized coordinator that reflects the global states of the system and can be substantially more efficient to run. On the other hand, a Reo circuit is inherently distributed, requiring no central authority or global state information, which makes it more suitable for systems composed of distributed components/services.

Like most program-generated code, the synthesis of a Reo circuit from a constraint automaton, as reported in [3], generally yields verbose circuits that do not "look natural" to the human eye. Using this method, then, to generate a Reo circuit from a constraint automaton (such as in Figure 5) synthesized from UML2 SDs yields Reo coordinator circuits that may not easily correlate back to their original SD specifications. The merit of synthesizing Reo circuits directly from SDs, which has been investigated in [5] lies in the greater structural fidelity between the resulting Reo circuits in this approach and their original SD specifications. However, for some SD operators like the negation operator **neg**, the behavior in the scenario is not permitted and it is impossible to synthesize the Reo circuit directly from the scenario. However, it is still possible to be dealt with by using CA. We can first construct the CA corresponding to a negative scenario according to the approach in Section 4, and then build its complement part, and finally generate the corresponding Reo circuits according to the approach in [3].

6 Related Work

Several semantics for scenario-based languages have been proposed, and a number of approaches for synthesis of state-based models from scenario descriptions have been developed. Some of these works provide algorithms for synthesizing state-based models from Message Sequence Charts. For example, [16] presents a state-chart synthesis algorithm, but the approach does not support High-Level Message Sequence Charts (HMSC), which provide a composition mechanism very close to UML2.0 SDs. The authors of [26,27] propose an approach to synthesize LTS models from MSC specifications, where the mechanism for communication between components is synchronous. The authors of [17] use MSCs for service specifications and propose an algorithm for synthesizing component automata from specifications.

In [11,12], the problem of synthesizing state machines from LSC models was tackled by defining the notion of consistency of an LSC model. A global system automaton can be constructed and then decomposed. However, this approach suffers from the state explosion problem due to the construction of the global system automaton, which is often huge in size because of the underlying weak partial ordering semantics of LSC. In [23], Sun and Dong combine the LSC notation with Z, and propose a synthesis approach for generating distributed finite state designs from the combined specifications.

The authors of [20] propose an interactive algorithm that can be used to generate flat state-charts from UML sequence diagrams. In [14], the authors also provide an interactive algorithm to generate state-charts from multiple scenarios expressed as UML collaborations. An algebraic approach was adopted in [29] to synthesize state-charts of components from sequence diagrams, but it takes only the operators **alt**, **seq** and **loop** into account, and does not consider any of the other UML2.0 operators on SDs. In [25], the existing LTS synthesis algorithms are extended to produce Modal Transition Systems from the combination of properties (expressed in temporal logic) and scenarios.

Regardless of the scenario notations used (MSC, LSC or UML), all these works focus only on generating the state-based models for separate components (or a global state machine for the whole system). These approaches differ from ours as (1) we are concerned about the coordination aspects in distributed applications instead of the behavior models for separate individual components, and (2) our synthesized connectors also provide the actual protocols used for communication among components/services in the system, and the components do not need to contain any protocol information. Therefore, when the communication protocol changes through system evolution, we need to change only the connector implementation without changing any components.

Another closely related work is the synthesis of adapters in component based systems. The authors of [28] propose an approach to modify the interaction mechanisms that are used to glue components together by integrating the interaction protocol into components. However, this approach acts only on the signature level. The work reported in [24] goes beyond the signature level and supports protocol transformations in the synthesis process, but the initial coordinator being synthesized behaves only as the "no-op" coordinator, which requires the assembly of new components to enhance its protocol for communication.

In [3], we have shown how to synthesize Reo circuits from CA specifications. The new contribution in this paper is that we go one step further and generates CA from scenario specifications represented by UML SDs. There is substantial benefit in this work which bridges the gap between requirements and implementation of coordination among services in service oriented application development.

7 Conclusion

In this paper we have presented an algebraic technique for constructing constraint automata from scenario specifications represented by UML SDs. This work aims to assist users and designers in the production of coordination models from scenario specifications. Our synthesis method produces a constraint automaton that can be used directly to generate a local executable glue code. It can also extend our previous work described in [3] that presents an algorithm for automatically synthesizing Reo circuits from constraint automata specifications. Together with the results in this paper, we can successfully derive a Reo circuit that coordinates the behavior of the components/services in a large-scale distributed application directly from its scenario specifications. Although we can synthesize Reo circuits directly from scenarios [5], using CA as the bridge between coordinators and scenario specifications still has potential benifits since we can use CA for model checking and testing properties of the connectors.

Among our next steps is the automation of the synthesis approach. We already have a set of integrated, visual tools to support coordination of components/services, including graphical editors, animation and simulation tools, and model checkers [1,8,15]. We expect our tool to be useful in model-based development of service-oriented applications. Our aim is to aid designers who are interested in complex coordination scenarios and to use UML SDs as the basis for generating implementations automatically using our synthesis approach. A prototype is being developed which uses a simple description of UML SDs, and in our future work, we will exploit some existing UML design tools and use their XMI exported UML models directly as input for our synthesis. Once the constraint automaton (or Reo circuit) is generated from scenario specifications, we can also apply the existing tools, for example, the model checker, to check the containment and equivalence of connectors. Furthermore, we will consider the appropriate representation of QoS aspects in UML and their connection with quantitative constraint automata and quantitative Reo circuits [4].

Acknowledgements. The work reported in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

References

1. Eclipse Coordination Tools, http://homepages.cwi.nl/~koehler/ect/
2. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Arbab, F., Baier, C., de Boer, F., Rutten, J., Sirjani, M.: Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 236–251. Springer, Heidelberg (2005)
4. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component Connectors with QoS Guarantees. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 286–304. Springer, Heidelberg (2007)
5. Arbab, F., Meng, S., Baier, C.: Synthesis of Reo Circuits from Scenario-based Specifications. In: Proceedings of FOCLASA 2008 (2008)
6. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61, 75–113 (2006)
8. Blechmann, T., Baier, C.: Checking Equivalence for Reo Networks. In: Proceedings of 4th International Workshop on Formal Aspects of Component Software, FACS 2007 (2007)
9. Carriero, N., Gelernter, D.: Coordination Languages and Their Significance. Communications of the ACM 35, 97–107 (1992)
10. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(0) (2001)
11. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. Foundations of Computer Science 13, 5–51 (2002)
12. Harel, D., Kugler, H., Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: Proc. Formal Methods in Software and Systems Modeling, pp. 309–324 (2005)

13. ITU-TS. Recommendation Z.120(11/99) : MSC 2000, Geneva (1999)
14. Khriss, I., Elkoutbi, M., Keller, R.K.: Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 132–147. Springer, Heidelberg (1999)
15. Klüppelholz, S., Baier, C.: Symbolic Model Checking for Channel-based Component Connectors. In: Canal, C., Viroli, M. (eds.) Proceedings of FOCLASA 2006, pp. 19–36 (2006)
16. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From mscs to statecharts. In: Distributed and Parallel Embedded Systems, pp. 61–72. Kluwer, Dordrecht (1999)
17. Krüger, I.H., Mathew, R.: Component Synthesis from Service Specifications. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 255–277. Springer, Heidelberg (2005)
18. Lazovik, A., Arbab, F.: Using Reo for Service Coordination. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 398–403. Springer, Heidelberg (2007)
19. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Comm. ACM 46(10), 25–28 (2003)
20. Mäkinen, E., Systä, T.: Mas - an interactive synthesizer to support behavioral modeling in uml. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, pp. 15–24. IEEE Computer Society, Los Alamitos (2001)
21. Meng, S., Arbab, F.: Web Services Choreography and Orchestration in Reo and Constraint Automata. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 346–353. Springer, Heidelberg (2007)
22. Object Management Group. Unified Modeling Language: Superstructure - version 2.1.1 (2007), http://www.uml.org/
23. Sun, J., Dong, J.S.: Design Synthesis from Interaction and State-Based Specifications. IEEE Transactions on Software Engineering 32, 349–364 (2006)
24. Tivoli, M., Autili, M.: SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors. RSTI L'object 12, 77–103 (2006)
25. Uchitel, S., Brunet, G., Chechik, M.: Behaviour model synthesis from properties and scenarios. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 34–43. IEEE Computer Society, Los Alamitos (2007)
26. Uchitel, S., Kramer, J.: A Workbench for Synthesising Behaviour Models from Scenarios. In: Proceedings of International Conference on Software Engineering (ICSE 2001), pp. 188–197. IEEE Computer Society, Los Alamitos (2001)
27. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In: Proceedings of the 9th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2001), pp. 74–82. ACM, New York (2001)
28. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems 19(2), 292–333 (1997)
29. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Revisiting Statechart Synthesis with an Algebraic Approach. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). IEEE Computer Society, Los Alamitos (2004)

State Space Reduction Techniques for Component Interfaces

Markus Lumpe, Lars Grunske, and Jean-Guy Schneider

Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, Australia
{mlumpe,lgrunske,jschneider}@swin.edu.au

Abstract. Automata-based interface and protocol specifications provide an elegant framework to capture and automatically verify the interactive behavior of component-based software systems. Unfortunately, the underlying formalisms suffer from *combinatorial state explosion* when constructing new specifications for composite components or systems and may therefore render the application of these techniques impractical for real-world applications. In this paper, we explore the *bisimulation* technique as a means for a mechanical state space reduction of component-based systems. In particular, we apply both strong and weak bisimulation to *Component Interaction Automata* in order to obtain a *minimal automata* that can serve as a behavioral equivalent abstraction for a given component specification and illustrate that the proposed approach can significantly reduce the complexity of an interface specification after composition.

1 Introduction

With the emphasis on reusable, self-contained software components that expose their capabilities through well-defined interfaces [30], there is a clear need for sound interface and protocol specifications. Such specifications allow, for example, for a precise characterization of both, the number and sequence of events and method calls, and have been well motivated for the definition, analysis, and verification of object-oriented software systems [24]. In component-based software engineering, these techniques become even more important [27, 13, 33] as they provide an effective means for the reduction of the complexity in component specifications through abstraction. The added benefit of this approach is that, ideally, component users only need to understand the interface specification of a given component in order to be able to use it correctly in a given deployment environment.

In the past decades, there has been a significant research interest in suitable and formal support for the specification and verification of component interfaces and their composition. Several approaches have emerged that focus either on service interfaces, interaction protocols, or both [6, 13, 21, 1, 22, 27, 31, 5, 7, 11, 10]. The majority of these formalisms rely on *finite state machines* as the underlying formal model and employ some form of automata-based notation to denote interface specifications. The finite

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 130–145, 2008.

state machine model allows for a fine-grained description of how and when specific service requests of components can interact with the deployment environment. This serves component developers, component testers, and component integrators alike as the entropy of a given component interface or system specification can be geared towards the compatibility requirements [8, 19] indicated for correctness, fitness, and safety checks, respectively.

Unfortunately, automata-based models suffer from *combinatorial state explosion*. That is, whenever one composes two or more component interface specifications, the result, often constructed as a product automaton [5, 11], will contain a significantly large number of states and transitions. Moreover, component compatibility verification techniques like model checking, substitutability checking, or refinement checking, and the constructions of the composed interface specification, in itself, have an exponential complexity with respect to the number of states and transition to be considered. Therefore, the application of some form of *state space reduction* becomes essential in order to permit actual component composition and component fitness checks to take place in real-world scenarios.

In this paper, we propose a state space reduction method for composite component interface specifications based on state partition refinement [25] using the *bisimulation technique* [26]. Bisimulation is a *co-inductive* proof technique to test whether two automata exhibit the same interaction behavior. When applied to component interface specifications, bisimulation allows for the identification of behavioral equivalent component states and, consequently, for the verification of "component substitutability" when constructing a *minimal component interface specification*.

Several variants of the bisimulation technique exist. Of particular interest are the *strong* and *weak* versions [18]. The composition of two or more component interface specifications can produce identical behavioral patterns in the resulting automata. We can equate these patterns by building a respective bisimulation relation. The discriminative power of the bisimulation technique allows us to define the pruning of the state space either over direct common component interaction prefixes with the environment (strong bisimulation) or the transitive closure of inter-component synchronizations (weak bisimulation). Moreover, the strong bisimulation relation is known to preserve temporal properties, as required for model checking [16, 14, 29], whereas weak bisimulation yields an observable equivalence that abstracts from the internal behavior.

We use *Component Interaction Automata* (CIA) [5, 7] as the underlying model for our state space reduction technique. Interestingly, Černá et al. [7] have already studied strong bisimulation in order to define component substitutability in the *Component Interaction Automata* framework. However, the focus of our work is on finding a redundant-free, minimal specification for a given composite system. Therefore, we seek to provide answers to the following questions:

- How can a minimal, strongly bisimilar automaton be constructed automatically?
- How can a minimal, weakly bisimilar automaton be constructed automatically?
- How efficient is the state space reduction in terms of the number of the eliminated states and transitions?
- What are the costs of state space reduction?
- When should the state space reduction be applied?

We have developed a prototype implementation in PLT-Scheme [28] to study the different composition alternatives and the feasibility of a partition refinement algorithm for *Component Interaction Automata*. This prototype allows us to configure and perform experiments with the different notions of bisimulation and to take time measurements in order to assess the effectiveness of composition under state space reduction. As a test bed for our experiments, we use both, specially-designed as well as randomly generated component interaction automata specifications. Parameters like number of actions, number of states, number of transitions, ratio of inter-component synchronizations, etc. are used to fine-tune the component interface specification generator. This approach limits the impact of human bias in selecting candidate automata and enables us to determine more effectively which reduction strategy yields the best results with respect to size and structure of the composite system being analyzed.

The rest of the paper is organized as follows: Section 2 provides some background on the interface specification formalism of *Component Interaction Automata* and presents some simple examples how this formalism can be used. In Section 3, we illustrate the main contributions of this paper: (i) the adaption of a state space reduction technique to identify bisimilar states and (ii) the construction of minimal component interaction automata. We proceed with a report on the results of a set of experiments to assess the effectiveness and applicability of the proposed techniques in Section 4. We discuss related work in Section 5 and conclude with a summary of our main observations as well as an outlook to future work in Section 6.

2 The Core of Component Interaction Automata

Component Interaction Automata [5, 7] provide a formal specification framework to denote not only the interaction behavior but also the hierarchical structure of components. This new model, which is an extension of *Interface Automata* [11] and *Team Automata* [31], offers two major innovations. First, all component interaction automata maintain a structural information to record their corresponding composition hierarchy. For one automaton the value associated with this information is just the component identifier of the component itself. However, for composite automata a tuple is used whose elements capture the composition architecture of the composed system. To illustrate this concept, consider two component interaction automata $A1, A2$ and their composition. We write $(A1)$ and $(A2)$ to capture the hierarchical structure of $A1$ and $A2$, respectively. In other words, $A1$ and $A2$ are primitive components that do not reveal any internal compositions [5]. In contrast, we denote by $((A1)(A2))$ the hierarchical structure of the composition of $A1$ and $A2$. Similarly, we express the hierarchical structure of the composition of the primitive component $A3$ with the composite of $A1$ and $A2$ as $((A3)((A1)(A2)))$.

The second novelty is the use of *structured labels*, which are triples that encode the action, originating component, and target component in the transitions of a component interaction automaton. There are three forms of structured labels: $(-, a, n)$, the *input* of a at component n, $(n, a, -)$, the *output* of a emitted from component n, and (n_1, a, n_2), the *synchronization* of components n_1 and n_2 through action a. The symbol $-$ in both

input and output stands for the *environment*. Consequently, input and output denote external interactions, whereas synchronization is an internal component interaction.

Formally, a component interaction automata is defined as follows [5]:

Definition 1. *A component interaction automaton $\mathcal{C}$ is a quintuple (Q, Act, δ, I, S) where:*

- *Q is a finite set of states,*
- *Act is a finite set of actions,*
- *$\delta \subseteq Q \times \Sigma \times Q$ is a finite set of labeled transitions with $\Sigma \subseteq \{(N \times Act \times N)\} \setminus \{(\{-\} \times Act \times \{-\})\}$, where $N = \{n \mid n \text{ occurs in } S\} \cup \{-\}$ is the set of structured labels induced by $\mathcal{C}$,*
- *$I \subseteq Q$ is a non empty set of initial states, and*
- *S is a tuple denoting $\mathcal{C}$'s hierarchical composition structure.*

To illustrate the use of the component interaction automata formalism, consider the following two small components $C1$ and $C2$[1]:

$$C1 = (\{q_0, q_1\}, \{a, b\}, \{(q_0, (1, a, -), q_1), (q_1, (-, b, 1), q_0)\}, \{q_0\}, (1))$$
$$C2 = (\{q_0, q_1\}, \{a, b\}, \{(q_0, (-, a, 2), q_1), (q_1, (2, b, -), q_0)\}, \{q_0\}, (2))$$

$C1$ has two states and is *output-enabled* in its initial state. In particular, upon activation component $C1$ emits action a and moves, after a successful delivery, into state q_1. In this state, $C1$ waits for action b and upon receiving b returns back to its initial state q_0. In contrast, $C2$, while also defining two states, is *input-enabled* in its initial state. That is, when activated, $C2$ waits for input action a. After receiving a, $C2$ moves into state q_1 and issues b. Once the output is completed, $C2$ moves back to its initial state q_0.

Two or more component interaction automata can be composed to form a new component interaction automaton. The composition, however, can be parameterized over a set of *reachable external* actions. These actions define the *provided* and *required* ports of a component. A particular feature of reachable actions is that they can also occur in component synchronizations. This allows for the specification of more collaborative interaction patterns than possible in *Interface Automata* [11] or *Team Automata* [31]. Specifically, component architects can add additional behavior to composites in order to reify their inter-component synchronizations. The corresponding rules for the selection of the reachable external actions originate from a secondary architectural description outside the *Component Interaction Automata* formalism [5]. In the following, we use R to stand for the set of reachable *required* actions and P to denote the set of reachable *provided* actions.

Definition 2. *Let $\mathcal{S}_R^P = \{(Q_i, Act_i, \delta_i, I_i, S_i)\}_{i \in \mathcal{I}}$, where $\mathcal{I} \subset \mathbb{N}$ is finite, R is the set of reachable required actions, and P is the set of reachable provided actions, be a system of pairwise disjoint component interaction automata. Then $\mathcal{C} = (\prod_{i \in \mathcal{I}} Q, \cup_{i \in \mathcal{I}} Act,$*

[1] We use only numerical values as component identifiers (i.e., $C1$ in denoted as 1).

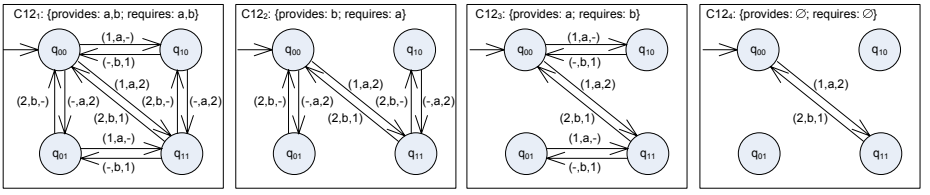


Fig. 1. Composition of $C1$ and $C2$ under varying architectural constraints

$\Delta_{OldInternal} \cup \delta_{NewInternal} \cup \delta_{Input} \cup \delta_{Output}, \prod_{i\in\mathcal{I}} I, (S_i)_{i\in\mathcal{I}})$ is a component interaction automaton restricted by R and P where:

$$\Delta_{OldInternal} = \{(q,(n_1,a,n_2),q') \mid \exists i \in \mathcal{I} : (q_i,(n_1,a,n_2),q_i') \in \delta_i \ \wedge\ \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\},$$

$$\delta_{NewInternal} = \{(q,(n_1,a,n_2),q') \mid \exists i_1,i_2 \in \mathcal{I}, i_1 \neq i_2 : (q_{i_1},(n_1,a,-),q_{i_1}') \in \delta_{i_1} \wedge (q_{i_2},(-,a,n_2),q_{i_2}') \in \delta_{i_2} \ \wedge\ \forall j \in \mathcal{I}, i_1 \neq j \neq i_2 : q_j = q_j'\},$$

$$\delta_{Input} = \{(q,(-,a,n),q') \mid a \in R \ \wedge\ \exists i \in \mathcal{I} : (q_i,(-,a,n),q_i') \in \delta_i \ \wedge\ \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\},$$

$$\delta_{Output} = \{(q,(n,a,-),q') \mid a \in P \ \wedge\ \exists i \in \mathcal{I} : (q_i,(n,a,-),q_i') \in \delta_i \ \wedge\ \forall j \in \mathcal{I}, j \neq i : q_j = q_j'\}.$$

Consider again the components $C1$ and $C2$. We write $\{C1, C2\}_R^P$ to denote their composition with respect to the provided actions P and the required actions R. Four possible outcomes[2] of their composition are shown in Figure 1. Common to all is that each composite interaction automaton contains two synchronizations: $(1, a, 2)$ and $(2, b, 1)$. These internal synchronizations reflect the established handshake protocol between $C1$ and $C2$. The remaining transitions are a direct result of the specified architectural constraints. In $C12_1$, the actions of $C1$ and $C2$ are declared as reachable and, therefore, occur in the final component interaction automata. In $C12_2$ and $C12_3$ either the actions of $C2$ or the actions of $C1$ are declared reachable, hence the reduced set of final transitions. Finally, $C12_4$ only contains internal synchronizations, as both sets, P and R, are empty. Moreover, due to the architectural constraints, the states q_{01} and q_{10} become isolated in $C12_4$. However, the *Component Interaction Automata* formalism does not provide any provisions for an explicit removal of isolated states and consequently such states remain in the corresponding composite automaton.

Unlike interface automata, where the set of actions of the composed components has to be pairwise disjoint [11], the structured labels of component interaction automata allow for the composition of components with common actions. Consider, for example, the two structurally equivalent buffer components $B1$ and $B2$ as shown in Figure 2. Both automata share the same set of common actions *set* and *get*. However, within structured labels, we can differentiate between these common actions by inspecting their corresponding originating and target component and, as a result, we can safely compose $B1$ and $B2$ to form $B12$.

[2] We omit the actual formal specification of the composition in favor of readability.

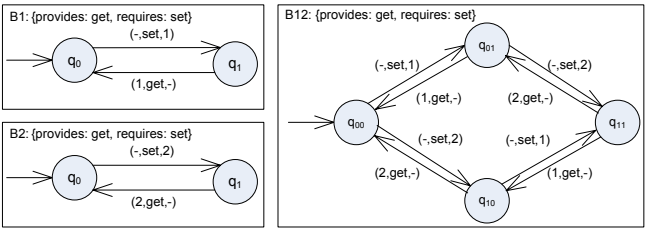


Fig. 2. The composition of structurally equivalent buffers

Unfortunately, composite automata such as $B12$ do not offer any possibility for the reduction of their complexity, as each state is unique and all transitions, even those with the same action, are required. This is an unavoidable consequence of the nature of component interaction automata composition. However, during the modeling process, component architects may sometimes wish to abstract from the internal hierarchical composition structure of a component in order to obtain a more coarse-grained specification. The result of such an abstraction is called *primitive component* and can be constructed using the following definition:

Definition 3. *Let $\mathcal{C} = (Q, Act, \delta, I, S)$ be a component interaction automata and n be a fresh component identifier. Then $\mathcal{C}' = (Q, Act, \delta', I, (n))$ is the primitive image of $\mathcal{C}$ with $\delta' = \delta'_{internal} \cup \delta'_{input} \cup \delta'_{output}$ where*

$$\delta'_{internal} = \{(q, (n, a, n), p) \mid (q, (n_1, a, n_2), p) \in \delta\},$$
$$\delta'_{input} = \{(q, (-, a, n), p) \mid (q, (-, a, n), p) \in \delta\},$$
$$\delta'_{output} = \{(q, (n, a, -), p) \mid (q, (n, a, -), p) \in \delta\}.$$

While abstraction necessarily results in a loss of information and may also produce a non-deterministic automaton (e.g., the outgoing transitions labeled with action set of state q_0 in $B12$ become indistinguishable), it provides us with a first option for state space reduction. Consider the composite component interaction automata $B12_1$ and $B123_1$ shown in Figure 3. Both automata, which are primitive, contain equivalent substructures. In $B12_1$ the states q_{01} and q_{10} are equivalent, whereas in $B123_1$ the states $q_{001}, q_{010}, q_{100}$ and $q_{011}, q_{101}, q_{110}$ are pairwise equivalent. We can use partition refinement and *strong bisimulation* to prune the state space of $B12_1$ and $B123_1$ and obtain the new minimal automata $B12_2$ and $B123_2$. The new automata are behavioral equivalent to their original automata. Moreover, both are now deterministic. Unfortunately, partition refinement cannot guarantee to always yield a deterministic automaton when applied to a non-deterministic one.

3 Minimal Component Interaction Automata Construction

We face a *state explosion problem* when composing two or more component interaction automata. An early indication of how fast the number of states and transitions grow in a composite automaton is already evident by revisiting the two component interaction

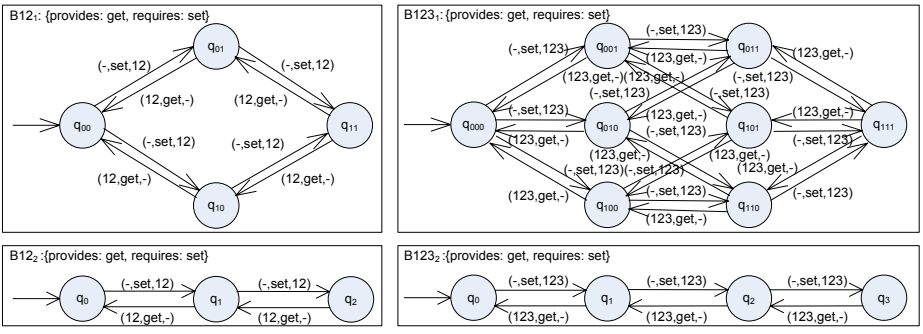


Fig. 3. The primitive CIA $B12_1$, $B123_1$ and their strongly bisimilar variants $B12_2$, $B123_2$

automata $B12$ and $B123$ shown in Figure 3. In fact, the number of states increases with 2^n, where n is the number of components being composed (cf. Table 1). However, the growth rate of transitions exceeds by far the growth rate of states. This is typical for the composition of structurally equivalent components. For this reason, we must pay special attention to the order in which we perform the analysis when defining a tool-based approach for state space reduction of composite component interaction automata.

Table 1. Growth Analysis of the Composition of Structurally Equivalent Buffers

# of Buffers Composed	2	3	4	5	6	7	8	9	10	11	12
# of States before Reduction	4	8	16	32	64	128	256	512	1024	2048	4096
# of States after Reduction	3	4	5	6	7	8	9	10	11	12	13
# of Transitions before Reduction	8	24	64	160	384	896	2048	4608	10240	22528	49152
# of Transitions after Reduction	4	6	8	10	12	14	16	18	20	22	24

The example of the nth composition of structurally equivalent buffers is a very special case that, nevertheless, vividly illustrates the possible gains when using some form of state space reduction technique. In particular, when using *strong bisimulation* to refine the state partitions of the composite buffers, the resulting composite automata grow only linearly. That is, the number of states of the nth composite automaton is $n + 1$, whereas the number of transitions is $2n$. As a consequence, the composition of a reduced composite buffer with another simple buffer results only in a linear growth.

Strong bisimulation provides the means for the definition of a *fine-grained* equivalence relation over component interaction automata. A specific feature of the *Component Interaction Automata* formalism is, however, that any equivalence relations must respect the hierarchical composition structure of the component in question. Recall the definition of Σ, the alphabet of structured labels for a given automaton $\mathcal{C}$. Σ is defined over both the set of actions, Act, and the hierarchical composition structure, S. As a consequence, two component interaction automata C and C' can only be considered equivalent, if and only if they exhibit the same underlying composition structure. In order words, when defining a minimal component interaction automaton, we can

reduce the original automaton's state space complexity but have to retain its underlying hierarchical make-up.

Definition 4. *Let $C = (Q, Act, \delta, I, S)$ and $C' = (Q', Act, \delta', I', S)$ be two component interaction automata. A binary relation $\sim \subseteq Q \times Q'$ is a strong bisimulation if it is symmetric and $q \sim p$ implies whenever*

- $(q, (n_1, a, n_2), r) \in \delta$, *then* $\exists s \in Q'$ *such that* $(p, (n_1, a, n_2), s) \in \delta'$ *and* $r \sim s$,
- $(q, (-, a, n), r) \in \delta$, *then* $\exists s \in Q'$ *such that* $(p, (-, a, n), s) \in \delta'$ *and* $p \sim s$,
- $(q, (n, a, -), r) \in \delta$, *then* $\exists s \in Q'$ *such that* $(r, (n, a, -), s) \in \delta'$ *and* $r \sim s$.

Two component interaction automata C and C' are strongly bisimilar, written $C \sim C'$, if they are related by some strong bisimulation.

To illustrate the application of strong bisimulation for partition refinement, consider again the two composite automata $B12_1$ and $B123_1$. We can compute the partition refinement over strong bisimulation and obtain the following new partitions:

$$B12_1 : \{\{q_{00}\}, \{q_{01}, q_{10}\}, \{q_{11}\}\}$$
$$B123_1 : \{\{q_{000}\}, \{q_{001}, q_{010}, q_{100}\}, \{q_{011}, q_{101}, q_{110}\}, \{q_{111}\}\}$$

That is, from the perspective of an external observer, the states q_{01}, q_{10} in $B12_1$ and the states $q_{001}, q_{010}, q_{100}$ and $q_{011}, q_{101}, q_{110}$ in $B123_1$ are indistinguishable. We can, therefore, construct new automata in which theses states are merged into one:

$$B12_2 = (\{q_0, q_1, q_2\}, \{set, get\}, \delta_{B12_2}, \{q_0\}, ((1)(2)))$$
$$B123_2 = (\{q_0, q_1, q_2, q_3\}, \{set, get\}, \delta_{B123_2}, \{q_0\}, ((1)(2)(3)))$$

where

$$\delta_{B12_2} = \{(q_0, (-, set, 12), q_1), (q_1, (-, set, 12), q_2), (q_2, (12, get, -), q_1), (q_1, (12, get, -), q_0)\}$$
$$\delta_{B123_2} = \{(q_0, (-, set, 123), q_1), (q_1, (-, set, 123), q_2), (q_2, (-, set, 123), q_3), (q_3, (123, get, -), q_2), (q_2, (123, get, -), q_1), (q_1, (123, get, -), q_0)\}.$$

Strong bisimulation does not distinguish between internal synchronizations and reachable external actions. This is the discriminating power of *weak bisimulation*, which allows for the formulation of an observable equivalence relation between component interaction automata. For example, the composite automata $C12_1$, $C12_2$, $C12_3$, and $C12_4$, as shown in Figure 1, can be simplified by merging the states q_{00} and q_{11}. In each instance, these states are indistinguishable under weak bisimulation and partition refinement collapses them to one state.

Weak bisimulation provides an abstraction over internal inter-component synchronizations. A crucial ingredient in the definition of weak bisimulation is the *transitive closure* of inter-component synchronizations in a given component interaction automaton $\mathcal{C}$. The elements of the transitive closure are *synchronization paths*.

Definition 5. *Let $C = (Q, Act, \delta, I, S)$ be a component interaction automaton. A binary relation $\Rightarrow \subseteq Q \times Q$ over C implies whenever $(q, q') \in \Rightarrow$, then there exists a finite path from q to q' of $k \geq 1$ synchronization transitions such that*

$$\{(q, (n_1, a_1, n_1')r_1), (r_1, (n_2, a_2, n_2')r_2), ..., (r_{k-1}, (n_k, a_k, n_k'), q')\}.$$

Furthermore, we write $\stackrel{*}{\Rightarrow}$ to denote the reflexive transitive closure of $\Rightarrow$.

Definition 6. *Let $C = (Q, Act, \delta, I, S)$ and $C' = (Q', Act, \delta', I', S)$ be two component interaction automata. A binary relation $\approx \subseteq Q \times Q'$ is a weak bisimulation if it is symmetric and $q \approx p$ implies whenever*

- *$(q, (n_1, a, n_2), r) \in \delta$, then $\exists u, u', s \in Q'$ and $(u, (n_1, a, n_2), u') \in \delta'$ such that $p \stackrel{*}{\Rightarrow} u$, $u' \stackrel{*}{\Rightarrow} s$, and $r \approx s$,*
- *$(q, (-, a, n), r) \in \delta$, then $\exists u, u', s \in Q'$ and $(u, (-, a, n), u') \in \delta'$ such that $p \stackrel{*}{\Rightarrow} u$, $u' \stackrel{*}{\Rightarrow} s$, and $r \approx s$,*
- *$(q, (n, a, -), r) \in \delta$, then $\exists u, u', s \in Q'$ and $(u, (n, a, -), u') \in \delta'$ such that $p \stackrel{*}{\Rightarrow} u$, $u' \stackrel{*}{\Rightarrow} s$, and $r \approx s$.*

Two component interaction automata C and C' are weakly bisimilar, written $C \approx C'$, if they are related by some weak bisimulation.

We can use weak bisimulation to simplify the composite component interaction automata $C12_1$, $C12_2$, $C12_3$, and $C12_4$. Partition refinement yields

$$C12_{1-4} : \{\{q_{00}, q_{11}\}, \{q_{01}\}, \{q_{10}\}\}$$

That is, the states q_{00} and q_{11} are weakly bisimilar and appear indistinguishable to an external observer. We use this information to construct new automata (we only show $C12_3$):

$$C12_3 = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta_{C12_3}, \{q_0\}, ((1)(2)))$$

where

$$\begin{aligned}\delta_{C12_2} = \{&(q_0, (1, a, -), q_1), (q_0, (1, a, 2), q_3), (q_1, (-, b, 1), q_0),\\ &(q_3, (2, b, 1), q_0), (q_2, (1, a, -), q_3), (q_3, (-, b, 1), q_2)\}.\end{aligned}$$

We have implemented the bisimulation-based partition refinement for component interaction automata in PLT-Scheme [28]. Our prototype consists of the two parts *composition* and *minimization*. Both processes can be configured over a variety of parameters that allow us, for example, to perform additional sanity checks to verify the soundness of specifications or to provide more fine-grained details about both the composition and the minimization process.

Internally, our implementation maintains dictionaries that provide maps, either from states to transitions or actions to transitions, in order to speed up the lookup process for transitions. As mentioned before, the number of transitions grows much faster than the number of states when composing component interaction automata. By using the dictionaries, which are created only once per run, we can process a greater class of

specifications, even though we still face exponential time complexity in the computation of component interaction automata composition.

At the heart of the minimization process is a partition refinement algorithm [17, 15, 25] that takes a *splitter function* as argument. At present, we have defined two splitter functions: one for strong bisimulation relation and one for weak bisimulation. The partition refinement tries to merge equivalent states. If this fails (e.g., the given automata cannot be minimized), the minimization process just returns the result of the composition. Otherwise, we construct a new, reduced automaton in which all occurrences of duplicate transitions are deleted.

Both, composition and minimization are timed to allow for a performance analysis. We record the actual processing time and the time spent in the garbage collector. We are primarily interested in the actual computation time, but the frequency of garbage collector invocations provides us with valuable information that assists us not only in asserting certain quantitative properties, but also in improving the quality of our partition refinement algorithm in the future.

4 Validation and Results

The *Component Interaction Automata* formalism provides a powerful means to capture and analyze the interaction behavior and hierarchical structure of component-based systems. Unfortunately, the inherent combinatorial complexity in terms of space and time makes it difficult to apply this formalism in real-world scenarios. Hence, a suitable state space reduction technique is needed to enable the effective composition of a large number of components. To further illustrate this fact, consider the scenario where a number of structurally equivalent buffers (cf. Section 2) are composed, with and without state space reduction of the resulting intermediate component interaction automata. As shown in Figure 4, in both cases composition always exhibits the expected exponential time complexity. However, the rate of growth is significantly smaller if intermediate component interaction automata are reduced. The time it takes to compose 25 structurally equivalent buffers, whose composites are simultaneously optimized through partition refinement, is still below the time required to compose 10 buffers without state space reduction.

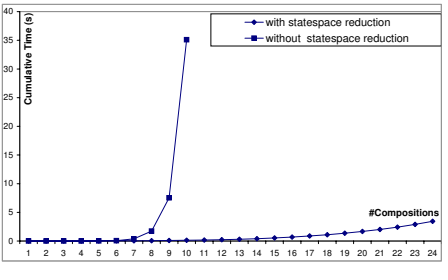


Fig. 4. Timed buffer composition

Nevertheless, the application of the proposed partition refinement technique for *Component Interaction Automata* raises two major questions: (i) when do we need to apply state space reduction and (ii) how efficient is state space reduction via bisimulation? In order to answer these questions we ran a series of experiments and applied our approach to both, specially-designed as well as randomly generated component interaction automata specifications. We configured the automata generation process in a way so that the obtained component interaction automata specifications resulted in

deterministic and coherent systems. In particular, all generated systems contained a variable number of states, a subset of predefined action labels, and used these action labels either in input or output transitions to allow for internal synchronizations between the components subject to composition. As testbed for our experiments we used a Windows-based PC equipped with a 2.2 GHz dual-core processor and 2GB of main memory.

When should we apply state space reduction via bisimulation? In order answer this question, we investigated the effectiveness of state space reduction in varying composition scenarios using both, strong and weak bisimulation. As shown in Figure 4, there is a clear benefit in applying state space reduction to the composition of structurally equivalent components. Furthermore, due to the absence of any internal synchronizations, strong and weak bisimulation yield the same results.

But would we obtain the same benefits if we were to apply state space reduction to the composition of structurally different components? For this purpose, we generated 25 component interaction automata, each having 4 states and in-between 9 and 12 transitions. Based on these 25 specifications, we then constructed composites of up to 10 components that were again computed with and without state space reduction. The best and worst cases composition scenarios are illustrated in Table 2. For each composition we list the number of transitions as well as the corresponding composition time. For state space reduction with weak bisimulation we also record the reduction time.

Table 2. Best and worst case scenarios for composition under weak bisimulation

		without bisimulation		**with weak bisimulation**		
	Series	# transitions	comp. time	# transitions	comp. time	reduction time
Best Case	A1	169	0.016s	169	0.016s	0.032s
	A2	1,152	0.078s	211	0.016s	0.016s
	A3	6,496	0.978s	120	0.016s	0.001s
	A4	34,368	21.46s	131	0.015s	0.016s
	A5	179,456	564.69s	131	0.015s	0.016s
	A6	887,808	$>$ 6h	146	0.016s	0.001s
Worst Case	B1	156	0.015s	156	0.015s	4.10s
	B2	1,000	0.057s	1,000	0.047s	3,084.76s
	B3	6,208	1s	5,914	0.890s	46.01s
	B4	34,048	21.24s	130	0.016s	0.001s
	B5	175,104	534.65s	132	0.016s	0.001s
	B6	863,232	$>$ 6h	137	0.016s	0.001s

Our experiments revealed that the composition of 6 components without state space reduction took, on average, more than 6 hours to compute. Compositions of a higher degree required a computation time of more than 24 hours. On the other hand, using state space reduction via weak bisimulation, we were able to determine the respective compositions in substantially less time for all test scenarios. The maximum time required to compute the composition of 10 components was approx. 51 minutes.

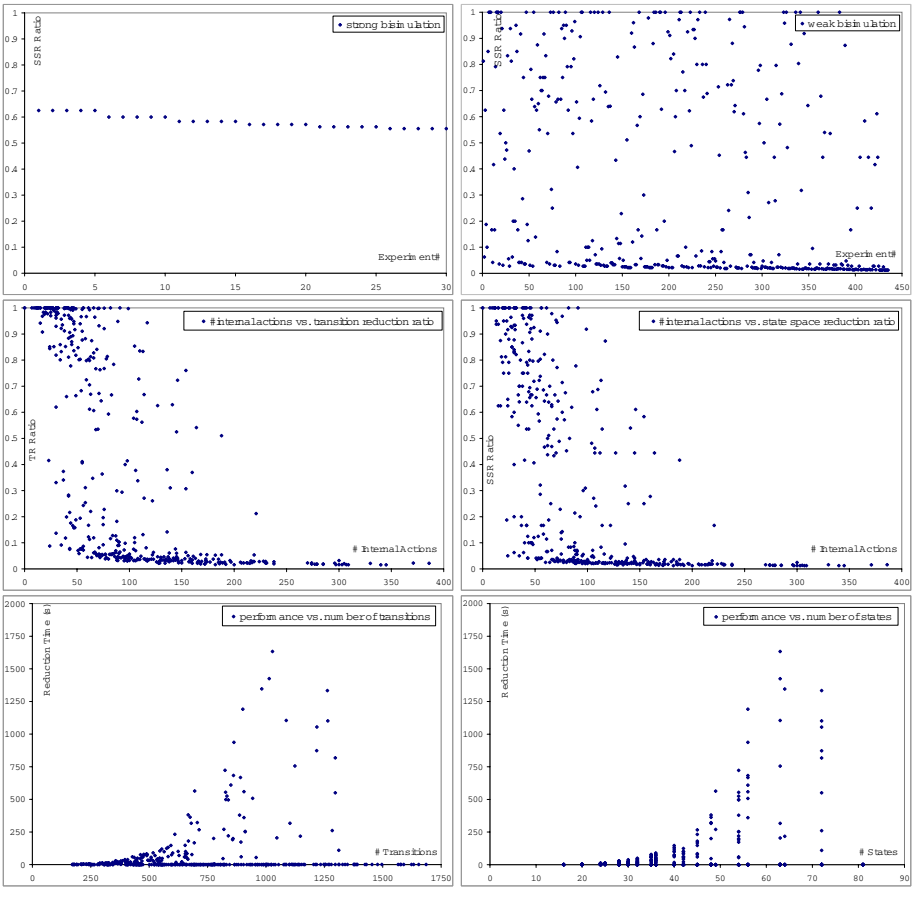


Fig. 5. Performance Characteristics for the State Space Reduction Technique

However, the efficiency of state space reduction varied significantly. In some cases, all composite interaction automata could be reduced, resulting in significantly smaller interaction automata for further compositions. In other cases, no or only a minor reduction was achieved and the remaining compositions as well as the partition refinement needed to be performed on a much larger component interaction automaton. But once a composite automata reached a certain threshold with respect to the number of internal synchronizations, weak bisimulation was able to collapse the state space substantially.

How efficient is state space reduction via bisimulation? In order to find an answer to this question, we altered our experiment setup to allow for specific types and densities of transitions occurring in the component interaction automata being composed. In particular, we sought to achieve a probability of 0.25 for the occurrences of internal synchronizations and a probability of 0.8 that two states were connected by a transition for 30 components with 3 to 10 states each.

Based on these settings, we obtained 435 composite component interaction automata specifications originating from the composition of two different automata and 30 composites in which two identical specifications were combined. For each of these 465 composite automata we applied both, strong and weak bisimulation to reduce the corresponding state space. Using strong bisimulation, only the 30 composites constructed from identical components could be reduced. For the remaining composites no state space reduction was possible. This is not surprising as strong bisimulation can only eliminate *identical substructures* and it is unlikely that such identical substructures exist when two arbitrary component interaction automata are composed. The results of the state space reduction using strong bisimulation are illustrated in Figure 5, top left chart. Please note that the reduction ratio depends on the number of states of the input automata. The type and number of transitions have no effect on the reduction ratio when composing two structurally equivalent component interaction automata.

The effects of the state space reduction using weak bisimulation are depicted in Figure 5, top right chart. The resulting state space reduction ratio ρ varies from case to case. Some composites could not be reduced ($\rho = 1$) whereas others shrunk significantly ($\rho < 0.05$). The distinguishing factor between these two cases is the number of inter-component synchronizations occurring between the composed components. If the composed components cannot synchronize, then the number of internal actions is zero and as a result, only state space reduction via strong bisimulation is possible. On the other hand, if composites contain internal transitions, then these internal transitions improve the likelihood to yield weakly bisimilar states. The relationship between the number of internal transitions in the composed automata and the state and transition reduction ratio is shown in the two middle charts of Figure 5. These charts confirm the direct relationship between reduction ratios and the number of internal transitions.

An analysis of the performance of the implemented partition refinement for component interaction automata is presented in the bottom two charts of Figure 5. It shows that the space reduction time depends on both, the number of states and the number of transitions. A small proportion of our results indicate an exponential worst case complexity, whereas a majority clearly performs better.

5 Related Work

The work presented in this paper builds upon previous research on bisimulation equivalences in the area of process algebras, where they have been studied extensively from many perspectives. For example, bisimulation relations are used to check equivalence between processes [23] or to implement efficient model-checking algorithms that verify temporal logical formulas [9, 29]. Recently, the application of bisimulation has been extended to include real-time [2] and probabilistic/stochastic algebras [3, 18].

However, in the area of protocol and interface specifications for component-based systems, the concept of bisimulation is rarely used. Černá et al. [7] define *Component Interaction Automata* as an interface specification formalism and use bisimulation to identify whether components are equivalent and as such can be safely substituted. Our work is inspired by this approach, but rather than using bisimulation as a means to check for substitutability, we aim at the identification of a minimal component

interaction automaton that requires a reduced set of states to facilitate further composition, refinement, and model-checking, respectively.

Component Interaction Automata follow the theory of interface-based design [6, 13, 21] in that they allow independent component compatibility checking and implementation based on interface information alone. Similar formalisms are *Interface Automata* [11] and *I/O Automata* [22]. However, these formalisms require that the components being composed have pairwise disjoint sets of input and output actions. This restricts the practicability of both formalisms as structurally identical components cannot be composed. As a result, compositions do not contain any strongly bisimilar states and, consequently, only state space reduction via weak bisimulation is effective.

De Alfaro et al. [10] have extended *Interface Automata* to support *one-to-many* and *many-to-one* communications. The resulting formalism is called *Sociable Interfaces*. In this formalism, action labels are not distinguished in either input or output actions and, therefore, internal labels can be attached simultaneously to input and output transitions.

Another formalism that allows for multiple synchronizations is *Team Automata* [31] in which the transition set can be chosen when automata are being composed. Given this approach, composition becomes more powerful than in the previously mentioned interface formalisms. Although an adaptation of the proposed state space reduction techniques via partition refinement and bisimulation is possible for all these automata-based models, the inherent strength and the clarity of specification makes *Component Interaction Automata* a more suitable formalism for our work.

6 Conclusions and Future Work

In this paper we have presented an approach to address the problem of combinatorial state explosion that occurs when large *Component Interaction Automata* are composed. The proposed solution aims at identifying minimal strongly or weakly bisimilar specifications that can substitute the resulting composite interface specifications. To identify these minimal interface specifications, we developed a partition refinement strategy based on state space reduction and studied its effectiveness with respect to effort/gain ratios.

Based on our analysis, a conclusive answer to the question when to use the proposed state space reduction techniques and how effective it is expected to be could not be clearly identified. However, two strategies should be considered: (i) state space reduction via strong bisimulation when the composites contain *similar substructures*, and (ii) state space reduction via weak bisimulation when the resulting component interaction automaton contains a high number of *internal synchronization*.

In future work we aim to investigate on-the-fly state space reduction techniques as proposed by Fernandez [15] that already check for reachable and bisimilar states during the composition process. We expect that an adaptation of these techniques will significantly improve the run-time characteristics of our approach for complex compositions. Similarly, using distributed algorithms for the partition refinement as described by Blom and Orzan [4] could result in an additional speed-up.

Another direction of future research is to extend the *Component Interaction Automata* formalism with real-time aspects such as clock, clock invariants, and real-time

constraints. This extension could follow the ideas of *Timed Interface Automata* [12]. The work presented in this paper could then be refined to include support for state space reduction of real-time component interaction automata, similar to the work on state space reduction for timed automata [20]. Finally, as an alternative to the weak bisimulation, an implementation of state space reduction via branching bisimulation [32] could prove valuable because branching bisimulation is known to preserve selected temporal logic properties [14].

References

1. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Baeten, J.C.M., Bergstra, J.A.: Real time process algebra. Formal Aspects of Computing 3(2), 142–188 (1991)
3. Baier, C., Hermanns, H.: Weak bisimulation for fully probabilistic processes. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 119–130. Springer, Heidelberg (1997)
4. Blom, S., Orzan, S.: Distributed state space minimization. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 280–291 (2005)
5. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. SIGSOFT Software Engineering Notes 31(2), 1–8 (2006)
6. Broy, M.: A core theory of interfaces and architecture and its impact on object orientation. In: Reussner, R.H., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 26–47. Springer, Heidelberg (2006)
7. Černá, I., Vařeková, P., Zimmerova, B.: Component Substitutability via Equivalencies of Component-Interaction Automata. Electronic Notes in Theoretical Computer Science 182, 39–55 (2007)
8. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Jurdzinski, M., Mang, F.Y.C.: Interface compatibility checking for software modules. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 428–441. Springer, Heidelberg (2002)
9. Cheung, S.C., Kramer, J.: Checking safety properties using compositional reachability analysis. ACM Transactions on Software Engineering and Methodology 8(1), 49–78 (1999)
10. de Alfaro, L., da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
11. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Gruhn, V., Tjoa, A.M. (eds.) Proceedings ESEC/FSE 2001, Vienna, Austria, September 2001, pp. 109–120. ACM Press, New York (2001)
12. de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Timed Interfaces. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 108–122. Springer, Heidelberg (2002)
13. de Alfaro, L., Stoelinga, M.: Interfaces: A game-theoretic framework for reasoning about component-based systems. Electronic Notes in Theoretical Computer Science 97, 3–23 (2004)
14. DeNicola, R., Vaandrager, F.: Three logics for branching bisimulation. Journal of the ACM 42(2), 458–487 (1995)
15. Fernandez, J.-C.: An implementation of an efficient algorithm for bisimulation equivalence. Science of Computer Programming 13(2–3), 219–236 (1990)

16. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. Formal Methods in System Design 21(1), 39–78 (2002)
17. Habib, M., Paul, C., Viennot, L.: Partition refinement techniques: An interesting algorithmic tool kit. International Journal of Foundations of Computer Science 10(2), 147–170 (1999)
18. Hermanns, H.: Interactive Markov Chains. LNCS, vol. 2428. Springer, Heidelberg (2002)
19. Inverardi, P., Wolf, A.L., Yankelevich, D.: Static checking of system behaviors using derived component assumptions. ACM Transactions on Software Engineering Methodology 9(3), 239–272 (2000)
20. Kang, I., Lee, I., Kim, Y.-S.: An efficient state space generation for the analysis of real-time systems. IEEE Transactions on Software Engineering 26(5), 453–477 (2000)
21. Lee, E.A., Xiong, Y.: System-Level Types for Component-Based Design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 237–253. Springer, Heidelberg (2001)
22. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 1987, pp. 137–151 (1987)
23. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
24. Nierstrasz, O.: Regular Types for Active Objects. In: Proceedings OOPSLA 1993, September 1993. ACM SIGPLAN Notices, vol. 28, pp. 1–15 (1993)
25. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
26. Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
27. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Transactions on Software Engineering 28(11), 1056–1076 (2002)
28. PLT Scheme v372 (2008), http://www.plt-scheme.org
29. Stirling, C.: Modal and temporal logics for processes. In: Moller, F., Birtwistle, G.M. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 149–237. Springer, Heidelberg (1996)
30. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley / ACM Press (2002)
31. ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in team automata for groupware systems. Computer Supported Cooperative Work 12(1), 21–69 (2003)
32. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
33. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems 19(2), 292–333 (1997)

Model Checking of Control-User Component-Based Parametrised Systems

Pavlína Vařeková* and Ivana Černá**

Faculty of Informatics, Masaryk University
Czech Republic

Abstract. Many real component-based systems, so called Control-User systems, are composed of a stable part (*control component*) and a number of dynamic components of the same type (*user components*). Models of these systems are parametrised by the number of user components and thus potentially infinite. Model checking techniques can be used to verify only specific instances of the systems. This paper presents an algorithmic technique for verification of safety interaction properties of Control-User systems. The core of our verification method is a computation of a *cutoff*. If the system is proved to be correct for every number of user components lower than the cutoff then it is correct for any number of users. We present an on-the-fly model checking algorithm which integrates computation of a cutoff with the verification itself. Symmetry reduction can be applied during the verification to tackle the state explosion of the model. Applying the algorithm we verify models of several previously published component-based systems.

1 Introduction

Model-checking [11] is a formal verification technique which has received a wide attention in industry as it can be used to detect design errors early in the design life-cycle. Model checking is based on state space generation and as such can be directly applied to finite state systems. In case of infinite state systems more involved techniques have to be employed.

Component-based software development is an alternative to existing software development techniques. Component-based development proposes to assemble software systems from reusable components, which helps to significantly reduce development time and costs. On the other hand, interaction among components opens new issues relevant to the correctness of interaction.

An extensive study of component-based systems reveals that many systems are composed of a beforehand unknown number of components. A typical situation is the composition of one fixed component (control component) with an unknown number of identical components (user components). These systems are usually called Control-User systems. Formal verification of Control-User systems

* The author has been supported by the grant No. 1ET400300504.
** The author has been supported by the grant No. 1ET408050503.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 146–162, 2008.

includes verification of every possible composition of the control component with any number of user components. Even if the composition of the control component with a specific number of user components is finite, the verification task itself is infinite.

It has been observed [14] that reachability properties are the most common properties arising in verification. Reachability is closely related to safety, expressing that no *unsafe* state is reachable in a system. This remains true also for Control-User systems.

In this paper we aim to use existing approaches to build-up two reachability verification algorithms. Our verification method is based on *cutoffs* [15,19]. If a system is proved to be correct for every number of user components lower than the cutoff then it is correct for any number of users. First of the two introduced algorithms is suitable for efficient verification of a finite set of reachability properties. The algorithm iteratively computes the minimal cutoff and during the computation it finds a finite set of representatives of all reachable states. Representatives bear all information needed for deciding reachability and thus to verify given properties of the whole Control-User system it is enough to check these representatives. Advantages of the verification algorithm are that it works with the minimal cutoff and allows to verify many properties at the same time. Evaluation of the algorithm reveals that the cutoff is typically rather small and thus the verification itself is efficient. The experimental studies were conducted on several previously published case studies as well as on a tailored Control-User system.

The second (bounding) algorithm is proposed for computing the highest possible number of users which are simultaneously in the same state (part of a computation). The bounding algorithm provides answers to questions like *What is the maximal number of users simultaneously requiring the same services?* The number of users is called the *bound*. The algorithm has two main parts. In the first part the algorithm finds a candidate for the bound and proves that the bound is less or equal to the candidate. In the second part it computes the exact value of the bound. Experiments demonstrate that typically in the first part the algorithm efficiently finds a candidate which is equal to the bound. In both algorithms the effectiveness can be supported by symmetry reduction over the reachable state space.

The paper is structured as follows. Section 2 presents a model of Control-User system while Section 3 introduces several types of reachability properties. Section 4 highlights backward reachability for C-U systems. The algorithm for verification of reachability properties is described in Section 5 and its evaluation is in Section 6. The bounding algorithm and its evaluation can be found in Section 7. Related work is summarised in Section 8.

2 The Control-User System Model

We consider a class of parametrised systems where a system consists of a unique component - in the literature called the control component [21,17] and an

arbitrary number of components with an identical model - user components. An example of such a system with n users is in Figure 1 a). Components are executing concurrently with the interleaving semantics, capturing that a component can communicate with another component using the pairwise rendezvous synchronisation (a component can send a message iff the receiver is enabled).

As the formal model of a Control-User system we use a labelled Kripke structure (LKS) [9]. An LKS is a structure underlying many other formalisms capturing interactions between components like I/O automata [25], Component-Interaction Automata [35], or Extended Behavior Protocols [24].

Definition 1 (LKS). *A* labelled Kripke structure *(or* LKS *for short) is a tuple* $(Q, I, Ap, L, \Sigma, \delta)$ *where* Q *is a set of* states, $I \subseteq Q$ *is a set of* initial states, Ap *is a set of* atomic propositions, $L : Q \to 2^{Ap}$ *is a* state-labelling function, Σ *is a finite set of* actions *and* $\delta \subseteq Q \times \Sigma \times Q$ *is a* transition relation.

We suppose that $\Sigma = \Sigma_{int} \cup \Sigma_{out} \cup \Sigma_{inp}$, where $\Sigma_{out} = \Sigma'_{out} \times \{!\}$, $\Sigma_{inp} = \Sigma'_{inp} \times \{?\}$. The alphabets Σ_{out} resp. Σ_{inp} represent output resp. input actions which can be used for pairwise rendezvous communication between LKSs. The alphabet Σ_{int} represents internal actions. We write $q \to q'$ if there is a label $l \in \Sigma$ such that $(q, l, q') \in \delta$, $\to^*$ is the transitive and reflexive closure of $\to$. A state q is reachable iff $in \to^* q$ for some $in \in I$. Let $q = (q_0, \cdots, q_n)$ be an $n+1$-tuple. Then $pr_i(q)$, $i = 0, \ldots, n$, denotes its i+1-th projection, $pr_i(q) = q_i$.

In this paper we restrict ourselves to systems in which the models of the control and user component are finite and have one initial state. Thus LKSs $C = (Q_C, \{in_C\}, Ap_C, L_C, \Sigma_C, \delta_C)$, $U = (Q_U, \{in_U\}, Ap_U, L_U, \Sigma_U, \delta_U)$ form a *Control-User model* (or *C-U model* for short) iff Q_C, Q_U, Σ_C, and Σ_U are finite.

Definition 2 (Composition). *Let* $n \in \mathbb{N}_0$ *and for each* $i = 0, \ldots, n$ *let* $K_i = (Q_i, I_i, Ap_i, L_i, \Sigma_i, \delta_i)$ *be an LKS. Then* $K_0 \| \ldots \| K_n$ *denotes the asynchronous composition of* $K_0, \ldots, K_n$ *and is defined as the LKS*

$$(Q_0 \times \cdots \times Q_n,\ I_0 \times \cdots \times I_n,\ (Ap_0 \times \{0\}) \cup \cdots \cup (Ap_n \times \{n\}),\ L,\ \Sigma,\ \delta),$$

where the state-labelling function L *assigns to each state* $(q_0, \ldots, q_n)$ *the set* $\bigcup_{i=0}^{n} L_i(q_i) \times \{i\}$. *The alphabet* $\Sigma = \bigcup_{i=0}^{n} \Sigma_{i,int} \cup \{l \mid l \in \bigcup_{i=0}^{n} \Sigma'_{i,inp} \wedge l \in \bigcup_{i=0}^{n} \Sigma'_{i,out}\}$. *The transition relation* δ *is defined by the prescription* $(q, l, q') \in \delta$ *iff any of the next possibilities holds*

- $\exists 0 \le i \le n : \forall 0 \le j \le n, j \ne i : pr_j(q) = pr_j(q')$, *and* $(pr_i(q), l, pr_i(q')) \in \delta_i$,
- $\exists 0 \le i, i' \le n, i \ne i' : \forall 0 \le j \le n, i \ne j \ne i' : pr_j(q) = pr_j(q')$, $(pr_i(q), (l, ?), pr_i(q')) \in \delta_i$, *and* $(pr_{i'}(q), (l, !), pr_{i'}(q')) \in \delta_{i'}$.

A C-U system with n clients is modelled as the composition of $n+1$ LKSs where the first LKS stands for the control component while the others are identical and represent the users. A Control-User system with arbitrary many clients is modelled as the union of LKSs modelling systems with n clients, for all $n \in \mathbb{N}$.

Definition 3 ($\mathbf{C}\|\mathbf{U}^n$, $\mathbf{C}\|\mathbf{U}^\infty$). *Let* C, U *be a C-U model,* $n \in \mathbb{N}$. *Then* $C\|U^n$ *denotes the composition* $C\|U\|\cdots\|U$ *of* C *and* n *copies of* U. $C\|U^\infty$ *is an infinite state LKS defined* $C\|U^\infty =$

$$(\bigcup_{n\in\mathbb{N}} Q_{C\|U^n},\ \bigcup_{n\in\mathbb{N}} \{in_{C\|U^n}\},\ \bigcup_{n\in\mathbb{N}} Ap_{C\|U^n},\ \bigcup_{n\in\mathbb{N}} L_{C\|U^n},\ \bigcup_{n\in\mathbb{N}} \Sigma_{C\|U^n},\ \bigcup_{n\in\mathbb{N}} \delta_{C\|U^n}).$$

The states of $C\|U^n$ or $C\|U^\infty$ are called *global states* while the states of C or U are called *local states*.

Example 1 (Coordinator System)
As a running example we present a part of the *Common Component Modelling Example (CoCoME)* system [35]. *Coordinator* is a part of the system that is used for managing express checkouts. For this purpose, *Coordinator* keeps a list of sales that were done within the last 60 minutes and decides whether an express cash desk is needed. *Coordinator* consists of two types of sub-components: *CoordinatorEventHandler* (control component) and *Sale* (user) see Figure 1 a). Anytime a new sale arrives, *CoordinatorEventHandler* creates a new instance of the *Sale* component and displays it in the list. Whenever a sale represented by an instance expires, *CoordinatorEventHandler* removes the instance from the list which causes its destruction.

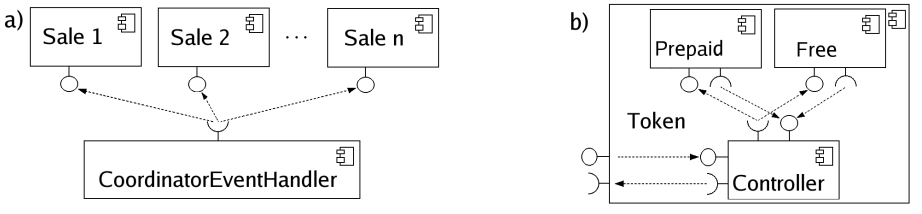


Fig. 1. a) CoordinatorEventHandler with n Sales, b) Component Token

We use the models of *CoordinatorEventHandler* and *Sale* as presented in [34]. In the models we abbreviate the name of the method `getNumberofItems()` to `gNI`, `getPaymentMode()` to `gPM`, `getTimeofSale()` to `gTS`, `updateStatistics()` to `uS`, and `isExpressModeNeeded()` to `iEMN`.

The control component C_{ex} is depicted in Fig. 2. Its set of atomic propositions corresponds to the set of labels, $Ap_{C_{ex}} = \Sigma_{C_{ex}}$. For a state $q \in Q_{C_{ex}}$ the set $L_{C_{ex}}(q)$ contains all labels which are enabled in the state. For example $L_{C_{ex}}(\mathtt{A}) = \{\mathtt{onEvent}\}$ and $L_{C_{ex}}(\mathtt{I}) = \{(\mathtt{gMP}, !), (\mathtt{SaleD}, !)\}$.

The user component U_{ex} is depicted in Fig. 3. Its atomic propositions are $Ap_{U_{ex}} = \Sigma_{U_{ex}} \cup \{activated, served\}$. For $q \in \{1, \ldots, 7\}$ the set $L_{U_{ex}}(q)$ con-

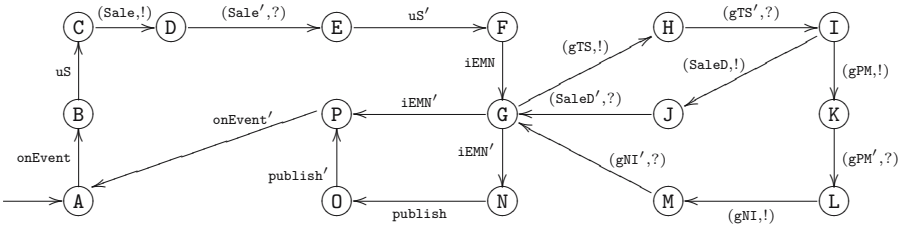


Fig. 2. LKS C_{ex} modelling *CoordinatorEventHandler*

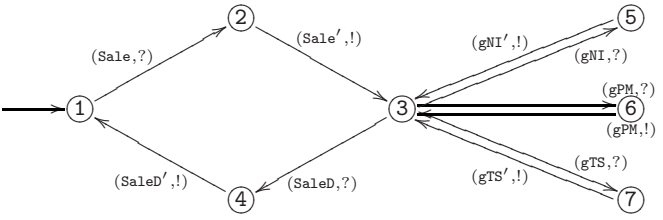


Fig. 3. LKS U_{ex} modelling *Sale*

tains labels which are enabled in the state, for $q \in \{3, 5, 6, 7\}$ it moreover contains the atomic predicate *activated*, and for $q \in \{2, 4, 5, 6, 7\}$ the set $L_{U_{ex}}(q)$ in addition contains *served*. For example $L_{U_{ex}}(2) = \{(\texttt{Sale}', !), \mathit{served}\}$ and $L_{U_{ex}}(7) = \{(\texttt{gTS}', !), \mathit{activated}, \mathit{served}\}$.

Note 1. Note that systems with more than one type of users can be modelled as C-U models as well. More precisely, the C-U model can represent an arbitrary system with a control part and a finite number of distinct types of users. The model of a user is an LKS which in the initial state non-deterministically chooses one of the given behaviours and after that it behaves like the choosen type of user. For example in the model of *Token and its support* (part of the prototype implementation of a payment system for public Internet on airports [29]) a Token (client) first chooses whether it will behave as a client with prepaid or free access to Internet (for model of the component see Figure 1 b)).

3 *l*-Symmetric Reachability Properties

We concentrate on verification of reachability properties of C-U models. Properties of interesting states are expressed as formulae of a propositional logic. Formulae are defined over a set of atomic propositions with the help of standard Boolean operators $\wedge, \vee, \neg$. A propositional formula is interpreted over a state of an LKS. A formula is *true* in a state iff after evaluating all atomic proposition assigned to the state as *true* and all others as *false* the result formula is *true*. In the following text we use a standard shortcut $\bigvee_{i \in \emptyset} \psi_i \equiv \mathit{false}$.

A reachability property (or RP for short) is a property capturing that a state satisfying a given propositional formula is reachable in the model $C \| U^n$ for some n. The *general reachability problem* for C-U models is formulated as:

Instance (reachability):
– *C-U model C, U*
– *sequence of formulae* $\{\varphi_n\}_{n \in \mathbb{N}}$*, where* φ_n *is a formula of the propositional logic over atomic propositions* $L_C \times \{0\} \cup L_U \times \{1\} \cup \cdots \cup L_U \times \{n\}$.
Problem: *Is there* $n \in \mathbb{N}$ *such that a state satisfying* φ_n *is reachable in* $C \| U^n$*?*

In the paper we are interested only in special types of reachability properties which make no distinction among users – so called *0-symmetric RP*, *1-symmetric RP*, etc. For a fixed $l \in \mathbb{N}_0$ and any $n \in \mathbb{N}$, an l-symmetric RP guarantees that

if a state $q \in Q_{C\|U^n}$ satisfies φ_n, then there are l users which together with the control component ensure that the state q satisfies φ_n. An instance of the l-symmetric reachability problem is:

Instance (l-symmetric reachability):

– *C-U model C, U, number $l \in \mathbb{N}_0$*

– *sequence of l-symmetric formulae $\{\varphi_n\}_{n\in\mathbb{N}}$, where for each $n \in \mathbb{N}$:*

$$\varphi_n = \bigvee_{f:\{1,\ldots,l\}\to\{1,\ldots,n\}} \psi_{(1,f(1)),\ldots,(l,f(l))}.$$

Here ψ is a formula of the propositional logic over atomic propositions defined $L_C\times\{0\}\cup L_U\times\{1\}\cup\cdots\cup L_U\times\{l\}$, f is an injective function, and $\psi_{(1,f(1)),\ldots,(l,f(l))}$ is the formula which results from ψ if we substitute each atomic proposition (a,i) by $(a,f(i))$ leaving $(a,0)$ untouched. We say that ψ is the propositional formula underlying $\{\varphi_n\}_{n\in\mathbb{N}}$.

Example 2. l-symmetric RPs of the C-U model C_{ex}, U_{ex} described in Example 1 are e.g. properties describing reachability of a state satisfying:

1. *Sale can send an event but CoordinatorEventHandler is not ready to accept it.* It is a 1-symmetric RP described by the sequence $\{\varphi_n\}_{n\in\mathbb{N}}$ with the underlying formula $\psi = \bigvee_{act\in\{\texttt{Sale'},\texttt{SaleD'},\texttt{gNI'},\texttt{gMP'},\texttt{gTS'}\}} ((act,!),1)\wedge\neg((act,?),0)$.
2. *Two Sales are able to send a response* `Sale'` *to CoordinatorEventHandler at the same time.* It is a 2-symmetric RP with the underlying formula $\psi = ((\texttt{Sale}',?),0)\wedge((\texttt{Sale}',!),1)\wedge((\texttt{Sale}',!),2)$.
3. *At least m Sales can be activated simultaneously.* It is an m-symmetric RP with the underlying formula $\psi = (activated,1)\wedge\cdots\wedge(activated,m)$.
4. *CoordinatorEventHandler can service (at least) m activated Sales simultaneously.* It is an m-symmetric RP with the underlying formula $\psi = (served,1)\wedge(activated,1)\wedge\cdots\wedge(served,m)\wedge(activated,m)$.

4 Backward Reachability

Backward reachability is one of the methods used in verification of parametrised systems for checking reachability of critical states [4,22,23]. For a given LKS S and a set of its states Q the question is whether a state from Q is reachable in the structure S. For iteratively increasing values j, one generates the set of states from which Q can be reached by a sequence of transitions of the length at most j. The backward reachability procedure terminates in the first iteration where the generated set of states does not increase comparing to the set generated in the previous iteration or if the last generated set of states contains an initial state. For transition systems with an infinite number of states termination of backward reachability is not guaranteed. However, the termination can be guaranteed for special sets Q.

Lemma 1. *Let C, U be a C-U model, $A \subseteq Q_C$. Then the backward reachability in $C\|U^\infty$ starting with $Q = \bigcup_{i\in\mathbb{N}} A\times\underbrace{Q_U\times\cdots\times Q_U}_{i}$ terminates.*

Proof: For the proof see [31].

5 Verification Algorithm

In this Section we present an algorithm which verifies a given l-symmetric RP. A naïve approach is to check all reachable states of the C-U model (more precisely, all reachable states of the LKSs $C\|U^1, C\|U^2, \dots$) for validity of the given property. As the number of the reachable states is infinite, we first define a finite number of their representatives.

Definition 4 (l+1-tuple). *Let C, U be a C-U model, $q \in Q_{C\|U^\infty}$ and $l \in \mathbb{N}_0$. Then an $l+1$-tuple $(q_C, q_1, \dots, q_l) \in Q_{C\|U^l}$ is* assigned *to the state q iff the local state of C in q is q_C and there are l different users in q with local states $q_1, \dots, q_l$.*

In a similar way we can assign an $l+1$-tuple to an $(l+i)+1$-tuple t' (t' is a state of $C\|U^{l+i}$).

$l+1$-tuples serve as an abstraction of the C-U model states where only the local states of l chosen users in an arbitrary order are maintained. Observe that for any fixed sequence $\{\varphi_n\}_{n\in\mathbb{N}}$ describing a l-symmetric RP a state q of $C\|U^i$ satisfies φ_i if and only if there is an $l+1$-tuple t assigned to q such that t (which is a state of $C\|U^l$) satisfies ψ (ψ is the propositional formula underlying $\{\varphi_n\}_{n\in\mathbb{N}}$). Hence if we find all $l+1$-tuples assigned to the reachable states of $C\|U^\infty$ (so called *reachable* $l+1$-tuples of $C\|U^\infty$) we can easily verify validity of the given l-symmetric RP[1]. Thus the core of our verification algorithm is a procedure for finding all reachable $l+1$-tuples of $C\|U^\infty$. As for each $l \in \mathbb{N}_0$ the number of $l+1$-tuples is finite there must exist a number k such that the set of all reachable $l+1$-tuples of $\{C\|U^n\}_{n\in\{l,\dots,k\}}$ is exactly the set of reachable $l+1$-tuples of $C\|U^\infty$. Let us call the number k *cutoff*.

The smallest number which can be a cutoff is the number $L = max(1, l)$. The algorithm which we propose searches for the minimal cutoff and returns all reachable $l+1$-tuples of $C\|U^\infty$. It iteratively traverses all reachable states of $C\|U^L, C\|U^{L+1}, \dots$ In the i-th iteration all $l+1$-tuples assigned to reachable states of $C\|U^{L+i}$ are computed and compared to those computed in the previous iteration. More precisely, as for any j the set of reachable $l+1$-tuples of $C\|U^j$ is a subset of the set of reachable $l+1$-tuples of $C\|U^{j+1}$, it is sufficient to compare their cardinality. Once there is no difference between the two sets, the number $L+i-1$ is the candidate for a cutoff. It can happen that there is an $l+1$-tuple assigned to a state reachable in $C\|U^{L+j-1}$ for some $j > i$ which is not covered yet. Therefore we need to verify whether $L+i-1$ is a cutoff.

To confirm that $L+i-1$ is a cutoff we run backward reachability in $C\|U^\infty$ from the set of its states to which a not yet covered $l+1$-tuple is assigned. If backward reachability finds that some of these states is reachable in $C\|U^\infty$, then the state must be reachable in $C\|U^k$ for some $k > L+i-1$ and the state is not reachable in $C\|U^k$ for $k \leq L+i-1$. Consequently $L+i-1$ is not a cutoff and we start the whole procedure with $L+i$. Otherwise $L+i-1$ is the minimal cutoff; the algorithm returns all reachable $l+1$-tuples of $C\|U^{L+i-1}$.

[1] In fact, having all reachable $l+1$-tuples of $C\|U^\infty$ we can verify any l'-symmetric RP for $l' \leq l$.

```
1  proc REACHABLE l + 1-TUPLES(C, U, l)
2    All_tuples := all l + 1-tuples
3    Cutoff := MAX(l, 1); Valid_cutoff := false
4    while Valid_cutoff = false do
5          FIND CUTOFF(Cutoff)
6          BACKWARD REACHABILITY(All_tuples \ Reached_tuples)
7          if Valid_cutoff = false then Cutoff := Cutoff + 1 fi
8    od
9    Cutoff = Cutoff − 1;
10   return Reached_tuples
```

```
1  proc FIND CUTOFF(Cutoff)
2    k := Cutoff; Reached_tuples := ∅
3    repeat  Old_tuples := Reached_tuples
4            Reached_tuples := all l + 1-tuples assigned to reachable states in C‖U^k
5            if |Old_tuples| ≠ |Reached_tuples| then k := k + 1 fi
6      until  |Old_tuples| = |Reached_tuples|
7    Cutoff := k − 1
```

```
1  proc BACKWARD REACHABILITY(T)
2    Q := {q∈ Q_{C‖U^∞} | an l + 1-tuple assigned to q belongs to T}
3    Q' := ∅; Reach := false
4    while (Q ≠ Q') ∧ (Reach = false) do
5          Q := Q ∪ Q'
6          Q' :=  predecessors of Q in C‖U^∞
7          if Q' ∩ in_{C‖U^∞} ≠ ∅ then Reach := true fi
8    od
9    if ¬Reach then Valid_cutoff := true fi
```

Fig. 4. Algorithm for computing all reachable $l + 1$-tuples

The pseudo-code of the algorithm is given in Fig. 4. The procedure FIND CUTOFF returns the first number k greater or equal to *Cutoff* such that the sets of reachable $l+1$-tuples of the LKSs $C\|U^k$ and $C\|U^{k+1}$ are the same. The set of all reachable $l+1$-tuples of $C\|U^j$ is monotonically increasing with increasing parameter j and at the same time the set of all possible $l+1$-tuples is finite. These two facts ensure that the procedure terminates.

The procedure BACKWARD REACHABILITY(T) first computes the set containing all states of $C\|U^\infty$ to which an $l+1$-tuple from T is assigned. By iterative searching of predecessors it decides reachability of states from T in $C\|U^\infty$.

Lemma 2. *The procedure* BACKWARD REACHABILITY *always terminates.*

Lemma 3. *Let for a C-U model C, U and $l \in \mathbb{N}_0$ the following condition is true:*

An unreachable $l+1$-tuple of $C\|U^\infty$ is assigned to every unreachable $(l+1)+1$-tuple of $C\|U^\infty$. (*)

Then BACKWARD REACHABILITY(T)*, where T is the set of all unreachable $l+1$-tuples of $C\|U^\infty$, terminates after the first iteration.*

Proof: For the proof of Lemma 2 and Lemma 3 see [31].

Example 3. Let us inspect the computation of REACHABLE $l+1$-TUPLES$(C_{ex}, U_{ex}, 1)$; where C_{ex},U_{ex} is the C-U model from Example 1.

In the first iteration of when $Reached\_tuples_{k=0} = \emptyset$ while-cycle FIND CUTOFF(1) is called and it iteratively computes the set *Reached_tuples*:

$$\begin{aligned} Reached\_tuples_{k=1} &= \{(x,\mathtt{1}),(x,\mathtt{3}) \mid x \in \{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{G},\mathtt{N},\mathtt{O},\mathtt{P}\}\} \cup \\ &\quad \{(x,\mathtt{3}) \mid x \in \{\mathtt{E},\mathtt{F},\mathtt{I},\mathtt{L}\}\} \cup \\ &\quad \{(\mathtt{D},\mathtt{2}),(\mathtt{H},\mathtt{7}),(\mathtt{J},\mathtt{4}),(\mathtt{K},\mathtt{6}),(\mathtt{M},\mathtt{5})\}, \\ Reached\_tuples_{k=2} &= \{(x,\mathtt{1}),(x,\mathtt{3}) \mid x \in \{\mathtt{A},\ldots,\mathtt{P}\}\} \cup \\ &\quad \{(\mathtt{D},\mathtt{2}),(\mathtt{H},\mathtt{7}),(\mathtt{J},\mathtt{4}),(\mathtt{K},\mathtt{6}),(\mathtt{M},\mathtt{5})\}, \\ Reached\_tuples_{k=3} &= Reached\_tuples_{k=2}. \end{aligned}$$

After that BACKWARD REACHABILITY($All\_tuples \setminus Reached\_tuples_{k=3}$) is called.

Iteration 0: Q contains all states of $C_{ex} \| U_{ex}^{\infty}$ to which a tuple from $T = \{(x,\mathtt{2}),(x,\mathtt{4}),(x,\mathtt{5}),(x,\mathtt{6}),(x,\mathtt{7}) \mid x \in \{\mathtt{A},\ldots,\mathtt{P}\}\} \setminus \{(\mathtt{D},\mathtt{2}),(\mathtt{H},\mathtt{7}),(\mathtt{J},\mathtt{4}),(\mathtt{K},\mathtt{6}),(\mathtt{M},\mathtt{5})\}$ is assigned,

Iteration 1: Q contains all states of $C_{ex} \| U_{ex}^{\infty}$ to which a tuple from $T \cup \{(\mathtt{D},\mathtt{2},\mathtt{2}),(\mathtt{H},\mathtt{7},\mathtt{7}),(\mathtt{J},\mathtt{4},\mathtt{4}),(\mathtt{K},\mathtt{6},\mathtt{6}),(\mathtt{M},\mathtt{5},\mathtt{5})\}$ is assigned,

Iteration 2: Q is the same as for the iteration 2.

Thus after 2 iterations BACKWARD REACHABILITY terminates and returns that the found cutoff 2 is valid, consequently REACHABLE $l+1$-TUPLES returns the set $Reached\_tuples_{k=2}$.

We should stress that even if the algorithm is presented as a procedure for finding all reachable $l+1$-tuples of $C \| U^{\infty}$ it is in fact a verification algorithm. As pointed out at the beginning of this Section, keeping the set of all reachable $l+1$-tuples of $C \| U^{\infty}$ one can decide validity of any l'-symmetric RP for an arbitrary $l' \leq l$.

Note 2 (Optimisation). There are several possible optimisations of the algorithm. We list the two most important.

- Symmetry reduction [10] in the algorithm decreases the number of both reachable states and tuples exponentially.
- On-the-fly approach to verification: as soon as an $l+1$-tuple is reached for the first time it is checked for validity of given reachability properties.

Note 3 (Verification). There is another important way how to use the presented algorithm, namely verification of an updated (modified) system. When updating a system we usually want to guarantee that the new system satisfies all important properties which the original system satisfies. The problem is that it is hard to enumerate all (important) properties of the original system. In such a case it is profitable to use the given algorithm and compute differences between the reachable $l+1$-tuples in the original and the updated system.

6 Evaluation

In order to test efficiency of the proposed algorithm we use several models of previously published real component-based C-U systems (R_I - R_{IV}), simplified

real systems (S_I, S_{II}), and simple C-U systems proposed for evaluation of our algorithm (E_I, E_{II}). The inspected C-U models are: R_I - model of *Coordinator* (Example 1), R_{II} - model of *Token and its support* (Note 1), R_{III} - model of *Cash desk and its support* (Fractal model in [7]), R_{IV} - model of *Subject - Observer system* with n subjects (published in [33], model of the system [3]). S_I and S_{II} are models of *Comanche Web Server* with a Sequential resp. Multi Thread Scheduler and their clients, published in [1]. E_I is a system where the controller provides 5 services in parallel and users can use the services sequentially. E_{II} is a system where the controller provides 2 services and in all states it can receive or return any request, users use the services sequentially. Detailed description of all models and their characteristics are on the web page [2].

Table on Fig. 5 displays for a given C-U model and a parameter $l \in \{1, 2, 3\}$ the number of states of $C\|U^k$ for maximal k for which the state space is generated in FindCutoff (*States*), the minimal cutoff (*Cutoff*), and the number of iterations of backward reachability (*Iterations*). Based on experimental evaluation we conclude:

1. In all cases the while-cycle in Reachable $l+1$ -tuples was performed only once - the *Cutoff* computed in FindCutoff was the valid minimal cutoff.
2. For each of the models and every $l \geq 2$ the condition *(*)* holds. It means that for each of the models and every $l \geq 2$ to arbitrary unreachable $l+1$-tuple of $C\|U^\infty$ is assigned an unreachable $2+1$-tuple of $C\|U^\infty$. Thus for each model and an arbitrary $l \leq 2$ Backward Reachability terminates after the first iteration.
3. The minimal cutoff for $l+1$-tuples is typically the minimal cutoff for $0+1$-tuples plus l.

model	*l*	*States*	*Cutoff*	*Iterations*
R_I	1	144	2	2
	2	332	3	1
	3	748	4	1
R_{II}	1	145 125	4	2
	2	1 091 875	5	1
	3	7 821 875	6	1
R_{III}	1	297 108	4	2
	2	1 706 103	5	1
	3	8 957 952	6	1
R_{IV}	1	48 320^n	4	2
	2	266 240^n	5	1
	3	1 425 920^n	6	1

model	*l*	*States*	*Cutoff*	*Iterations*
S_I	1	1 126	2	2
	2	4 910	3	1
	3	20 830	4	1
S_{II}	1	7 514	2	2
	2	142 476	3	1
	3	2 672 672	4	1
E_I	1	4 051	6	2
	2	9 276	7	1
	3	19 080	8	1
E_I	1	69	3	1
	2	245	4	1
	3	312	5	1

Fig. 5. Evaluation of the verification algorithm

7 Bounding Algorithm

When analysing C-U systems we are often interested in the highest possible number of users which are simultaneously in the same state (situation, part of a computation). For instance, we can ask how many users have started a communication with the control component and have not finished it yet, or how many users are demanding the same service. This can be described by a sequence of reachability properties $\{P_m\}_{m\in\mathbb{N}}$ such that for each m the property $P_m = \{\varphi_n^m\}_{n\in\mathbb{N}}$ is an m-symmetric RP expressing that *It is possible to reach a global state in which at least m users are in the same specified setting.*

Example 4. Motivations can be found e.g. in Example 2, properties 3 and 4: *What is the maximal number of Sales which can be activated simultaneously?* or *What is the maximal number of activated Sales which can CoordinatorEventHandler service simultaneously?*

Lemma 4. *Let $\{P_m\}_{m\in\mathbb{N}}$ be a sequence where every P_m is an m-symmetric RP with the underlying formula ψ_m. Let the implication $\psi_j \Rightarrow \psi_i$ be true for every $j > i$. Then if $C\|U^\infty$ satisfies P_j then it also satisfies P_i for each $i \leq j$. (**)*

A sequence of RPs satisfying the condition of Lemma 4 is denoted *integrated sequence of RPs*. Note that an equivalent to the condition (**) is: if $C\|U^\infty$ does not satisfy P_i then it does not satisfy P_j for any $i \leq j$.

For a given integrated sequence of RPs we propose an algorithm for computing a *bound* which is the number b such that P_b is satisfied and P_{b+1} is not satisfied (if it exists). In practise we are often given a value *Max* and the question is whether the *bound* is at most *Max* and only if this is the case we want to know the exact value of *bound*. The task which we study can be described as

Instance:

– *C-U model C, U, number $Max \in \mathbb{N}$*
– *sequence $\{P_m\}_{m\in\{1,\ldots,Max\}}$, where P_i is an i-symmetric RP satisfying (**)*

Problem: *Compute*

$$bound \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } P_1 \text{ is not satisfied,} \\ b & \text{if } P_b \text{ is satisfied and } P_{b+1} \text{ is not satisfied}, 1 \leq b < Max, \\ Max & \text{if } P_{Max} \text{ is satisfied.} \end{cases}$$

From the Section 5 it follows that for checking the property P_i from the sequence $\{P_m\}_{m\in\{1,\ldots,Max\}}$ it suffice to compute all $i+1$-tuples reachable in $C\|U^\infty$. A trivial bounding algorithm, using the algorithm given in the previous section, first finds all $1+1$-tuples reachable in $C\|U^\infty$ and checks P_1, then it finds $2+1$-tuples reachable in $C\|U^\infty$ and checks P_2, etc. However, this approach is not effective enough, for explanation see item 3 in Section 6. This motivate us to propose another algorithm which instead of computing all $i+1$-tuples reachable in $C\|U^\infty$ for each i over-approximate the sets of tuples.

The core idea is that if we have a high integer b, we can choose a small integer k and under-approximate the set $Unreach\_b$ of all unreachable $b+1$-tuples of $C\|U^\infty$ by the set $Unreach\_b\_k$ of all $b+1$-tuples to which is assigned

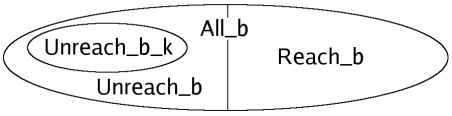


Fig. 6. The sets of $b+1$-tuples *All_b*, *Unreach_b*, *Reach_b*, and *Unreach_b_k*

an unreachable $k+1$-tuple of $C\|U^\infty$. The profit is that to compute $Unreach\_b\_k$ instead of $Unreach\_b$ it is necessary to find all reachable $k+1$-tuples instead of all reachable $b+1$-tuples. The set $Unreach\_b\_k$ serves also as an over-approximation of all reachable $b+1$-tuples of $C\|U^\infty$. Let us denote All_b the set of all possible $b+1$-tuples of $C\|U^\infty$. Then the set of all reachable $b+1$-tuples of $C\|U^\infty$ can be over-approximated $Reach\_b\_k = All\_b \setminus Unreach\_b\_k$ of all $b+1$-tuples to which is not assigned an unreachable $k+1$-tuple of $C\|U^\infty$.

The pseudo-code of the algorithm is given in Fig. 7. The algorithm in the first cycle (lines 3-9) iteratively finds the minimal k such that to each unreachable $k+1$-tuple of $C\|U^\infty$ is assigned an unreachable $(k-1)+1$-tuple of $C\|U^\infty$. For every value of $l \leq k$ the algorithm tests whether $l-1 = bound$ (line 7). If the *bound* is greater than $k-1$, then in the next step (line 10) the algorithm computes the maximal $b \in \{k, \ldots, Max\}$ such that a $b+1$-tuple from $Reach\_b\_k$ satisfies the property P_b. The inclusion $Reach\_b\_k \supseteq Reach\_b$ implies that the property P_b may but must not be satisfied. On the other hand if $b < Max$ then the property P_{b+1} can not be satisfied. Thus the number b is a maximal value which can be the bound and for us it is a candidate for the bound. Consequently the algorithm computes which of the values $k, \ldots, b$ is the bound (line 11).

The procedure REACHABLE $l+1$-TUPLES is described in Section 5. The procedure IS_$k-1$_BOUND computes, using the set of all reachable k-tuples of $C\|U^\infty$ *Reach_New*, whether $k-1$ is the bound. GENERATE_POSSIBLE_NEW generates all $k+1$-tuples to which is assigned a $(k-1)+1$-tuple in the input set. The procedure LAST_SATISFIED returns the maximal number b from the set $\{k, \ldots, Max+1\}$ such that the intersection of all tuples unsatisfying P_b and $Reach\_b\_k$ is not empty. The procedure gradually tests $b = k, \ldots, Max+1$. The procedure VALIDATEBOUND tests which of the numbers $k, \ldots, b$ is the result. If $b = Max$ then it is the valid result, else it firstly checks whether b is the bound and after that it tests $k, k+1$, etc.

Evaluation. As noted in item 2 in Section 6 for each examined model and an arbitrary $k \geq 2$ the condition *(*)* holds. Consequently for any of the presented models and arbitrary b, k, where $b \geq k \geq 2$, equalities $Unreach\_b\_k = Unreach\_b$ and $Reach\_b\_k = Reach\_b$ hold. Thus the procedure BOUND for all studied models and any sequence of described properties found the correct bound in the while-cycle (lines 3-9) - if it is less then 3. If the bound is greater of equal to 3, than the algorithm computes b as the bound (line 10) and than it successfully verifies, that P_b is satisfied. Our experience with verification of different properties of component-based systems is that a good choice for the value Max is $|Q_C|+1$ as usually if the bound is finite then it is at most $|Q_C|$.

```
1  proc Bound(C, U, Max, {P_i}_{1≤i≤Max})
2    k := 0; Unreach_New = ∅
3    repeat  k := k + 1
4            Unreach_Old := Unreach_New
5            Reach_New := Reachable l + 1-tuples(C, U, k)
6            Unreach_New := all k+1-tuples \ Reach_New
7            if Is_k − 1_Bound() then return k-1 fi
8            Changes = Unreach_New \ Generate_Possible_New(Unreach_Old)
9      until (Changes = ∅) ∨ (k ≥ Max)
10   b := Last_Satisfied(Unreach_Old)
11   return ValidateBound(b, Unreach_Old)
```

```
1  proc Is_k − 1_Bound
2    Old_Satisfied = New_Satisfied;
3    New_Satisfied = Is_P_k_Satisfied(Reach_New)
4    if Old_Satisfied ∧ (¬New_Satisfied) then return true
5                                        else return false
```

Fig. 7. Algorithm for computing the bound

8 Related Work

Many papers address parametrised systems and their verification, we relate our contribution to the previously published results in several aspects.

Computational model We consider control-user systems of the form $C \| U^n$ with finite models of C and U. Components communicate using the pairwise rendezvous synchronisation, and there are no variables in the model. Similar models are studied in [6,8,21].

Two other approaches to modelling C-U system can be found in the literature. The first one is a model containing a parametrised number of identical finite state components (modelling users) with a finite set of global variables (modelling states of the control component) where an individual user can make a transition if the global variables satisfy a Boolean guard, see [12,26,27,28]. The second approach is to model a C-U system with a finite number of control states (modelling the control component), infinite set of data values $\mathbb{N}_0^{|Q_U|}$ (corresponding to the number of users in each state from Q_U), and appropriate synchronisation, see [4].

Cutoff Verification methods based on a cutoff have been successfully applied to several types of properties of various parametrised systems. In [21] an approach implicitly based on a cutoff was used for proving that verification of l-symmetric reachability properties is decidable but the algorithm runs in triple exponential time and thus it is impractical. Other papers [15,16,19,20] propose algorithms which are more efficient however these are not applicable to our model of control-user systems. In our previous work [32] we studied verification of LTL_X properties using a cutoff. This algorithm is incomplete and for several l-symmetric properties with $l > 0$ does not terminate.

Verification Among a number of approaches to verification of parametrised systems [4,6,8,12,18,19,20,21,22,23,26,27,28] there are several fully automatic techniques which can be used for systems which we study in the paper.

Several of the approaches are based on (backward or forward) reachability analysis [4,22,23,26,30]. The paper [4] presents verification of reachability properties for general types of systems using backward reachability analysis. The

authors prove that for a general type of systems and a special type of reachability properties (including l-symmetric properties) the algorithm terminates, but the number of iterations in which it terminates is not known. The symbolic backward and forward reachability analysis for general rich assertional languages using regular sets and acceleration is performed in [23]. The paper [22] extends [4]. It studies general program model and using a transitive closure generation and acceleration of actions it performs a reachability analysis on those systems. In [26] the author uses a tree based decision procedure which generates predecessors to solve this problem. The main purpose of the paper is to prove a decidability of verification of safety properties for a broad class of systems and thus the paper does not contain any experimental results. Authors in [30] study backward analysis using the local backward reachability algorithm.

Another approach to verification of safety properties works with invisible invariants [5,20,27,28]. This incomplete approach is based on computing an inductive assertion. Our approach can be seen in some aspects as a restriction of this approach (inductive assertion is exactly determined by reachable $l+1$-tuples). Contrary to the mentioned algorithms our algorithm always terminates.

Papers [8,12] propose an algorithm based on generating abstract finite model of infinite users (an over-approximation of the model of the system with U^{∞}). Paper [12] moreover studies in which cases the algorithm returns a valid counterexample.

The technique presented in [6] takes each instance of a parametrised system as an expression of a process algebra and interprets this expression in modal mu-calculus, considering a process as a property transformer. The result is an infinite chain of mu-calculus formulae and technique solves the verification problem by finding the limit of a chain of formulae.

To sum it up, our verification algorithm is complete (contrary to [5, 8, 12, 27, 28, 32]), it computes a cutoff for l-symmetric RPs (not only checks an l-symmetric RP, contrary to [4, 6, 22, 23, 26, 30]), and the found cutoff is minimal (contrary to [21, 32]). Experiments demonstrate that the number of backward reachability iterations is typically very small (contrary to [4, 22, 23, 26, 30]) but steps of backward reachability in our algorithm are usually quite complex. As it was mentioned our verification algorithm computes the cutoff for l-symmetric RPs. Consequently our verification algorithm is important especially whenever one needs to find several types of possible errors in the system. The algorithm computes the reachable $l+1$-tuples and after (or during) this computation it verifies which of the given properties are fulfilled.

As far as we know there is no other algorithm for verification of integrated sequences of RPs. Experiments show that the algorithm BOUND typically estimates the value of the bound correctly and thus is very efficient.

9 Conclusions

The paper studies systems composed of a control component and a dynamic number of user components (Control-User systems). Such type of systems is

often of use in component-based systems e.g. when a central component provides services to unspecified number of clients. Safety properties are used to express that the system cannot enter an undesired configuration. The complexity of verification of reachability problems for Control-User systems stems from the fact that we want to guarantee the correctness for every possible number of users communicating with the control component. Though the problem of verification of symmetric safety properties on C-U system is decidable [21,4], the state-space explosion is highly limiting factor for practical usage of verification techniques on those systems.

The paper presents two verification algorithms. The first algorithm solves the problem whether a given C-U model satisfies a given (finite) set of l-symmetric properties. The second algorithm is for computing the largest number of users which can be at the same time in a specific situation, state (so called *bound*).

The algorithms are evaluated on several C-U models of component-based systems (see also [2]). Characteristics of these models confirm practical usability of both algorithms as only instances with very low number of user components have to be explored during the algorithm.

An open question is whether similar approaches can be used to verify a wider class of reachability properties.

References

1. http://fractal.objectweb.org/tutorial/index.html
2. http://anna.fi.muni.cz/coin/CUmodels/
3. Poetzsch-Heffter, A., Aldrich, J., Barnett, M., Giannakopoulou, D., Leavens, G.T., Sharygina, N.: Challenge Problem SAVCBS 2007 (May 2007), http://www.eecs.ucf.edu/leavens/SAVCBS/2007/challenge.shtml
4. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: LICS 1996. IEEE Computer Society, Los Alamitos (1996)
5. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102. Springer, Heidelberg (2001)
6. Basu, S., Ramakrishnan, C.R.: Compositional analysis for verification of parameterized systems. Theor. Comput. Sci. 354(2), 211–229 (2006)
7. Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parizek, P., Plasil, F., Poch, T., Rivierre, N., Sery, O., Tuma, P.: CoCoME in Fractal. In: PACS 2000. LNCS, vol. 5153. Springer, Heidelberg (2008)
8. Calder, M., Miller, A.: An automatic abstraction technique for verifying featured, parameterised systems. Theoretical Computer Science (to appear, 2008)
9. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999. Springer, Heidelberg (2004)
10. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1-2), 77–104 (1996)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, USA (2000)

12. Clarke, E.M., Talupur, M., Veith, H.: Proving ptolemy right: The environment abstraction principle for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
13. Dickson, L.E.: Finiteness of the odd prerfect and primitive abundant numbers with r distinct prime factors. Amer. Journal Math. 35, 413–422 (1913)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP, pp. 7–15. ACM Press, New York (1998)
15. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
16. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
17. Emerson, E.A., Kahlon, V.: Model checking guarded protocols. In: LICS 2003, pp. 361–370. IEEE Computer Society, Los Alamitos (2003)
18. Emerson, E.A., Kahlon, V.: Rapid parameterized model checking of snoopy cache coherence protocols. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 144–159. Springer, Heidelberg (2003)
19. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL 1995, pp. 85–94. ACM, New York (1995)
20. Fontaine, P., Gribomont, E.P.: Decidability of invariant validation for paramaterized systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 97–112. Springer, Heidelberg (2003)
21. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
22. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 220–234. Springer, Heidelberg (2000)
23. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theor. Comput. Sci. 256, 93–112 (2001)
24. Kofroň, J.: Behavior Protocols Extensions. PhD thesis, Charles University in Prague (2007)
25. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly 2(3), 219–246 (1989)
26. Maidl, M.: A unifying model checking approach for safety properties of parameterized systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 311–323. Springer, Heidelberg (2001)
27. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
28. Pnueli, A., Zuck, L.D.: Model-checking and abstraction to the aid of parameterized systems. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, p. 4. Springer, Heidelberg (2002)
29. Component reliability extensions for fractal component model, http://kraken.cs.cas.cz/ft/public/public_index.phtml
30. Rybina, T., Voronkov, A.: Using canonical representations of solutions to speed up infinite-state model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404. Springer, Heidelberg (2002)

31. Vařeková, P., Černá, I.: Model Checking of Control-User Component-Based Parametrised Systems. Technical Report FIMU-RS-2008-06, Masaryk University, Faculty of Informatics, Brno, Czech Republic (2008)
32. Vařeková, P., Moravec, P., Černá, I., Zimmerova, B.: Effective-Verification of Systems with a Dynamic-Number of Components. In: SAVCBS 2007, pp. 3–13. ACM Press, New York (2007)
33. Vařeková, P., Zimmerova, B.: Solution of challenge problem. In: SAVCBS 2007. ACM Press, New York (2007)
34. Zimmerova, B., Vařeková, P.: Reflecting creation and destruction of instances in CBSs modelling and verification. In: MEMICS 2007, Znojmo, Czech Republic (2007)
35. Zimmerova, B., Vařeková, P., Beneš, N., Černá, I., Brim, L., Sochor, J.: Component-Interaction Automata Approach (CoIn). In: PACS 2000. LNCS, vol. 5153, pp. 146–176. Springer, Heidelberg (2008)

Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems

Andreas Both and Wolf Zimmermann

Institute of Computer Science, University of Halle, 06099 Halle/Saale, Germany
andreas.both@informatik.uni-halle.de,
wolf.zimmermann@informatik.uni-halle.de

Abstract. Today model checking of security or safety properties of component-based systems based on finite protocols has the flaw that either parallel or sequential systems can be checked. Parallel systems can be described often by well known Petri nets, but it is not possible to model recursive behaviour. On the other hand sequential systems based on pushdown automata can capture recursion and recursive callbacks [27], but they do not provide parallel behaviour in general.

In this work we show how this gap can be filled if process rewrite systems (introduced by Mayr [16]) are used to capture the behaviour of components. The protocols of the components interfaces specified as finite state machines can be combined to a system equal to a process rewrite system. By calculating the reachability of the fault state range one gets a trace (counterexample) which does not satisfy the properties specified by all protocols of the combined components, if any error exists.

1 Introduction and Motivation

Modern software development contains a big share of reusing previously developed software called components. Often these components are developed by third party companies and supplied in binary code or as Web Service. So it is not easy to have a look at the source code to collect the behaviour. Hence the supplier should deliver together with the component a protocol of the interfaces and an abstraction of the component which specifies the behaviour, to give us the ability to check certain properties, e. g. abortion freeness.

The protocol is in general specified as finite state machine. Today the abstractions are often specified in one of the following four ways:

- By using *Petri nets* it is possible to specify parallel behaviour like threads as well as synchronous and asynchronous method calls, but no recursion in general.
- By using *pushdown automata* (PDA) it is possible to specify recursion and synchronous method calls, but no threads or asynchronous calls in general. [27]

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 163–179, 2008.

- By using *finite state machines* (FSM) ([18, 23]) a kind of parallel behaviour can be described (cf. [22]), but no recursion in general. Normally FSM can be represented as PDA or Petri nets.
- By using *process algebras* such as CSP [1], these approaches are more powerful than FSM and PDA, but at the end the conformance checking reduces to checking FSM [12].

In (modern) component systems like OMG's CORBA, Sun's JavaEE or Microsoft's .NET parallel concepts as well as sequential concepts are permitted. Hence it will be nearly impossible to use exclusively Petri nets or pushdown automata in practice to model the behaviour of components.

Our idea is to use another representation to model the behaviour of components and component-based systems and to develop a tool which proves the absence of component protocol violations. This tool should advance the verification of a component-based system by automatically verifying the protocol conformance of its components. No expert of verification will be necessary.

In Section 2 definitions of components, protocols and abstractions will be described. Creating abstractions of a full program will be shown in Section 3. In Section 4 the so called Combined Abstraction will be introduced, which is a representation of the full component-based system in combination with a considered protocol. We show in Section 5 how a counterexample can be calculated, while solving a reachability problem. To converge our approach to real component-based systems we show in Section 6 how abstractions of single components can be constructed without knowledge of the other components in the component-based system, and how they are assembled to a full component-based system abstraction.

2 Terms and Definitions

2.1 Components and Component-Based Systems

We assume that a component is an implementation of each provided interface. It is possible, that a component uses provided interfaces, thus has required interfaces. We make no restrictions on the language nor location where the component is deployed. Of course the implementation could be inaccessible (e. g. Web Service). Our assumptions are summarized in Figure 1.

A component-based system is assembled by components which communicate only over required (and provided) interfaces. These interfaces consist of a set of functions/procedures. We allow synchronous and asynchronous interfaces. Called synchronous interfaces block their caller until the callee has been completed the call, while asynchronous interfaces start a new thread when they are called. Hence the caller can proceed without waiting for completion. We also assume, that a components interface does not contain the information if it is implemented synchronous or asynchronous in general.[1]

We allow call-backs, but no external dynamic instances of a component.

[1] For simplification we specify in our example the communication method.

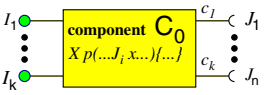


Fig. 1. Component C_0 with the required interfaces $J_1, \ldots, J_n$ and the provided interfaces $I_1, \ldots, I_n$

2.2 Protocol

A protocol of a component describes the use of all interfaces (remote use of functions) of the component. It can be used to verifiy dynamically incoming (remote) method calls, and also to verifiy the components statically. Using model checking we consider the latter in this work because it has the advantage that component-based systems are checked before they are deployed by the customer. This is a model checking problem, which is much harder to solve than verifing dynamically. Nevertheless component protocols could be used with both approaches to increase safety and security.

Creating and verifying protocols can help to ensure the restrictions of business rules. For example, using a SSO-component[2] in a system with the following actions, *a*. register and sign in, *b*. sign in, *c*. optionally change password, *d*. logout, could have the following business rule respectively protocol A formulated as regular expression: $R_A = ((a|b)c^*d)^*$.

This protocol should be obeyed by every client. We will check automatically, if a component-based system using the mentioned SSO-service protocol obeys the defined constraints.

In accordance with other works [11, 21, 27] we use FSM to represent the protocol A. The FSM $A = (Q_A, \Sigma_A, \rightarrow_A, I_A, F_A)$ is defined as usual, i. e. Q_A is a finite set of states, Σ_A is a finite set of atomic actions, $\rightarrow_A \subseteq Q_A \times \Sigma_A \times Q_A$ is a finite set of transition rules, $I_A \in Q_A$ is the initial state, $F_A \subseteq Q_A$ is the set of final states.

Note that the protocol as FSM gives us the ability to show this protocol to the developers, to create a graphical representation (to use it in the development process), and to use it for automated verification.

Many approaches [11, 21] model the use of required interfaces by regular languages obtained by finite transducers. In [27] it is shown that this approach leads to false positives if recursion is present.

The use of a component C_i in a component-based system is the set of possible sequences of calls to C_i. Thus, this can be also modeled as a language L^i_{Π}. Hence the protocol conformance checking is equivalent to check whether $L^i_{\Pi} \subseteq L_{P_i}$, when L_{P_i} is the language defined by the protocol P_i of C_i.

In Figure 2 an example of a component-based system including the components implementations and protocols is shown.

[2] Single Sign On. A component which provides the functionality of a login/logout/session management, so different applications can use this mechanism to verify a user.

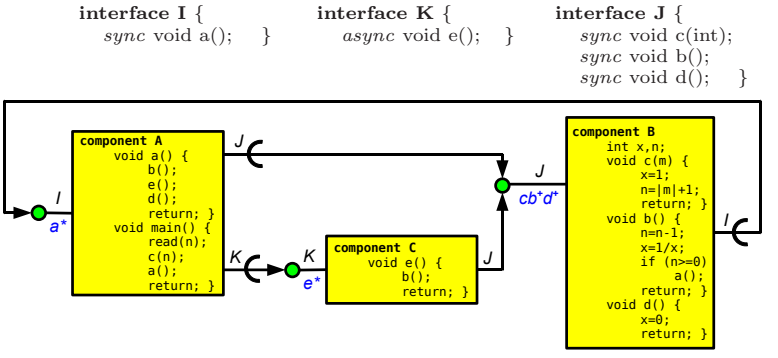


Fig. 2. Example. A component-based system assembled from the components A (implements interface I), B (implements interfaces J) and C (implements interface K). The components have following protocols, given as regular expessions: A: a^*, B: cb^+d^+, C: e^*.

In this work we will verify if components interfaces are used in the manner the developer specified by a protocol. By doing this, we can exclude semantic errors which appear because of an unexpected sequence of (remote) method calls.

For a more detailed proof we also need an abstraction of the behaviour of the component.

2.3 Process Rewrite Systems (Short PRS)

The abstracted behaviour of a component can be modeled with different representations. An abstraction $\mathcal{A}_C$ of a component C describes the behaviour $\mathcal{B}_C$ of C. Every possible execution path of C has a counterpart (trace) in $\mathcal{A}_C$. There exists a mapping from $\mathcal{B}_C$ to $\mathcal{A}_C$.[3]

An abstraction $\mathcal{A}_C$ has to implement every path of a component C, every control flowpath has to be recognized. In the work [27] parameterized context free systems (equal to PDA) were used to integrate recursion. The parameterization is required to implement callbacks. Because we transform a turing-powerful implementation to a not turing-powerful representation, the created abstraction $\mathcal{A}_C$ will approximate the behaviour of C, but this way we find a protocol violation, if there exists one.

As mentioned above, both representations (Petri nets and PDA) have advantages. We consider a representation which contains parallel semantic (like Petri nets) as well as sequential semantic (like PDA). Hence the base of this work will be the use of a representation called process rewrite systems (short PRS) defined by Mayr [16].

PRS unify the semantic of Petri nets and PDA. Mayr introduced an operator for parallel composition "||" and sequential composition ".". A process rewrite system $\Pi = (Q, \Sigma, I, \rightarrow, F)$ is defined as followed:

[3] The mapping $\mathcal{B} \rightarrow \mathcal{A}$ is often not bidirectional.

Q	is a finite set of atomic processes,
Σ	is a finite alphabet over actions,
$I \in Q$	is the initial process,
$\rightarrow \subseteq PEX(Q) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q)$	is a set of process rewrite rules,
$F \subseteq PEX(Q)$	is a finite set of final processes.

We introduce a special action λ, denoting *no action* or *emtpy word*. The set $PEX(Q)$ contains all process-algebraic expressions over the set of atomic processes Q. The process rewrite rules define a derivation relation $\stackrel{a}{\Rightarrow} \in PEX(Q) \times \Sigma^* \times PEX(Q)$ by the following inference rules ($a \in \Sigma \cup \{\lambda\}, x \in \Sigma^*$):

$$\frac{(u \stackrel{a}{\rightarrow} v) \in \Pi}{u \stackrel{a}{\Rightarrow} v}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{u.w \stackrel{a}{\Rightarrow} v.w}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{u||w \stackrel{a}{\Rightarrow} v||w}, \quad \frac{u \stackrel{a}{\Rightarrow} v}{w||u \stackrel{a}{\Rightarrow} w||v}, \quad \frac{u \stackrel{x}{\Rightarrow} v \quad v \stackrel{a}{\Rightarrow} w}{u \stackrel{x\,a}{\Longrightarrow} w}$$

$L_\Pi \hat{=} \{w : \exists f \in F | I \stackrel{w}{\Rightarrow} f\}$ is the language accepted by Π.

Based on the operators Mayr defined a hierarchy of PRS classes that allows us the classification of process rewrite systems by the appearance of operands. Mayr uses the following base classes:

- 1: terms are composed of atomic processes only
- P: terms are composed of atomic processes or parallel composition
- S: terms are composed of atomic processes or sequential composition
- G: terms can be formed with all operators

These classes model different behaviour. Hence it is not possible to model all behaviour of a parallel system only with sequential composition and vice versa (cf. Figure 3a).

With the four base classes, a hierarchy based on bisimulation was formed (cf. Figure 3b), which allows us to classify all possible and sensible PRS.[4]

As we see, the $(1, S)$-PRS allows rules, which contain a process constant at the left-hand side and allows the sequential operator at the right-hand side. Thus this class is equivalent to PDA with one state, which accepts a language, if the stack is empty. The empty stack is represented in a $(1, S)$-PRS with the emtpy process ε. The (S, S)-PRS is the companion piece to PDA with several states.

PRS which allows the parallel operator are among others the class (P, P)-PRS which is equivalent to the well known Petri nets. The $(1, P)$-PRS are Petri nets, where every transition of the net has only one incoming arc, hence it does not contain synchronization.

By looking at the PRS hierarchy in Figure 3b we see, that if we search for a fusion of Petri nets with sequential concepts we have to use the (P, G)-PRS "PAN"[5], but also the $(1, G)$-PRS "Process Algebra" (short PA) could be interesting, if we have no synchronization of components.

[4] [16] points out that the left-hand side of a PRS-rule must be at most as large as the right-hand side in the sense of Figure 3a.

[5] Caused by the fact, that the grammar described by a (S, S)-PRS PDA can be accepted by a $(1, S)$-PRS called BPA, which is a pushdown automata with only one state.

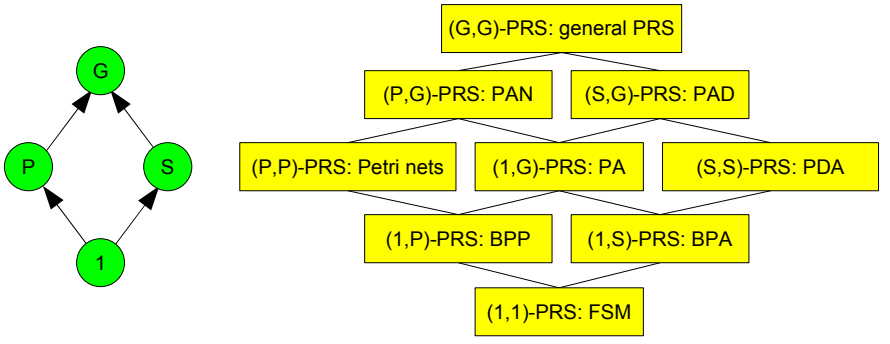


Fig. 3. a) Hierarchy of basic PRS operators, b) Hierarchy of process rewrite systems (cf. [2]), classification by appearance of operands on the left-hand side and right-hand side, (lhs,rhs)-PRS

Using PRS as an abstraction model, we are able to deal with real programs, because all important behaviour like recursion, threads, synchronous and asynchronous remote function calls can be modeled.

In this paper we will focus on $(1, G)$-PRS called Process Algebras because it can handle recursion and parallel behaviour, and our component model does not contain synchronization. Mayr has shown, that reachability is solvable, therefore Process Algebras can be considered for our application.

3 Building the Use of Components as PRS

Now we will create an PRS abstraction $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S)$ of the component-based System S in a $(1, G)$-PRS representation. We assume here, that the full source code is available. The main ideas for the construction are:

1. Create an atomic process for each program point p_i of a component C: Without loss of generality we assume, that every control flow path of a method ends with a `return`-statement. For `return`-statements no program point will be created.
2. Create transition rules, which map the control flow of the component in process rewrite rules: We use the mapping function $next : p_i \rightarrow p_j$, which results in the program points $p_j \in Q$, which are the possible succeeding program points of p_i. The mapping result contains ε if there exists a control path, that ends in the next step (`return`-statement).
   - If at a program point p_i a synchronous method call a is performed, we create rewrite rules $p_i \overset{a}{\rightarrow}_S p_j \ . \ p_k$ and $p_i \overset{a}{\rightarrow}_S p_j \ || \ p_k$, if a is an asynchronous method call.
   - If at a program point p_i another operation is performed, we create rewrite rules $p_i \overset{\lambda}{\rightarrow} p_k$, where $p_k \in next(p_i)$. This transition rule has the semantic, that this operation is not interesting for the protocol verification.

We always update Q_C, if we create a new rewrite rule.

Note that all these pieces of information can be derived automatically from the source code of the component and the interfaces used by the component. We have chosen a left-to-right evaluation order according to semantics of Java or C#. If the evaluation order of the regarded component is implementation-dependent one has to choose here the order used by a compiler.

Remark: As in [27], we can encode reference parameters in our component abstraction too, to regard even recursive call-backs. Also resolving the reference parameters to all possible dynamically chosen services is possible and equal to the mentioned earlier work. Because of the lack of space we do not describe these calculations here.

For technical reasons we add a new start rule $I_S \xrightarrow{\lambda} p_i$, where p_i is the first program point to be executed.

After this construction we get a Process Algebra, but it contains all possible remote method calls, so we have to eliminate every action which is not included in the protocol P_i of the component C_i that is checked. For this purpose we use the following mapping function Φ_i:

$$\Phi_i : \Sigma \to \Sigma_{C_i} \text{ defined by } \Phi_i(x) = \begin{cases} x & \text{if } x \in \Sigma_{C_i} \\ \lambda & \text{otherwise} \end{cases}$$

Where Σ_{C_i} containts all remote methods of component C_i. Thus every translation rule using an action x which is not part of the components protocol alphabet will be replaced by the same rule which uses λ as action. Now we have a representation $\Pi_S^i = (Q_S, \Phi_i(\Sigma), I_S, \to_S, F_S)$ of the component-based system S according to the component C_i, it is valid $L_{\Pi_S^i} \subseteq L_{P_i}$. Π_S^i is used to create the Combined Abstraction in the next section.

In Figure 4 the example[6] – mentioned before – has been extended by labelled program points. For better understanding, we chose these labels unique over all components. In Figure 5 we show the abstraction Π_S^B of the example component-based system according to the protocol P_B of the component B. Because our example has a main component, we can use the first directive to create a start rule of S, hence we create one extra start rule only.

4 Combined Abstraction

To verify the component-based system abstraction with respect to a component C_i, we have to check, if $L_{\Pi_S^i} \subseteq L_{P_i}$, where L_{P_i} is the regular language described by the components protocol P_{C_i}, and $L_{\Pi_S^i}$ is the language over the actions in Π_S^i, specifying a superset of the use of C_i. In order to check $L_{\Pi_S^i} \subseteq L_{P_i}$ it is usual

[6] Because of the lack of space the example has no reference parameters nor dynamically chosen services. All components are hard coded.

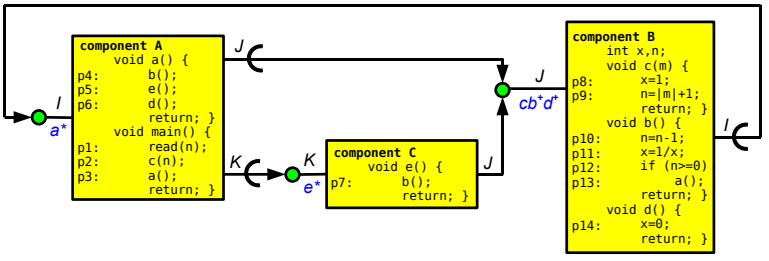


Fig. 4. Example. System with labelled program points.

$$I_S \xrightarrow{\lambda}_S p_1 \qquad p_4 \xrightarrow{b}_S p_{10}.p_5 \qquad p_8 \xrightarrow{\lambda}_S p_9 \qquad p_{12} \xrightarrow{\lambda}_S p_{13}$$

$$p_1 \xrightarrow{\lambda}_S p_2 \qquad p_5 \xrightarrow{\lambda}_S p_7||p_6 \qquad p_9 \xrightarrow{\lambda}_S \varepsilon \qquad p_{12} \xrightarrow{\lambda}_S \varepsilon$$

$$p_2 \xrightarrow{c}_S p_8.p_3 \qquad p_6 \xrightarrow{d}_S p_{14} \qquad p_{10} \xrightarrow{\lambda}_S p_{11} \qquad p_{13} \xrightarrow{\lambda}_S p_4$$

$$p_3 \xrightarrow{\lambda}_S p_4 \qquad p_7 \xrightarrow{b}_S p_{10} \qquad p_{11} \xrightarrow{\lambda}_S p_{12} \qquad p_{14} \xrightarrow{\lambda}_S \varepsilon$$

Fig. 5. Rewrite rules of Abstraction Π_S^B of the example component-based system according to the protocol of the component B

to check the equivalent problem $L_{\Pi_S^i} \cap \overline{L}_{P_i} = \emptyset$. Unfortunately this question is undecidable.

Theorem 1 (Undecidability of Protocol Checking Problem). *It is undecidable if $L_\Pi \subseteq L_P$ where Π is a $(1, G)$-PRS and L_P is regular.*

Proof (Sketch). As we know from model checkers (e. g. SPIN, MOPED), for each propositional LTL-formula ϕ, a FSM P can efficiently be constructed s. t. $L(\phi) = L_P$, where $L(\phi)$ is the set of action sequences specified by ϕ. Thus if the protocol checking problem would be decidable, we could also decide LTL-formula model checking for $(1, G)$-PRS. Contradiction, because LTL is undecidable in $(1, G)$-PRS. [5] □

We therefore construct a $(1, G)$-PRS K which describes a language $L_\approx$, where $L_{\Pi_S^i} \cap \overline{L}_{P_i} \subseteq L_\approx$. We call K *Combined Abstraction*. Thus if $L_\approx = \emptyset$, we know that $L_{\Pi_S^i} \subseteq L_{P_i}$. However, there might be a sequence $w \in L_\approx$ s. t. $w \notin L_{\Pi_S^i} \cap \overline{L}_{P_i}$. We call these sequences spurious false negatives.

In the following, we present the construction of the Combined Abstraction.

Roughly spoken the Combined Abstraction encodes in one model K the parallel execution paths of the abstraction of our system Π_S^i and the execution paths (which are forbidden by the regarded protocol P_i of C_i) formulated as finite state machine $\overline{P}_i$.

A combination of a protocol (FSM) A and an abstraction (PA) Π_S is to our knowledge not defined yet. The Combined Abstraction $K = (Q_K, \Sigma_K, I_K, \rightarrow_K, F_K)$ is defined as follows:

$Q_K = Q_A \times Q_S \times Q_A$	is a finite set of processes,
Σ_K	is a finite set of atomic actions,
$I_K \in Q_K$	is a start process,
$\rightarrow_K \subseteq Q_K \times \Sigma \times Q_K$	is a finite set of transition rules,
$F_K \subseteq Q_K$	is a finite set of final processes.

In accordance with [13] the processes $(q_i, q_j, q_k) \in Q_K$ encodes, that the FSM A is in the state q_i while the PA S has the process q_j created. The aim of (q_i, q_j, q_k) is, that $q_i \stackrel{x}{\Rightarrow}_A q_k$ while $q_j \stackrel{x}{\Rightarrow} \varepsilon$ can be performed.

The transition rules of the Combined Abstraction K have the same form and semantic as transition rules of PRS.

The construction of the other transition rules follows the directives shown in Figure 6 (described below) and is a generalization of the standard construction of the intersection of a finite state machine and a pushdown automaton in [13].

For technical reasons we also have to introduce the following two sets of rules:

$$R_S = \{I_K \stackrel{\lambda}{\rightarrow} (I_A, I_S, q_{F_A}) : q_{F_A} \in F_A\}$$
$$R_E = \{(q_A, q_P, q_A) \stackrel{\lambda}{\rightarrow} \varepsilon : q_A \in Q_A, q_P \in F_P\}$$

The chain transition rules R_{1C} and the set of sequential transition rules R_{1S} are handled similar to creating an intersection of pushdown automata and finite state machines in [13] (cf. Figure 6). The transition rules with a parallel operator R_{1P} are constructed as shown in directive r1p. As we see, a transition rule in the Combined Abstraction is similar to a transition rule in $\rightarrow_P$.

If one of the parallel threads in K accepts a $a \in \Sigma$ the protocol state in the other to p parallel threads should change to the same protocol states. With the transition rules in R_0, it is possible to implement these state changes.

The set of transition rules $\rightarrow_K$ is formed by uniting the sets R_S, R_E, R_{1C}, R_{1S}, R_{1P} and R_0.

After constructing the transition rules of the Combined Abstraction, we receive a rewrite system K in the syntax of PRS (we can also call K interleaving PRS). Now every possible interleaving sequence of the actions contained in the protocol is represented by at least one path in the Combined Abstraction K.

Theorem 2 (Correctness of construction of Combined Abstraction). *The construction K results in a representation, s. t. $L_{P_i} \cap L_{\Pi_S^i} \subseteq L_K$.*

Proof (Idea). Counterexamples constructable only by sequential rules are gathered by using the rewrite rules of R_{1C} and R_{1S}.

If a counterexample is conductable while using parallel rules, we have to look at rules of the form $(p_1||p_2).p_3$. Using the construction directive r1p the parallel traces are calculated independently.

If a rule out of R_{1P} is applied to p_1 thus this trace reaches a new state x in the protocol automaton. However in p_2 the protocol automaton has still the old

$$R_{1C} = \{(s,r,t) \xrightarrow{a}_K (s',r',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r') \wedge (a = \lambda)\} \quad \text{(r1c)}$$

$$R_{1S} = \{(s,r,t) \xrightarrow{a}_K (s',r',s'').(s'',r'',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r'.r'') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r'.r'') \wedge (a = \lambda)\} \quad \text{(r1s)}$$

$$R_{1P} = \{(s,r,t) \xrightarrow{a}_K (s',r',t)||(s',r'',t) \quad :(s \xrightarrow{a}_A s') \wedge (r \xrightarrow{a}_S r'||r'') \vee (s = s') \wedge (r \xrightarrow{\lambda}_S r'||r'') \wedge (a = \lambda)\} \quad \text{(r1p)}$$

$$R_0 = \{(s,r,t) \xrightarrow{\lambda}_K (s',r,t) \quad :(s \xrightarrow{a}_A s')\} \quad \text{(r0)}$$

$$\text{with } s,s',s'',t \in Q_A;\ r,r',r'' \in Q_P;\ a \in \Sigma_A \cup \{\lambda\};$$
$$(s,r,t),(s',r',t),(s',r'',t),(s',r',s''),(s'',r'',t) \in Q_K$$

Fig. 6. Directives for construction of transition rules of a Combined Abstractions K

state. With a rule from R_0, it is possible that p_2 reaches x, too, while using a rule out of R_0.

As we can easily see, using the explained construction, we create false negatives, too. These will be described at the end of the next chapter.

5 Performing Protocol Checking

Now, we want to verify if there exists a path from the start process I_K to the empty process ε. This is a reachability problem.

As seen before we create $\overline{A} = (\Sigma_A, Q_A, \delta_{\overline{A}}, q_A, F_A)$, where $L(\overline{A}) = \overline{L}_A = \Sigma^* \setminus L(A)$. $\overline{A}$ contains all possible traces to the final states $q_{F_A} \in F_A$ which are not part of the protocol A, we want to verify. Using $\overline{A}$ and P, the Combined Abstraction K will be created as described in the previous section. After this construction every existing path $I_K \overset{*}{\Rightarrow} \varepsilon$ is a candidate for a counterexample, because ε encodes the error process. Creating the counterexamples is possible using the logic EF, which is decidable in the class of $(1,G)$-PRS. [17]

We get a sequence of actions s as counterexample. Because we named the process constants of the system abstraction as program points, we are able to point out each program point of the regarded component, where a possible protocol violation can appear.

If we look at the Combined Abstractions in Figure 5 we can easily see that there is no protocol violation in component A and C. But there is a protocol violation in B. In Figure 7 a trace constructed by the reachability algorithms is shown. For the lack of space we reduce the language $\overline{L}_B$ to the one given in Figure 7 and show only a trace which results in the counterexample *cbdb*. This counterexample can appear only, if the method call of e is asynchronously performed. If we look at the original source code of the component B in Figure 4 we can see, that this sequence of actions really results in a division by zero, which is a non expected behaviour.

FSM $P_{B'}$, that describes a subset of the inverted protocol $P_{\overline{B}}$ of component B. $P_{B'} = (\{I_A, x_2, x_3, x_4, x_F\}, \{b, c, d\}, I_A, \{I_A \xrightarrow{c} x_2, x_2 \xrightarrow{b} x_3, x_3 \xrightarrow{d} x_4, x_4 \xrightarrow{b} x_F\}, \{x_F\})$ We can see that $L_{P_{B'}} \subseteq \overline{L_B}$.

Trace of K constructing the protocol violation *cbdb* in component B:

$\underline{(I_A, I_S, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(I_A, q_1, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(I_A, p_2, x_F)}$

$\stackrel{c}{\Rightarrow}$ $\underline{(x_2, p_8, x_2)}.(x_2, p_3, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_2, p_9, x_2)}.(x_2, p_3, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_2, \varepsilon, x_2)}.(x_2, p_3, x_F)$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_2, p_3, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_2, p_4, x_F)}$ $\stackrel{b}{\Rightarrow}$ $\underline{(x_3, p_{10}, x_3)}.(x_3, p_5, x_F)$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_3, p_{11}, x_3)}.(x_3, p_5, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_3, p_{12}, x_3)}.(x_3, p_5, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_3, \varepsilon, x_3)}.(x_3, p_5, x_F)$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_3, p_5, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $(x_3, p_7, x_F)||\underline{(x_3, p_6, x_F)}$ $\stackrel{d}{\Rightarrow}$ $(x_3, p_7, x_F)||\underline{(x_4, p_{14}, x_F)}$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_3, p_7, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_4, p_7, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{b}{\Rightarrow}$ $\underline{(x_F, p_{10}, x_F)}||(x_4, \varepsilon, x_F)$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_F, p_{11}, x_F)}||(x_4, \varepsilon, x_F)$ $\stackrel{\lambda}{\Rightarrow}$ $(x_F, p_{12}, x_F)||\underline{(x_4, \varepsilon, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $(x_F, p_{12}, x_F)||\underline{(x_F, \varepsilon, x_F)}$

$\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_F, p_{12}, x_F)}$ $\stackrel{\lambda}{\Rightarrow}$ $\underline{(x_F, \varepsilon, x_F)}$ $\square$

Fig. 7. Example. Trace that will be constructed by the reachability algorithm, it results in the counterexample *cbdb* for the protocol of B in Figure 4. For better understanding the processes of K rewritten in the considered step are underlined.

False Negatives There are two causes of false negatives:

- real false negatives: Because the component abstractions are created without any data flow or control flow analysis, it is possible that a trace will be contained in the component abstraction, which is not possible in the implemented component. Moreover e. g. a return value can route the control flow, thus if we have no access to the implementation of the other components of the component-based system, it is possible to create more false negatives. The constructable counterexample c in our example is such a false negative.
- spurious false negatives: Because we only construct an approximated intersection of the language described by the component-based system and the regarded protocol, it is possible to get false negatives.

If the component code is not available, it will only be possible to reduce the spurious false negatives.

6 Component Composablity

We now show how a kind of PRS can be individually computed for each component, and how these can be composed in order to obtain a PRS describing the use of the component whose protocol has to be checked.

As in reality not every component (e. g. Web Service) is accessible by a component developer. Thus it is necessary to define an abstraction for each component C, which is composable to an abstraction of the full component-based

system. We call the PRS of a single component C *stripped process rewrite system* $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C)$. Π_C is defined as follows:

Q_C	is a finite set of atomic processes,
Σ_C	is a finite set of atomic actions,
$\rightarrow_C \subseteq PEX(Q_C) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q_C)$	is a set of process rewrite rules,
R_C	is a finite set of required interfaces,
$P_C : S \mapsto Q_C$	is a mapping from the services to the first program point in the provided interfaces.

The foundation for creating $\rightarrow_C$ are the directives described in Section 3. We extend the considered directives by the following:

- If at a program point p_i a synchronous *remote* method call a is performed, we create rewrite rules $p_i \xrightarrow{a}_C q_{J,s} \,.\, p_k$, if a is an asynchronous *remote* method call we create rewrite rules $p_i \xrightarrow{a}_C q_{J,s} \,||\, p_k$, where $q_{J,s}$ specifies the interface J of the required service s, and $p_k \in next(p_i)$. Note if we do not know how the interface is implemented, we have to create both sets of rewrite rules to ensure, that we create a conservative abstraction.
- If the considered program point p_i is the first in a method implementing a provided service s, we will extend the mapping P_C with $s \mapsto p_i$.

The set R_C contains all interfaces $q_{J,s}$ where s is a service of a required interface J. P_C mappes the set of services S (provided by the interfaces of C) to the initial process of the provided interface.

In the case considered in this paper, we have to look at stripped Process Algebras only. You can see the abstractions of the example components in Figure 8.

After having constructed abstractions for each component, we have to combine each component C_i (respectively Π_{C_i}) to the component-based system S (respectively Π_S) we want to verify. In the first phase, this can easily be constructed by uniting the relevant sets. Thus we define the abstraction $\Pi_S = (Q_S, \Sigma_S, I_S, \rightarrow_S, F_S)$ of S as follows:

$Q_S = \{I_S\} \uplus \bigcup_{C_i} Q_{C_i}$	is a finite set of processes,
$\Sigma_S = \{\lambda\} \cup \bigcup_{C_i} \Sigma_{C_i}$	is a finite set of actions,
$I_S \in Q_S$	is a new start process,
$\rightarrow_S = \bigcup_{C_i} \rightarrow_{C_i} \cup Init$	is a finite set of transition rules,
$F_S = \bigcup_{C_i} F_{C_i} \subseteq Q_S$	is a finite set of final processes.

$$\rightarrow_A = \{p_1 \xrightarrow{\lambda} p_2,\ p_2 \xrightarrow{c} q_{J,c} \,.\, p_3,\ p_3 \xrightarrow{a} p_4,\ p_4 \xrightarrow{b} q_{J,b} \,.\, p_5,\ p_5 \xrightarrow{e} q_{K,e} \,||\, p_6,\ p_6 \xrightarrow{d} q_{J,d}\}$$

$$\rightarrow_C = \{p_7 \xrightarrow{b} q_{J,b}\}$$

$$\rightarrow_B = \{p_8 \xrightarrow{\lambda} p_9, p_9 \xrightarrow{\lambda} \varepsilon, p_{10} \xrightarrow{\lambda} p_{11}, p_{11} \xrightarrow{\lambda} p_{12}, p_{12} \xrightarrow{\lambda} p_{13}, p_{12} \xrightarrow{\lambda} \varepsilon, p_{13} \xrightarrow{a} q_{I,a}, p_{14} \xrightarrow{\lambda} \varepsilon\}$$

$$P_A(q_{I,a}) \mapsto p_4,\ P_C(q_{K,e}) \mapsto p_7,\ P_B(q_{J,c}) \mapsto p_8,\ P_B(q_{J,b}) \mapsto p_{10},\ P_B(q_{J,d}) \mapsto p_{14}$$

Fig. 8. Example. Transitions rules of the abstractions Π_A, Π_B, Π_C and mapping of provided interfaces of components A, B, and C in Figure 4.

To ensure that every initial program point I_i of a component C_i is reachable from the new start process I_S, we add the following transition rules to $\rightarrow_S$:

$$(I_S \xrightarrow{\lambda}_S I_i) \quad \text{iff } C_i \text{ is the main component/the client of } S$$

$$(I_S \xrightarrow{\lambda}_S I_i) \quad \text{iff any component } C_i \text{ of } S \text{ can start}$$

$$(I_S \xrightarrow{\lambda}_S I_0||I_1||...||I_n) \quad \text{iff each component } C_i \text{ of } S \text{ can process independently}$$

In the second phase, every used interface has to be resolved to a process, that specifies the first program point of the called interface implementation. Thus we have to resolve all interfaces $q_{J,s} \in \bigcup_{C_i} R_{C_i}$ using the mapping function P_{C_i} of the component implementing the interface J.

As in Section 3 we still have to create new start rules and eliminate every action, which is not included in the protocol P_{C_i} of the considered component C_i using the mapping function Φ_i.

In Figure 8 the abstractions of the components A, B and C are shown. As can easily be seen the abstraction S of the full system in our example is easy to build, unifying the sets and resolving the provided interfaces as mentioned above. It results in the PRS shown in Figure 5.

7 Related Work

Many works on static protocol-checking of components consider local protocol checking on FSMs. The same approach can also be applied to check protocols of objects in object-oriented systems. The idea of static type checking by using FSMs goes back to Nierstrasz [18]. Their approach uses regular languages to model the dynamic behaviour of objects, which is less powerful than context-free grammars (CFG). In the work of Yellin and Strom [25] also only regular representations of the components are used, but they describe a protocol by send and receive synchronous method calls, and generate adapters if the protocol check fails. These approaches cannot handle recursive call-backs. [15] considers object-life cycles for the dynamic exchange of implementations of classes and methods using a combination of the bridge/strategy pattern. It also based on FSMs. The approach comprises dynamic as well as static conformance checking. Tenzer and Stevens [23] investigate approaches for checking object-life cycles.

They assume that object-life cycles of UML-classes are described using UML state-charts and that for each method of a client, there is a FSM that describes the calling sequence from that method. In order to deal with recursion, Tenzer and Stevens add a rather complicated recursion mechanism to FSMs. It is not clear whether this recursion mechanism is as powerful as pushdown automata and therefore could accept general context-free languages. All these works are for sequential systems. Schmidt et al. [12] propose an approach for protocol checking of concurrent component-based systems. Their approach is also FSM-based and unable to deal with recursive call-backs.

Even modeling the use of a component with context-free languages may abstract too much from the real behaviour. Other approaches [9, 20] therefore use dynamic protocol-checking. Dynamic protocol checking does not exclude protocol faults as static protocol checking does. On the other hand, they identify bugs at the right place. In particular, dynamic adapters might support avoiding protocol faults whenever possible.

An alternative approach for investigation of protocol conformance is the use of process algebras such as CSP, cf. e. g. [1]. These approaches are more powerful than FSMs and context-free grammars. However, mechanized checking requires some restrictions on the specification language. For example, [1] uses a subset of CSP that allows only the specification of finite processes. At the end the conformance checking reduces to checking FSMs similar to [12].

FSMs are also used for checking Liskov's substitution principle for subtyping in object-oriented systems based on class protocols. Reussner [21] generalizes on the idea of Nierstrasz and adds counters and conditions over counters to the regular types to decide, whether Liskov's substitution principle is satisfied. Freudig et al. [11] use sub-classes of CFGs for describing protocols and checking Liskov's substitution principle. They need subclasses of CFGs because the subset-problem on general context-free languages is algorithmically undecidable. They do not model calling sequences stemming from a method which is required for checking whether the use of an object of a certain class conforms to its protocol.

The work on model checking context-free processes and pushdown systems started with [6, 7]. The model checking of LTL-formulas can be done linear in the size of the system and cubic in the number of states [2, 3, 10]. However, these approaches would require that the complete system is available as a context-free process or as a pushdown system. The framework described in [4] contains among others an algorithm for checking whether $L(G) \subseteq L(A)$ for context-free grammars G and finite state machines A.

The approach in this paper is a generalization of [26, 27]. In these papers recursion is modeled by CFG, so only sequential behaviour is considered. It is demonstrated how the approach can be made compositional. Moreover recursive callbacks are respected, which is possible but not considered within our work. Like in our approach every components abstraction has to be known at the verification time, but in contrast to this work counterexamples can be created exactly, if a fault has been discovered.

Chaki et al. described in [8] a method to verify communicating recursive C programs. This problem seems to be similar to verification of component-based systems, although they considered synchronous method calls only. In contrast to our work they consider even the data manipulation and synchronization statements. The problem can be reduced to the intersection of – by C programs described – context free languages, which were calculated approximately by a CEGAR-loop. There are other works [14, 19, 24] which consider the verification of concurrent programs, but these reduce the problem with bounded context switching, which results in a bounded parallelism.

8 Summary and Conclusions

In this paper we discussed the automatic verification of components according to their protocol. This static verification can be used to find semantic errors, i. e. to verify defined non functional business rules.

To apply our check, we require static knowledge of the used components. But this abstraction can be part of the component description, so we do not need access to the components source code. As other works in this research area, we use FSM for describing component protocols.

In contrast to previous approaches, we are able to handle recursion and parallel behaviour in a local and global view without any restrictions using process rewrite systems to represent the behaviour of each component instead of finite state machines or context free grammars. The decidability of the reachability problem has been proven by Mayr. In order to circumvent the undecidability of the protocol checking problem, we define an approximated intersection of protocols and Process Algebras – the so called Combined Abstraction.

We implemented a two phase process to consider the component composition, where in the first phase the components were composed, like in the real system and in the second phase the required interfaces (and reference parameters) were resolved, so every information depending on the component-based system can be included in the system abstraction. Because of this process we are able to compose the abstractions of the components like the components in reality. Moreover our approach makes it possible to deal with components implemented in different programming languages, because the abstraction layer hides the implementing details.

The tool provides a counterexample if the protocol conformance check fails. So our approach is a model-checking approach. A counterexample is a word over all protocol actions, which are remote method calls. A calculated counterexample may not occur in the real system, because we create a conservative abstraction, hence false negatives may be delivered. But we are sure to find a counterexample if any exists.

At this stage of our work we only consider static verification, i. e., the abstractions of each component are known statically. CORBA, COM, .NET and EJBs also allow dynamic instances of components. It is subject to further work to handle this property. As demonstrated in [27], a points-to analysis might help to solve the problem.

Our approach is adaptable for object-oriented programming where the protocols are defined over the public interfaces. It will be part of future work to research if our approach is suitability for daily use in OOP.

We currently implement a framework which creates abstractions of components implemented in Python (finished) and C++ (in progress). Creating abstractions of Java components is planed. This framework is currently be used to verify our approach in industrial case studies. Early results show that our approach is applicable and can result to real (so far undiscovered) bugs.

False negatives may be reduced by integration of data and control flow analysis algorithms into the component abstraction process.

We thank Heinz W. Schmidt for pointing us to process rewrite systems.
We are grateful to OR Soft GmbH for providing us with industrial case studies.

References

1. Allen, R., Garlan, S.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
2. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 207–220. Springer, Heidelberg (2001)
3. Benedikt, M., Godefroid, P., Reps, T.: Model checking of unrestricted hierarchical state machines. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 652–666. Springer, Heidelberg (2001)
4. Bouajjani, A., Esparza, J., Finkel, A., Maler, O., Rossmanith, P., Willems, B., Wolper, P.: An efficient automata approach to some problems on context-free grammars. Information Processing Letters 74(5-6), 221–227 (2000)
5. Bouajjani, A., Habermehl, P.: Constrained properties, semilinear systems, and petri nets. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 481–497. Springer, Heidelberg (1996)
6. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992)
7. Burkart, O., Steffen, B.: Pushdown processes: Parallel composition and model checking. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 98–113. Springer, Heidelberg (1994)
8. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing c programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
9. Chambers, C.: Predicate classes. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 268–296. Springer, Heidelberg (1993)
10. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
11. Freudig, J., Löwe, W., Neumann, R., Trapp, M.: Subtyping of context-free classes. In: Proceedings 3rd White Object Oriented Nights (1998)
12. Schmidt, H.W., Krämer, B.J., Poernemo, I., Reussner, R.: Predictable component architectures using dependent finite state machines. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 310–324. Springer, Heidelberg (2004)

13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
14. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
15. Löwe, W., Neumann, R., Trapp, M., Zimmermann, W.: Robust dynamic exchange of implementation aspects. In: TOOLS 29 – Technology of Object-Oriented Languages and Systems, pp. 351–360. IEEE, Los Alamitos (1999)
16. Mayr, R.: Process rewrite systems. Information and Computation 156(1-2), 264–286 (2000)
17. Mayr, R.: Decidability of model checking with the temporal logic ef. Theor. Comput. Sci. 256(1-2), 31–62 (2001)
18. Nierstrasz, O.: Regular types for active objects. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, pp. 99–121. Prentice-Hall, Englewood Cliffs (1995)
19. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS, pp. 93–107 (2005)
20. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 83–94. ACM, New York (2002)
21. Reussner, R.H.: Counter-constraint finite state machines: A new model for resource-bounded component protocols. In: Grosky, W.I., Plášil, F. (eds.) SOFSEM 2002. LNCS, vol. 2540, pp. 20–40. Springer, Heidelberg (2002)
22. Schmidt, H.W., Krämer, B.J., Poernomo, I., Reussner, R.: Predictable component architectures using dependent finite state machines. In: Wirsing, M., Knapp, A., Balsamo, S. (eds.) RISSEF 2002. LNCS, vol. 2941, pp. 310–324. Springer, Heidelberg (2004)
23. Tenzer, J., Stevens, P.: Modelling recursive calls with uml state diagrams. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 135–149. Springer, Heidelberg (2003)
24. Torre, S.L., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
25. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19(2), 292–333 (1997)
26. Zimmermann, W., Schaarschmidt, M.: Model checking of client-component conformance. In: 2nd Nordic Conference on Web-Services. Mathematical Modelling in Physics, Engineering and Cognitive Sciences, vol. 008, pp. 63–74 (2003)
27. Zimmermann, W., Schaarschmidt, M.: Automatic checking of component protocols in component-based systems. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 1–17. Springer, Heidelberg (2006)

Structural Testing of Component-Based Systems

Daniel Sundmark, Jan Carlson, Sasikumar Punnekkat, and Andreas Ermedahl

MRTC, Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
daniel.sundmark@mdh.se

Abstract. Component based development of software systems needs to devise effective test management strategies in order fully achieve its perceived advantages of cost efficiency, flexibility, and quality in industrial contexts. In industrial systems with quality demands, while testing software, measures are employed to evaluate the thoroughness achieved by execution of a certain set of test cases. Typically, these measures are expressed in the form of *coverage* of different structural test criteria, e.g., statement coverage. However, such measures are traditionally applicable only on the lowest level of software integration (i.e., the component level). As components are assembled into subsystems and further into full systems, general measures of test thoroughness are no longer available. In this context, we formalize the added test effort and show to what extent the coverage of structural test criteria are maintained when components are integrated, in three representative component models. This enables focusing on testing the right aspects of the software at the right level of integration, and achieves cost reduction during testing — one of the most resource-consuming activities in software engineering.

1 Introduction

The component-based development paradigm has been quite successful in enterprise computing and is being explored as an attractive option in other domains with quality demands, e.g., embedded software systems. However, in order to gain wide acceptance in such domains, the component-based approach also needs to devise efficient testing strategies and test management approaches, since testing accounts for a lion's share of the development cost in such systems. Quality concerns also demand that the developer presents evidence of thoroughness of verification efforts performed. Structural coverage is a set of measures for evaluating the thoroughness of software testing, based on how exhaustively the tests exercise certain aspects of the structure of the software under test [1]. Test criteria based on structural coverage are well-defined for component-level testing [1,2], but most of these definitions are not generally applicable for testing performed post-integration, where issues of, e.g., multi-tasking and shared resources come into play.

Building a system out of well-tested components does not necessarily result in a well-tested system. However, during integration testing, it is possible to make use of the information available on what aspects of the software that have already

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 180–195, 2008.

been tested before integration. Ideally, during integration testing, testing should only focus on the correctness of the actual interaction between the integrated components.

The contribution of this paper is twofold. First, we describe the added test effort required by component integration, by introducing the concept of *compositionally introduced test items*. Second, we describe the impact of this concept for a number of common structural test criteria and three representative component models. We also outline how this concept can be used in order to maintain the quality assurance achieved by structural test coverage at an arbitrary level of integration in multi-level hierarchical development of component-based software.

The main motivation for this work is that it facilitates a less ad-hoc way of determining the adequacy of integration- and system-level testing in addition to the functional testing traditionally used at this level, while also clearly separating the test items that could be tested at component-level from those that need to be tested post-integration.

2 Background

Testing is the primary means for verification used in the software industry, and methods and strategies for testing come in many different shapes. Depending on the type of software system to be developed or maintained, hypotheses regarding the types of bugs suspected in the software, and many other aspects, the testing approach will be different. There is, however, a common ground for reasoning about test techniques.

2.1 Software Testing and Coverage

A *test criterion* is a specification for evaluating the test adequacy given by a certain set of test cases. Test criteria are defined in terms of *test items*, the "atoms" of test criteria. A test criterion is generally formulated such that test adequacy (with respect to that criterion) is attained when all test items are exercised during testing. For example, for the statement coverage criterion, statements are the test items. Test items are also called coverage items. *Coverage* is a generic term for a set of metrics used for expressing test adequacy (i.e., the thoroughness of testing or determining when to stop testing with respect to a specific test criterion [1]). A coverage measure is generally expressed as a real number between 0 and 1, describing the ratio between the number of test items exercised during testing and the overall number of test items. Hence, a statement coverage of 1 implies that all statements in the software under test are exercised. Rules for when to stop testing can be formulated in terms of coverage. For example, a statement coverage of 0.5 (indicating that half of the statements in the software are exercised) may be a valid, if not very practical, stopping rule.

Test criteria may be *structural* or *functional*, where structural test criteria are based on the actual software implementation, or on abstract representations of the software implementation (e.g., control flow graphs). As *structural* test criteria

are strictly based on the actual software implementation and different inherent aspects of its structure, these are possible to define formally. Examples of structural test criteria include exercising of all instructions, all execution paths, or all variable definition-use paths in the software. It should be noted here that a full coverage (i.e., a coverage of 1) is not generally achievable for structural test criteria. This is due to the fact that the control flow graph representations used to define these criteria typically contain infeasible paths or statements (i.e., paths or statements that are not actually exercisable when executing the code), and the exact determination of feasible paths and statements is undecidable [3,4].

On the other hand, test case selection based on *functional* test criteria is, in the general case, ad-hoc in the sense that it depends on the quality, expressiveness, and the level of abstraction of the specification. Basically, a more detailed and thorough specification will result in a more ambitious and thorough functional test suite. Examples of functional test techniques are boundary value testing and equivalence class partitioning testing based on the software specification [5].

In the traditional view of the software engineering process, verification testing is performed at different levels. Throughout the literature, many such levels are discussed, but the most commonly reappearing levels of verification testing are *component, integration* and *system testing* [5,6], see Fig. 1. **Component testing** (also known as unit testing) is performed at the "lowest" level of software development, where the smallest units of software are tested in isolation. Such units may be functions, classes or components. **Integration testing** can be performed whenever two or more components are assembled into a system or a subsystem. Specifically, integration testing focuses on finding failures that are caused by interaction between the different components in the (sub)system. **System testing** focuses on the failures that arise at the highest level of integration [7], where all parts of the system are incorporated and executed on the intended target hardware(s). The execution of a system-level test case is considered correct if its output and behaviour complies with what is stated in the system specification.

Generally, the lower the level of testing, the more likely that both structural and functional criteria will be considered. In traditional system-level testing, only functional criteria are considered [6], and integration testing poses several problems for structural criteria, e.g., definition of control- and data-flow structures over component boundaries. Hence, as we recognize that structural and functional techniques complement each other by focusing on different aspects of the same software, this paper aims at facilitating the additional use of structural criteria on higher levels of integration. It is our firm belief that, compared to the traditional testing performed at the higher levels of integration, a combination of structural and functional testing will provide a more thorough software verification.

2.2 CBSE and Structural Software Testing

During component integration, the control- and data flow may be modified or compromised, and coverage based on these concepts may be invalidated.

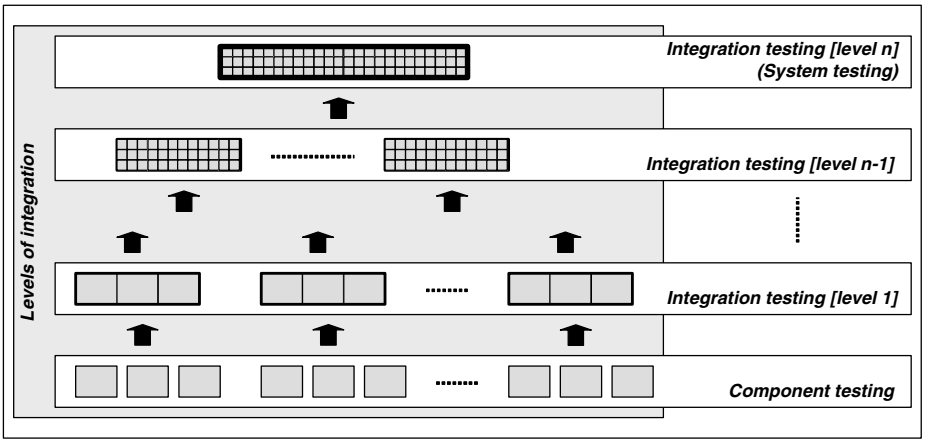


Fig. 1. Testing/integration levels in the software development process

Considering the system-level counterparts of control- and data flow, there are no widely accepted general definitions of these concepts. Given unrestricted component interaction, the control and data flow of a component assembly may exhibit an unmanageable complexity. However, in practice the interaction between the software parts in an integration is restricted by several factors, e.g., the run-time system, and the architectural style used. To prevent an overwhelming complexity, it is often desirable to adopt some level of component or unit isolation.

In Component-Based Software Engineering (CBSE), software applications are built by composing software components into component assemblies [8,9]. The main idea is that, by building systems out of well-tested components, an increase in the predictability of the behaviour of the software could be gained; provided that experience from component testing is taken into account.

Components are independent software units that interact via well-defined interfaces. According to the basic CBSE principles, there should be no hidden dependencies between components, except for those explicitly represented in the component interfaces. This facilitates reuse, allowing a component to be replaced without affecting the other components in the system.

In the context of structural testing at higher levels of integration, the additional information provided by component interfaces could be exploited while reasoning about the control and data flow in an assembly, and, in a later stage, when generating test cases. Thus, with a strong notion of component interface, the use of CBSE gives benefits during integration level testing, compared to traditional approaches where the corresponding information must be derived from low level code.

In general, system composition out of a set of components is guided by the architectural style chosen for the system. According to Shaw and Garlan [10], examples of such architectural styles include:

– **Dataflow systems**, which include systems based on *pipes and filters*, where components act as filters of data, and the interconnections between

components act as data pipelines. This type of system typically manipulates sequential streams of data passed through components by pipelines.
- **Call-and-return systems**, e.g., object oriented systems, where the components (objects) of the system interact through the use of inter-component method calls.
- **Independent components**, e.g., event-based systems, uses an approach where the invocation of components and component methods are not triggered by explicit calls from other components, or on the explicitly stated data flow through the system, but rather on the occurrences of internal or external events.

In the following section, we will consider instances of these styles in order to see how component composition according to each instance affects component interaction, and the structural testing thereof. It should be noted that Shaw and Garlan [10] also mention **virtual machines** and **data centered systems (repositories)** as examples of architectural styles, but we will not consider them in this paper.

3 Structural Testing of Component-Based Systems

In this section we describe what is required to achieve structural test coverage for component assemblies, including whole component-based systems. In doing this, we aim at a software development process where the knowledge of what has already been tested, and the effects of component interaction, are used in order to perform a more conscious, non-ad-hoc integration testing. Ideally, we consider a process as the one described in Fig. 2, where the composition of a set of components (1) is followed by an analysis determining if any further testing is required to maintain the desired coverage (2). If such testing is required, we generate (3) a set of test cases required to achieve the desired coverage, whereafter test execution (4) and evaluation (5) follows, leading to an integrated component assembly (or system) with the desired coverage maintained (6).

In doing this, our primary goal is to find a set of test cases that are required in order to safely cover the aspects of the software added by component integration. The secondary goal would be to find the minimal set of test cases that fulfils this criterion. Unfortunately, since we are generally unable to determine exactly which test items are feasible (i.e., executable on the level of integration where the testing is performed), any analysis performed to safely determine feasible test items will be over-approximative, and potentially find false positives [11]. Once again, it should be noted that this problem is not unique to the higher levels of testing we consider in this paper (even though it is likely to be more severe), since not all items are feasible on component-level, and the exact determination of feasible test items, even on component-level, is provably incalculable [3,4].

Also, note that some test items that are feasible when testing a component in isolation might be made infeasible by system integration [12]. For example, a definition of a shared variable in one component may influence the flow of control

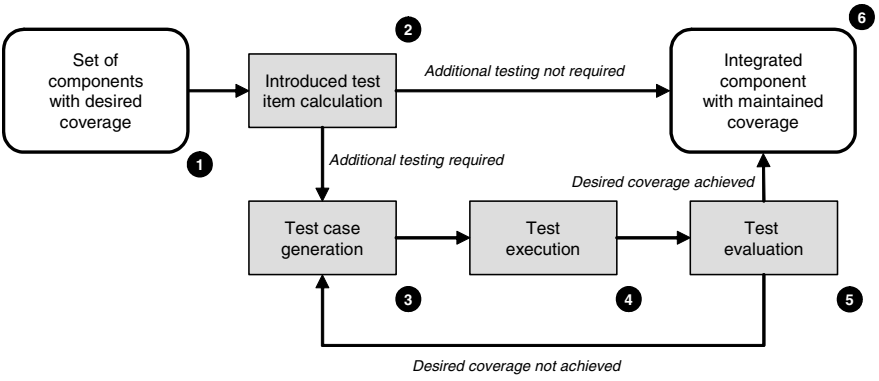


Fig. 2. An outline of the envisioned process

and make paths considered feasible in other components infeasible [13,14,15]. This should be considered when performing testing of multi-tasking (and parallel) systems.

3.1 Impact of Architectural Style

The architecture of the software under test will affect how test criteria will be affected by component composition, since it, to a large extent, determines the means of inter-component communication. Further, the choice of architectural style to different degrees limits the component interaction, e.g., in terms of temporal perturbation.

Although the proposed approach is not limited to a particular component model or architectural style, it is exemplified by representatives of three different architectural styles:

- **CM1:** As an example of the dataflow style, we consider a component model (see Fig. 3a), similar to that of SaveCCM [16] or PECOS [17]. Contrasting, e.g., the UNIX pipes-and-filters architecture where the filters execute concurrently, we consider an interleaved model where components execute non-preemptively. When activated, a component consumes one set of input data and then executes to completion.
- **CM2:** Representing call-and-return systems, we consider a more traditional model (see Fig. 3b), where components are invoked by method calls. Examples of such models include Sun's JavaBeans [18], Microsoft's COM [19], and the Koala component model [20].
- **CM3:** As a third example, covering the architectural style of independent components, we consider a component-based preemptive real-time system where components correspond to individual tasks executing on an underlying real-time operating system (see Fig. 3c). Here, we consider components that are strictly periodic, inter-arrival and assume that execution is

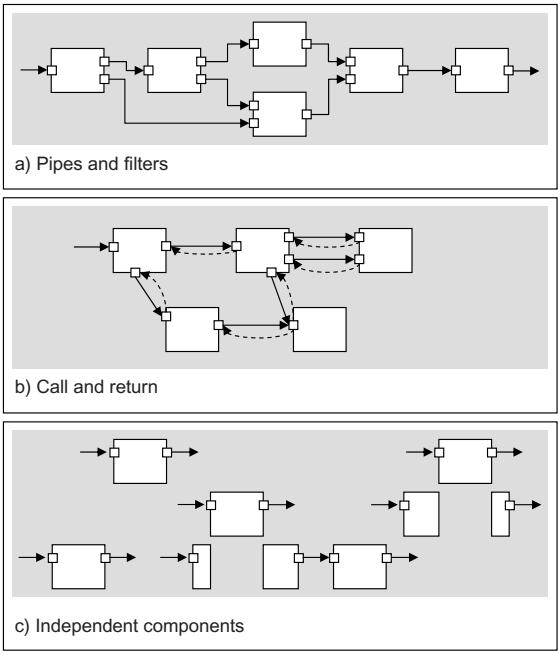


Fig. 3. Architectural Styles

controlled by a system-level scheduler that distributes computation among the components, based on, e.g., priority levels. Moreover, components are preemptive, meaning that the scheduler is allowed to interrupt a component during its execution, should a component of higher priority level become available for execution. Examples of component models of this type include Rubus [21] and Autocomp [22].

It should be noted that the components in the third example are only independent in the sense that any transfer of control between components is initiated by the system-level scheduler, and not from within a component. In general, the components are not functionally independent, since they may communicate via shared memory. Also, since tasks compete for the same computational resources, they are clearly not independent with respect to timing.

It should also be noted that different architectural styles are sometimes adopted at different integration levels of the same system, as in COMDES [23] or ProCom [24]. For example, a large system might be built from a few independent components, each of which in turn can be further decomposed into smaller components interacting in a pipes and filter fashion. The aim of our method is that the choice of architectural style at lower levels of integration should be transparent when determining the test coverage at a specific level of integration.

3.2 Compositionally Introduced Test Items

In this section, we describe what needs to be covered by testing on a certain level of component integration in order to maintain the coverage held by the components to be integrated. This is done by defining the concept of *compositionally introduced test items*. Simplified, these are the test items, given a certain test criterion, that only exist on the current level of integration and above. These additional test items particularly have two sources: The interaction between the integrated components, and extra code added merely for the integration of the components. Starting with the latter, during the development of systems based on components there could be "extra" code involved, depending on the architectural style followed. Particularly, this extra code could comprise of different categories such as:

- operating system code (e.g., driver routines, task switch routines and other system functions); and
- glue code, written or automatically generated to connect components, and for configuration and initialization.

For simplicity, we will consider the extra code as completely untested in the remainder of this paper.

Given the existence of previously non-covered additional code, one will have to pay special attention to ensure test coverage of this code during the higher level integration. Also, even if the additional code is already covered, it might give rise to additional test items for some test criteria and architectural styles, caused by its interaction with the components in the assembly.

Before formally defining the concept of compositionally introduced test items, some notation needs to be introduced. We consider an assembly $\mathcal{A}$ consisting of components $C_1, \ldots, C_\alpha$, composed in accordance with some component model (i.e., α denotes the number of components in the assembly). To simplify the presentation, we denote by C_0 all the extra code of the assembly. In all other aspects, C_0 is not to be considered as a component. Moreover, given a particular test criterion TC, the test items of C_i and $\mathcal{A}$ are denoted $TI_{C_i}^{TC}$ and $TI_{\mathcal{A}}^{TC}$, respectively.

Definition 1. *The set of* compositionally introduced test items *of a test criterion TC and an assembly $\mathcal{A}$, is defined as follows:*

$$CI_{\mathcal{A}}^{TC} = TI_{\mathcal{A}}^{TC} \setminus \bigcup_{i=1}^{\alpha} TI_{C_i}^{TC}$$

Thus, $CI_{\mathcal{A}}^{TC}$ denote the test items of $\mathcal{A}$ that are not present when the constituent components $C_1, \ldots, C_\alpha$ are considered in isolation. Since most structural test criteria are defined in terms of control flow paths or control flow graphs, these concepts must be defined on an assembly level. For this, we denote by S_i the statements of component C_i.

Definition 2. *The statements of an assembly $\mathcal{A}$ are denoted $S_{\mathcal{A}}$, and defined as*

$$S_{\mathcal{A}} = \bigcup_{i=0}^{\alpha} S_i.$$

Definition 3. *The* control flow graph *of an assembly $\mathcal{A}$ is a directed graph where the nodes are the statements in $S_{\mathcal{A}}$, and a directed edge $\langle s_1, s_2 \rangle$ represents a possible control flow from statement s_1 to s_2.*

Definition 4. *A* control flow path *of an assembly $\mathcal{A}$ is a finite path in the control flow graph of $\mathcal{A}$.*

The control flow graph of an assembly can be very complex, particularly for component models where transfer of control from one component to another is not related to explicit constructs in the component code (as, for example, in **CM3** where the scheduler may preempt a component at any point). Further, we note that each edge $\langle s_k, s_{k+1} \rangle$ in the control flow path of an assembly is either

1. part of the local control flow of a component C_i (i.e., $\langle s_k, s_{k+1} \rangle$ is in the control flow path of C_i and $1 \leq i \leq \alpha$);
2. part of the control flow of the additional code C_0; or
3. a transfer of control between two components, or between a component and additional code (i.e., $s_k \in S_i$ and $s_{k+1} \in S_j$, with $i \neq j$, $0 \leq i \leq \alpha$ and $0 \leq j \leq \alpha$).

Categories 2 and 3 are of particular interest, since they are the ones introduced as a result of the composition. The third category is the main source of complexity, and this is also where the choice of component model has the biggest impact.

For the usage proposed here, i.e., to measure test coverage and guide test case generation, it is preferrable if the transfer of control between components can be determined, or approximated, from the component interfaces. The impact of the component model on the control flow graph is further discussed in Section 4.2.

4 Test Criteria

This section lists a number of structural test criteria, and, for each criterion, defines its set of compositionally introduced test items. In the cases where the choice of component model (**CM1**–**CM3**) affects this set, this effect is described for each different choice. The structural test criteria we investigate in this section with respect to the set of compositionally introduced test items are:

- **Statement coverage** is chosen since it is the most basic, and in our experience, the most widely recognized structural test criterion, even to the point that it is sometimes used synonymously with *code coverage* or *coverage* in general.
- **Branch coverage** is chosen since it has a large similarity to statement coverage, but still differs with respect to compositionally introduced test items.

- **Path coverage** is chosen based on the fact that it, in its basic form, requires the execution of each path through the system. As such, it is one of the more exhaustive test criteria available.
- **Modified condition/decision coverage (MC/DC)** is chosen since it is required as a part of the de-facto standard process in software development of some safety critical software, e.g., avionics software [25].
- **All uses coverage** is chosen since it, when considering shared variables in multi-tasking environments (a typical integration or system-level testing concern), targets failures related to race conditions and similar interleaving problems [11,26].

In our work, we make use of definitions of these criteria from [1,27] in defining the compositionally introduced test items for each of the three representative component models.

4.1 Statement Coverage Criterion

According to Zhu et al. [1], "*a set P of execution paths satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path $p \in P$ such that node n is on the path p*".

For **CM1**, **CM2**, and **CM3**, the set of compositionally introduced test items of the statement coverage criterion are just the statements of the additional code, i.e., $CI_{\mathcal{A}}^{\mathrm{SC}} = S_0$. This follows directly from Definitions 1, 2 and 4, since for each statement $s \in S_{\mathcal{A}}$ in the control flow graph of $\mathcal{A}$ we have $s \in S_i$, $0 \leq i \leq \alpha$. Thus, either $s \in S_0$ or s is among the test items of component C_i, in which case it should not be included in $CI_{\mathcal{A}}^{\mathrm{SC}}$.

4.2 Branch Coverage Criterion

Again, according to Zhu et al. [1], "*a set P of execution paths satisfies the branch coverage criterion if and only if for all edges e in the flow graph, there is at least one path $p \in P$ such that p contains the edge e*".

As discussed in Section 3.2, there are three categories of edges in the assembly control flow graph: (1) the control flow within the components, (2) the control flow within the additional code, and (3) the transfer of control between two components, or between a component and additional code. Edges from the first category are not included in the set of compositionally introduced test items, which thus can be described as $CI_{\mathcal{A}}^{\mathrm{BC}} = B_2 \cup B_3$, where B_2 and B_3 correspond to categories 2 and 3 above, respectively.

Regarding B_3, for **CM1**, these edges go from an exit statement of one component to the entry statement of another. Alternatively, if communication is carried out by glue code, from exit statements to some $s \in S_0$ and from some $s \in S_0$ to the entry statement of a component.

For **CM2**, B_3 consists of edges going from a method call statement in one component to an entry statement in the called component, and from the return statement of a method to the next statement of a caller, possibly linked by additional code statements.

For **CM3**, let C_i and C_i be two components, such that C_j has strictly higher priority than C_i. Then B_3 contains edges from all statements in S_i to the first statement of C_j, and from each final statement in S_j to all statements in S_i. Note that if the assembly constitutes the entire system, then additional information is available, such as periods, response times, etc. of all components in the system. This additional system-level information can be exploited to further reduce the number of edges in B_3.

4.3 Path Coverage Criterion

"*A set P of execution paths satisfies the path coverage criterion if and only if P contains all execution paths from the begin node to the end node in the flow graph*" [1].

For this criterion, the compositionally introduced test items are those paths in the control flow graph of $\mathcal{A}$ which includes a statement from S_0, or two statements $s_k \in S_i$ and $s_l \in S_j$, such that $i \neq j$.

For **CM1**, the paths in $CI_{\mathcal{A}}^{\mathrm{PC}}$ are sequential combinations of local control flow paths of the components, respecting the order in which they are connected in the pipes and filter scheme. For **CM2**, $CI_{\mathcal{A}}^{\mathrm{PC}}$ is the set of interleaved control flow paths, where the points of interleaving are constituted by the calls and returns of methods between components. For **CM3**, $CI_{\mathcal{A}}^{\mathrm{PC}}$ consists of interleaved control flow paths, where the points of interleaving are governed by component priority levels, similarly to the branch coverage criterion discussed above.

4.4 MC/DC Criterion

According to Chilenski and Miller [27], the Modified Condition/Decision Coverage (MC/DC) criterion is satisfied when "*every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions*".

For **CM1**, **CM2**, and **CM3**, the set of compositionally introduced test items is the set of test items introduced by the additional code. To show that these are the only test items introduced by the integration of components, we establish that no new points of entry and exit are introduced by composition, and that the only conditions in the resulting assembly are the conditions residing in the assembled components.

4.5 All Uses Coverage Criterion

"*A set P of execution paths satisfies the all-uses criterion if and only if for all definition occurrences of a variable x and all use occurrences of x that the definition feasibly reaches, there is at least one path p in P such that p includes a subpath through which that definition reaches the use*" [1].

The compositionally introduced test items of the all uses criterion is given by $CI_{\mathcal{A}}^{\mathrm{US}} = D_1 \cup D_2$, where D_1 is the set of pairs of definiton and uses that fulfil the criterion in the control flow graph of the additional code; D_2 is the set of pairs of definitions and uses that fulfil the criterion, and where the definition of the variable is performed in a component C_i and the use of the variable is performed in a component C_j, such that $i \neq j$.

In **CM1**, all component communication is supposed to take place via the explicit component ports. If this can be ensured, e.g., by the development framework, D_2 is empty. For **CM2**, it is also the case that D_2 is empty if the underlying formalism does not permit shared variables. Since **CM3** allows shared variables, D_2 is simply the set of feasible inter-component shared variable definition and use pairs.

5 Discussion

Our research so far has been aimed at developing a general formal theory for identifying additional structural test efforts required under different models of component interactions. The consideration of three architectural styles and five test criteria yields fifteen possible architectural style/test criterion combinations, of which not all are reasonably applicable. Below, we reflect upon the most notable combinations.

The fact that statement coverage composes nicely might be no major surprise, since it is intuitive that no new statements are introduced during integration (besides the ones in the extra code). Branch coverage, however, is more interesting during integration testing, since the set of all compositionally introduced test items for branch coverage describes all transfers of control from one component in the assembly to another. For **CM3**, covering these transfers of control quickly becomes impracticable without rigid restrictions on the scheduling of components, but for **CM1** and **CM2**, covering these branches would be a feasible way of testing explicit component interactions.

Path coverage suffers severely from complexity issues even at the component level [1], and would at best be applicable for very small systems conforming to **CM1** and **CM2**, with a handful of branching statements. Furthermore, the fact that MC/DC coverage scales well for all architectural styles might be interesting to component-based software developers building systems that should conform to a standard that requires such coverage (e.g., [25]). Finally, as related work shows [11,26], data flow (e.g., all uses) coverage on integration level is useful for detecting interleaving failures such as race conditions and stale-value errors in systems conforming to **CM3**.

6 Related Work

Previous contributions in testing of component-based systems range from verification of execution time properties by evolutionary testing [28], through regression testing of components based on information provided regarding component

changes [29], to model-based testing of component-based systems [30]. Despite this variety of contributions in this field, there exists, to our knowledge, no previous work discussing traditional structural test coverage in the integration testing of component-based systems. However, the fact that components need to be tested in the integrated setting in which they are intended to operate is recognized, e.g., by Weyker [31]. It is our firm belief that the quality of the verification would benefit by complementing the functional testing traditionally performed during integration with structural coverage as described in this paper.

Outside the component-based development community, the most notable efforts regarding structural testing on integration- or system level has been investigations of how to achieve structural coverage in concurrent systems of different flavours (typically focusing on definitions and uses of shared variables [11,26]). For structural testing of concurrent systems, many approaches combine the internal control flow structure of concurrent threads with the possible synchronizations between the threads. By doing this, a system-level control flow representation for structural testing can be achieved. Also related to this work, are contributions describing specialized structural test criteria focusing on the control flow paths of concurrent programs [32,33,34,35]. In contrast to the above works, the contribution of this paper is an effort to generalize the problem by considering several architectural styles, and a variety of test criteria.

7 Conclusions and Future Work

Building a system out of well-tested components does not necessarily result in a well-tested system. Generally, after integration of well-tested components into a subsystem or a system, the interaction between components remain to be tested. Using functional testing, we will cover the functional aspects of the integration (if the specification used as the base for test case generation is of a sufficient quality), but in order to cover structural and non-functional aspects, structural testing is required. Furthermore, structural coverage measures are not perfect, but they constitute the main formal quality assurance measures available in software testing.

In this paper, we have described the added test effort required by component integration, by introducing the concept of *compositionally introduced test items*. Also, we have described the impact of this concept for a number of common structural test criteria considering common architectural styles. Second, we have shown what is required in order to achieve structural test coverage at an arbitrary level of integration in multi-level hierarchical development of component-based software. By doing this, we have facilitated a less ad-hoc way of determining the adequacy of integration- and system-level testing than the functional testing traditionally used at this level.

Extending this approach to other component models and identifying impacts of relaxing some of our assumptions will be the immediate followups of this work. Several interesting questions also need to addressed such as a) what information we need to provide at the component interface level and b) what happens if

we do not have access to source code. We have also assumed strong adherence to component model semantics at lower levels of implementation, which cannot be taken for granted in many systems where the underlying implementation could be based on languages such as C. Scenarios that are potentially capable of invalidating the results need to identified and appropriate additional test efforts need to be incorporated.

Furthermore, in the continuation of this work, a goal would be to, for different architectural styles, and different test criteria, find a set test cases that safely covers the set of compositionally introduced test items. A second goal would be to find the minimal set of test cases that fulfils this criterion.

References

1. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys (CSUR) 29(4), 366–427 (1997)
2. Juristo, N., Moreno, A.M., Vegas, S.: Reviewing 25 Years of Testing Technique Experiments. Journal of Empirical Software Engineering 9(1-2), 7–44 (2004)
3. Frankl, P.G., Weyuker, E.J.: An Applicable Family of Data Flow Testing Criteria. IEEE Transactions of Software Engineering 14(10), 1483–1498 (1988)
4. Pavlopoulou, C., Young, M.: Residual test coverage monitoring. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 277–284. IEEE Computer Society Press, Los Alamitos (1999)
5. Craig, R.D., Jaskiel, S.P.: Systematic Software Testing. Artech House Publishers (2002)
6. van Veenendaal, E.: The Testing Practitioner. Uitgeverij Tutein Nolthenius (2002)
7. Copeland, L.: A Practitioner's Guide to Software Test Design. STQE Publishing (2003)
8. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House Publishers (2002)
9. Lau, K.K., Wang, Z.: A Survey of Software Component Models, 2nd edn., May 2006. Pre-print CSPP-38, School of Computer Science, The University of Manchester (2006)
10. Shaw, M., Garland, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
11. Sundmark, D., Pettersson, A., Sandberg, C., Ermedahl, A., Thane, H.: Finding DU-Paths for Testing of Multi-Tasking Real-Time Systems using WCET Analysis. In: Proceedings of the 7thInternational Workshop on Worst-Case Execution Time Analysis (WCET 2007) (July 2007)
12. Pretschner, A.: Compositional Generation of MC/DC Integration Test Suites. Electronic Notes in Theoretical Computer Science 82(6) (2003)
13. Goldberg, A., Wang, T.C., Zimmerman, D.: Applications of Feasible Path Analysis to Program Testing. In: ISSTA 1994: Proceedings of the 1994 ACM SIGSOFT international Symposium on Software Testing and Analysis, pp. 80–94. ACM Press, New York (1994)
14. Gustafsson, J., Ermedahl, A., Lisper, B.: Algorithms for Infeasible Path Calculation. In: Sixth International Workshop on Worst-Case Execution Time Analysis (WCET 2006), Dresden, Germany (July 2006)
15. Hayes, I., Fidge, C., Lermer, K.: Semantic Characterisation of Dead Control-Flow Paths. IEE Proceedings - Software 148(6), 175–186 (2001)

16. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software 80(5), 655–667 (2007)
17. Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Black, A.P., Müller, P.O., Zeidler, C., Genssler, T., van den Born, R.: A component model for field devices. In: Proc. of the 1st Int. IFIP/ACM Working Conference on Component Deployment, pp. 200–209. Springer, Heidelberg (2002)
18. Sun Microsystems: JavaBeans Specification 1.01 (August 1997), http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html
19. Box, D.: Essential COM. Addison-Wesley, Reading (1997)
20. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer 33(3), 78–85 (2000)
21. Lundbäck, K.L., Lundbäck, J., Lindberg, M.: Component Based Development of Dependable Real-Time Applications. Technical report, Arcticus Systems, http://www.arcticus.se
22. Sandström, K., Fredriksson, J., Åkerholm, M.: Introducing a component technology for safety critical embedded realtime systems. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 194–209. Springer, Heidelberg (2004)
23. Ke, X., Sierszecki, K., Angelov, C.: COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In: Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 199–208. IEEE, Los Alamitos (2007)
24. Bureš, T., Carlson, J., Crnković, I., Sentilles, S., Vulgarakis, A.: ProCom - the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University (June 2008)
25. RTCA: Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. RTCA (December 1992)
26. Yang, C.S.D., Pollock, L.L.: All-uses Testing of Shared Memory Parallel Programs. Software Testing, Verification and Reliability 13(1), 3–24 (2003)
27. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal, 193–200 (1994)
28. Groß, H.G., Mayer, N.: Evolutionary testing in component-based real-time system construction. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA, USA, p. 1393. Morgan Kaufmann Publishers Inc., San Francisco (2002)
29. Mao, C., Lu, Y.: Regression testing for component-based software systems by enhancing change information. In: APSEC 2005: Proceedings of the 12th Asia- Pacific Software Engineering Conference, Washington, DC, USA, pp. 611–618. IEEE Computer Society, Los Alamitos (2005)
30. Pelliccione, P., Muccini, H., Bucchiarone, A., Facchini, F.: TeStor: Deriving Test Sequences from Model-based Specifications. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 267–282. Springer, Heidelberg (2005)
31. Weyuker, E.J.: Testing Component-Based Software: A Cautionary Tale. IEEE Softw. 15(5), 54–59 (1998)
32. Katayama, T., Itoh, E., Ushijima, K., Furukawa, Z.: Test-Case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences. In: Proceedings of sixth Asia-Pacific Software Engineering Conference (APSEC 1999), p. 590 (1999)

33. Taylor, R.N., Levine, D.L., Kelly, C.D.: Structural Testing of Concurrent Programs. IEEE Transactions on Software Engineering 18(3), 206–215 (1992)
34. Wong, W.E., Lei, Y., Ma, X.: Effective Generation of Test Sequences for Structural Testing of Concurrent Programs. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005), Washington, DC, USA, pp. 539–548. IEEE Computer Society, Los Alamitos (2005)
35. Yang, R.D., Chung, C.G.: Path Analysis Testing of Concurrent Program. Information and Software Technology 34(1), 43–56 (1992)

Towards Component-Based Design and Verification of a μ-Controller*

Yunja Choi and Christian Bunse

[1] School of Electrical Engineering and Computer Science
Kyungpook National University, Daegu, Korea
yuchoi76@knu.ac.kr
[2] School of IT
International University, Bruchsal, Germany
Christian.Bunse@i-u.de

Abstract. Model-driven and component-based software development seems to be a promising approach to handling the complexity and at the same time increasing the quality of software systems. Although the idea of assembling systems from pre-fabricated components is appealing, quality becomes a major issue, especially for embedded systems. Quality defects in one component might not affect the quality of the component but that of others. This paper presents an integrated, formal verification approach to ensure the correct behavior of embedded software components, as well as a case study that demonstrates its practical applicability. The approach is based on the formalism of abstract components and their refinements, with its focus being on interaction behavior among components. The approach enables the identification of unanticipated design errors that are difficult to find and costly to correct using traditional verification methods such as testing and simulation.

1 Introduction

Concerning stand-alone devices, the correctness and reliability of the relevant control systems have only limited effects. In today's new computing environments, such as ubiquitous computing and autonomous systems, however, the reliability of one, even small, embedded system may affect a large network of embedded systems. Thus, there is an urgent need for a structured development methodology with integrated verification support [13]. Model-driven Development(MDD), combined with component architecture, has been increasingly attracting researchers and industry practitioners in this regard.

Component-oriented MDD helps to cope with the increasing complexity of software system development. MDD development processes, such as Marmot [2,4] explicitly distinguish component specifications (contracts or interfaces) from component realizations (implementations). The Marmot approach

* A longer version of this paper is under review for publication in Formal Aspects of Computing.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 196–211, 2008.

maintains simplicity through a divide-and-conquer strategy, while keeping continuity and consistency through systematic, iterative refinements of components across different life-cycle stages (design, implementation, etc.). Furthermore, it facilitates automated formal verification as well as model-based simulation and testing, which can be naturally blended into the abstraction refinement process.

In this paper, we present a formal verification approach integrated into model driven development process. Our approach is two-fold: (1)We formalize components at each iteration of the component specification process and apply model checking [8] as a formal verification method to ensure behavioral consistency with the external environment, even before each component is completely realized. (2) We formalize gradual refinements through decomposition and verify their validity by checking behavioral consistency. While traditional verification methods, such as testing and simulation, focus on verifying *expected outputs* from *planned inputs*, model checking is based on exhaustive verification for all possible input sequences, and, thus, is better suited for ensuring high reliability and safety.

We demonstrate our approach by using a component-oriented MDD framework (based on MARMOT) in the development of a mirror-control system (i.e., an embedded system controlling the movement of a car's exterior mirror). The focus is on checking the behavioral consistency and correctness of the system's μ-controller. We present a system model and formalize three important notions, namely: abstract component, realization, and refinement. These provide the basis for a systematic transformation into a formal language that can be checked. The model checker SPIN [11] is used for checking interaction consistency and the essential properties of the controller, which reveals a potentially fatal problem in the original controller design – the possibility that a user request may be postponed indefinitely. We have identified the centralized control of events in the design of the μ-controller as the source of the problem by analyzing the counter-examples generated by the model checker. We show that changing each driver component into an independent and active component addresses the issue. This exemplifies the importance of formal verification in early design stages.

The remainder of this paper is organized as follows; Section 2 and Section 3 provide a brief description of the mirror control system and of using MARMOT for its development. Section 4 defines the notion of abstract components and their interaction behavior. Section 5 demonstrates how components of a mirror control system can be formalized and translated into the formal specification language PROMELA [10], depending on the level of abstraction, and explains the effect of formal verification on the change of design. We conclude with a brief discussion in Section 6.

2 Mirror Control System

The mirror control system is an embedded system composed of electrical and mechanical components and is used to control the movement of a car's exterior mirror. The system allows a mirror to be moved horizontally and vertically into a

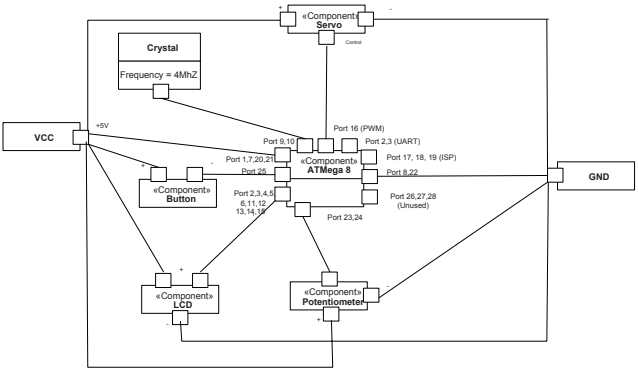


Fig. 1. UML representation of hardware

position that is convenient for the driver. Cars supporting different driver profiles can store the mirror position and recall it as soon as the profile is activated.

In the context of this paper, the mirror control system was realized in a simplified version using a μ-controller, i.e., an $ATMEL^{TM}$ $Mega$ 8, a button, and two servos (the Servo-Control System). In detail, this system requires the μ-controller to read values from the potentiometers (i.e., analog-digital conversion), converts them into the mirror turning degree, and generates the needed servo control signals, while at the same time indicating movement and degree on an LCD display. In addition, the system stores a mirror position that can be recalled by simply pressing the button. Positions are stored by pressing the button for more than five seconds. Storing and recalling is also visualized on the LCD display. Figure 1 shows a simplified UML[1] representation of the electronic circuit, representing the structural model of the mirror control system.

The requirements of the mirror control system are described by use case diagrams (Figure 2(a)) and an interaction diagram (Figure 2(b)) representing the general flow of control. The use case diagrams describe how the actor 'User' initiates the task of controlling the servo rotation. The interaction diagram provides an alternative view of the way in which user tasks are performed and shows the typical sequence of operations concerning the overall system.

3 Component-Based Development

3.1 Abstract Component and Refinements

Following the component-based development process MARMOT [2,4], we view a system as a tree-shaped hierarchy of components, where the root represents the system as an abstract component, the parent/child relation represents composition, and the leaf components represent final implementation. Figure 3 illustrates

[1] The Unified Modeling Language [7].

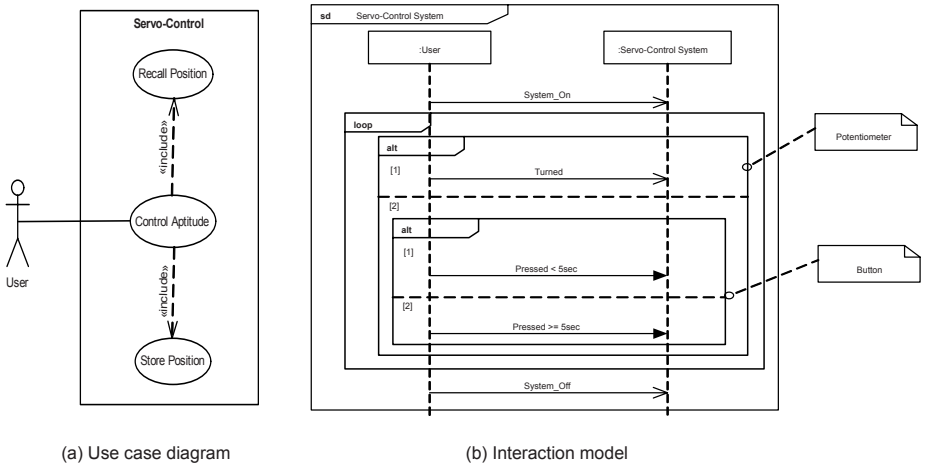


Fig. 2. Mirror control system context realization

the structure of the MARMOT component divided into an externally visible contract named specification and the internal realization part; UML class diagrams and object diagrams are used to specify the external and internal structure of the component. Activity/interaction diagrams are used to specify external and internal behavior of the component. Note that we can use HDL or system C for lower-level component specifications instead of UML diagrams.

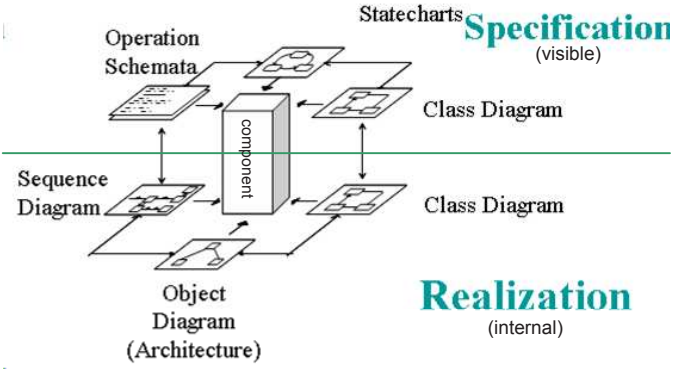
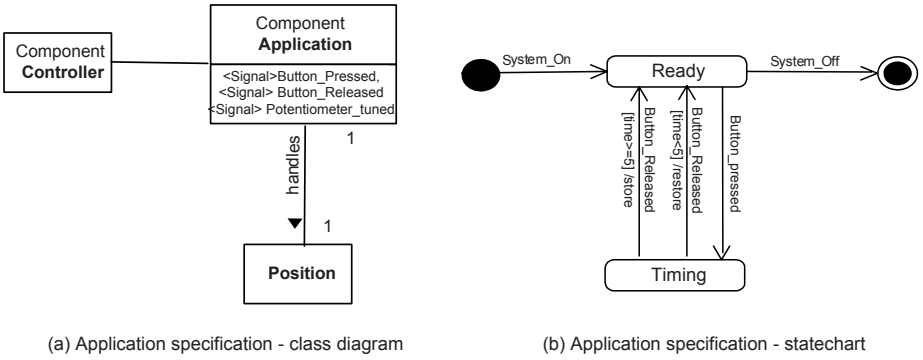


Fig. 3. Structure of a component

3.2 Component Specification

In this section, we illustrate with examples how MARMOT (i.e., the MDD methodology) is applied to the development of the mirror controller. Since the hardware environment is pre-defined, we focus on the development of the software part, namely the Driver and the Application components. The Controller



(a) Application specification - class diagram

(b) Application specification - statechart

Fig. 4. Specifications for the Application component

component is a container without any software functionality, but contains the Application component. Figure 4(a) shows the specification-level class diagram of the Application component in its context. The component does not offer any operations to the outer world, but reacts to signals. This is denoted by the UML 2.0 stereotype *signal*. The state diagram of the Application component (see Figure 4(b)) shows that the Application component is in the *ready* state, after the *system_on* event. When the *button_pressed* event happens, the component transits to the *timing* state where "the time till the next *button_released* event occurs" is measured. Depending on whether the time is less than 5 seconds or not, it signals the *restore* action or the *store* action to the *Servo* component and transits back to the *ready* state.

3.3 Component Realization

While the externally visible behavior of the Application component can be uniquely specified according to the functionality required by the component, there can be various ways of realizing such functionality depending on the design decision and available hardware components that can be utilized. The MARMOT realization step specifies how the externally visible behavior of the component is actually realized through decomposition and collaboration among sub-components. Figure 5(a) shows the Driver component used to realize the functionality of the Application component, which is again decomposed into five sub-components that handle the corresponding hardware component. The interaction behavior among components is specified in an interaction diagram. For example, Figure 5(b) specifies the interaction behavior between the Application component and the button driver; the *button_pressed* (*button_released*) event from the user initiates the *Timer_starts* (*timer_stops*) action in the *button* driver, and then, depending on the duration of the button pressed event, the Application component interacts with the button driver to either set or store the mirror position in the *Servo* component.

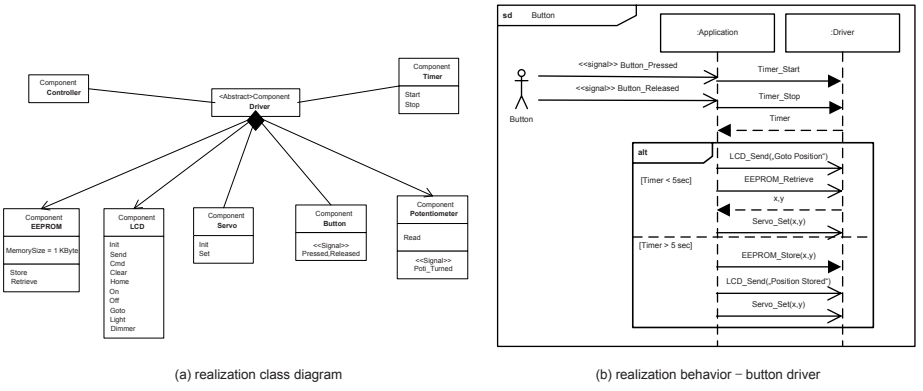


Fig. 5. Realization of the Application component

4 Formal Definition and Translation

To ensure the correctness and consistency of its complex behavior produced by the compositions of dozens, possibly hundreds, of components, we first define the meaning of component interactions and their inter-relationships using π-calculus [16], with an emphasis on their communication behavior. The formalism defined with π-calculus serves as a basis for defining translation rules from MARMOT components to formal modeling languages such as PROMELA for verification purposes. π-calculus supports parallel composition of processes, synchronous communication between processes through channels, dynamic creation of channels, and non-determinism — characteristics suitable for formalizing abstract components and their interaction behaviors.

4.1 Formal Meaning of the Abstract Component

Figure 6 summarizes the formal descriptions for the major artifacts of a MARMOT component.

We define an abstract component as a composition of two parallel processes consisting of an interface I and an externally visible body $Spec_0$ (see row 1 of Figure 6); each *Comp_spec* has pre-defined input/output channels (i, o), a set of operations *op_set*, and a set of actions *action_set* that are visible from outside the component. These are used for the component to interact with its external environment. The interface I and $Spec_0$ interchange messages and events through the internal channels u, v. Here, *new* u, v means that the channels u, v are dynamically created within the component with a limited scope. The symbol " | " represents parallel composition of two processes.

An interface $I(i, o, u, v)$ (see row 2 of Figure 6) either receives a message x from input channel i and forwards it to the internal message channel u, or receives a message y from the internal output channel v and forwards it to output channel o. Here, $i?x$, $u!x$ represents an input event on channel i, an output event on channel

	Type	Formal description
1	Abstract component	Comp_spec(i,o,op_set, action_set) = new u,v (I(i,o,u,v) | Spec_0(u,v, op_set, action_set))
2	Interface	I(i,o,u,v) = i?x.u!x.I(i,o,u,v) + v?y.o!y.I(i,o,u,v)
3	Component behavior	Spec_i(u,v,op_set,action_set) = u?x.[x=op_k].Action_spec_k(u,v,op_set,action_set) + u?x.[x!=op_k].Spec_i(u,v,op_set, action_set)
4	Abstract implementation	Action_spec_i(u,v,op_set,action_set) = (v!a_j)*.Spec_j(u,v,op_set,action_set)
5	Component realization	Comp_real(i,o,op_set,action_set) = new u,v, {(u_i,v_i)}_i (I(i,o,u,v) | !_i SubComp_i (u_i,v_i,sub_op_set_i, sub_action_set_i) | Composit_Rel(u,v, {(u_i,v_i)}_i)
6	Component Relations	Composit_Rel(u,v,{(u_i,v_i)}_i) = ∑ v_i?x.f(v_i)!x.Composit_Rel(u,v,{(u_i,v_i)}_i)

Fig. 6. Definitions of components and refinements

u, with message/signal x, respectively. Two events that are concatenated with a "." symbol occur sequentially; for example, $i?x.u!x$ means that an input event is followed by an output event. A "+" symbol means a non-deterministic choice between two different event sequences. Note that I is recursively defined so that it transits back to itself after any pair of input/output events.

The behavior of a component is defined with a series of processes from $Spec_0$, representing the process at the initial state, to $Spec_i$ representing the process at the i_{th} state, as defined in row 3 in Figure 6. $Spec_i$ receives a message x, checks whether it matches one of the operations in the op_set, and performs corresponding actions $Action\_spec_k$ if it matches an operation op_k, and does nothing if it does not match any of the operations in the set[2]. $Action\_spec_i$ (the row 4 in Figure 6) defines a series of actions that need to be performed by the component for a particular operation. The actions are notified to the internal output channel v, which is forwarded to the external output channel o by the interface I[3]. After all the action outputs, the process reduces to $Spec_j(u, v, op\_set, action\_set)$ where the mapping from $Spec_i$ to $Spec_j$ is predefined by the conditions on the message values. In other words, there is a mapping from $\{(i, x) \mid \textit{process state } i,\ \textit{message value } x\}$ to $\{j \mid \textit{process state } j\}$. This mapping can be extracted from the statechart diagrams in the specification model.

4.2 Formal Meaning of Component Realization

Specifications of a component uniquely define the externally visible behavior of the component regardless of how it is realized internally. On the other hand, each functionality can be realized in many different ways through decomposition and refinements. The focus of this realization process is to make it as flexible

[2] The simplified notation $[x = op_i].Action\_spec_i(u, v, action\_set)$ is used instead of enumerating all possible matches.

[3] The simplified notation $(v!a_j)^*$ is used to denote a series of output actions instead of $v!a_1.v!a_2.v!a_3..v!a_n$.

as possible so that the change of a certain realization of a component does not affect the overall interaction behavior. To this end, we formally refine the abstract component *Comp_Spec* process with the *Comp_real* process, as defined in the row 5 and row 6 in Figure 6, consisting of a number of parallel *SubComp* processes that collaboratively realize the *Spec* process of *Comp_Spec*. Note that each *SubComp* is considered as an independent component on its own, and, thus, can be recursively specified as an abstract component in the same way as *Comp_Spec*.

In Figure 6 (Component Realization), $\{u_i, v_i\}_i$ abbreviates an i number of input/output channel pairs, and $!_i$ abbreviates the parallel composition of a number of *SubComp* processes whose interrelation is defined in *Composit_Rel*; for each component, the destination of its output message is uniquely defined in *Composit_Rel* in the form of a function $f : \{v, \{v_i\}_i\} \longrightarrow \{u, \{u_i\}_i\}$. This function f is used to wire sub-components and can be changed independently from the implementation of each *SubComp*, supporting flexible design for component-based development.

4.3 From MARMOT to PROMELA

We use the model checker SPIN [11] as a back-end verifier for MARMOT models. There are a couple of automated verification tools directly supporting π-calculus, such as the Mobility Workbench from Uppsala University. Nevertheless, their efficiency and usability are not as good as those of general-purpose model checkers such as SPIN and SMV, and, thus, they are not yet suitable for routine use during the development process.

The use of SPIN requires a translation of MARMOT models into PROMELA, the input language of SPIN. The syntactic transformation from MARMOT to PROMELA is based on the formal meaning of MARMOT components defined in the previous sections. Note that our translation approach is specialized in MARMOT components. For more general translations, please refer to [21].

Figure 7 shows some of the syntactic translation definitions from MARMOT to PROMELA; the names of operations and actions in a MARMOT component are translated into elements of the PROMELA *mtype* construct. Each communication channel in a MARMOT component is declared as a message channel of *mtype* in PROMELA. Each MARMOT component specification, component interface, and component realization corresponds to a *proctype* declaration. The PROMELA *run* construct is used to activate an interface process or a specification process in a component. Message sending and receiving actions can be directly translated into $u!x$ and $u?y$ where x and y are declared as *mtype*. A behavioral specification $Spec_i$ corresponds to a state of a component whose transition is defined by the transitions in $Spec_i$. A similar translation applies to $Action\_spec_i$. Non-conditional action transitions are translated into sequential actions followed by a *goto* statement. The PROMELA if construct is used for conditional transitions.

Note that we omit detailed translation rules from UML diagrams to PROMELA statements to save space. Interested readers may refer to existing approaches [9,15].

	MARMOT *construct*	PROMELA *construct*
messages	$O = \bigcup_{op\_set,action\_set}$ $\{n \mid n \in op\_set \text{ or } n \in action\_set\}$	$mtype = \{n_1, n_2, \ldots, n_k\}$, where $n_i \in O$.
channels	new u	chan u = [1] of mtype
Processes	I(i,o,u,v) Comp_spec(i,o,op_set, action_set) Comp_real(i,o,op_set, action_set)	proctype Interface(chan i,o,u,v) proctype Comp_spec(chan i,o){...} proctype Comp_real(chan i,o){ ...}

Process Activation	Comp_Spec(i,o,O,A) = new u,v I(i,o,u,v) | Spec(u,v,O,A)	proctype Comp_Spec(chan i,o){
		chan u = [1] of mtype;
		chan v = [1] of mtype;
		run Interface(i,o,u,v);
		run Spec(u,v); }

actions	u?x u!y	mtype x; u?x; mtype y; u!y;
states	$Spec_i(u, v, op\_set, action\_set)$	$state_i$:
transitions	$\pi.Spec_i(u, v, op\_set, action\_set)$	π; goto $state_i$;
conditionals	$u?x.[x = a]Spec_i(u, v, O, A)$	if :: u?[a] $\rightarrow$ goto $state_i$; fi;

Fig. 7. Syntactic translation from MARMOT to PROMELA

5 Applying Formal Methods

Based on the formalism introduced in the previous section, we now transform the UML representation of the Application component in Figure 4 into formal models in PROMELA. PROMELA [10] is the input language of the SPIN [11] model checker, which is widely used for software verification.

5.1 Formalizing the Specification of the Application Component

Direct Translation. First, we specify the Application component *Comp_Spec* using the formal definition for abstract components in Figure 6 as follows:

```
mtype = { system_on, system_off, button_pressed, button_released,
          poti_tuned, store, restore};

proctype Comp_spec(chan i, o){
   chan  u = [1] of {mtype};
   chan  v  = [1] of {mtype};
   run Interface(i, o, u, v);
   run  Spec(u,v);
}
```

In this specification, *mtype* declares the set of actions and operations used in the Application component. The *proctype* declaration is used to declare the component process with the name *Comp_spec* and the input, output channels i, o are declared in the signature of the component. Within the process *Comp_spec*, u, v are declared as channels with the message type *mtype*, i.e., the two internal channels are used to deliver messages/signals of actions and operations. Two parallel processes, $Interface$ and *Spec*, are activated by *Comp_spec* as its sub-processes using the keyword *run*.

The next PROMELA code shows the specification of the *Spec* process whose behavior is derived from the statechart of the Application component in Figure 4(b); the four labels, *Spec_0*(line 3), *Spec_1*(line 9), *Spec_2*(line 16), and *end_state*, represent the initial state, *ready* state, *timing* state, and the final state, respectively. The *Spec* process is initially in the *Spec_0* state and transits to the *Spec_1* state if the *system_on* signal is received. The transition from *Spec_1* occurs either to *Spec_2* or to *end_state* when the button is pressed or the *system_off* signal is received. In *Spec_2*, it non-deterministically sends out *store* or *restore* messages and transits to *Spec_1* if the *button_released* event occurs (line 19–line 22). Otherwise, it transits to *Spec_2* (line 23). Note that predicate abstraction [6] is applied to the original guarded action, "if $time < 5$ then *restore*, else if $time \geq 5$ then *store*", so that it is transformed into a non-deterministic choice of actions between *restore* and *store*; we first replace $time < 5$ with a boolean variable t transforming the guarded action into "if t then *restore*, else if $\neg t$ then *store*". Since the value of t is determined non-deterministically at this abstract level, we replace the guarded action with "non-deterministic choice between *store* and *restore*" as expressed in line 19 – line 22.

The specification for the Interface process is transformed similarly.

```
 1:   proctype Spec(chan u,v){
 2:     mtype x;
 3:      Spec_0:
 4:           u?x;
 5:           if
 6:           :: x == system_on -> goto Spec_1;
 7:           :: else -> goto Spec_0;
 8:           fi;

 9:      Spec_1:
10:           u?x ;
11:           if
12:            :: x == button_pressed -> goto Spec_2;
13:            :: x == system_off -> goto end_state;
14:            :: else -> goto Spec_1;
15:           fi;

16:       Spec_2:
17:           u?x;
18:           if
19:           :: x == button_released -> if
20:                                      :: 1 -> v!store; goto Spec_1;
21:                                      :: 1 -> v!restore; goto Spec_1;
22:                                      fi;
23:           :: else -> goto Spec_2;
24:            fi;

25:       end_state: goto Spec_0;
     }
```

Formal Consistency Checking. Once the abstract component is specified in PROMELA, we can check whether the behavior of the abstract component is consistent with its environment or not, even before we specify the actual implementation of the component. The notion of interaction consistency, which is formally defined in [4], can be informally stated as follows;

A component is consistent with its environment in its behavior if it either terminates normally or runs infinitely under the infinite sequence of stimuli generated from its environment.

Note that the negation of the interaction consistency implies a process deadlock situation, and thus, it is quite important to ensure that the initial design of a component satisfies the interaction consistency.

The specification for the environment of the Application component is derived from the use case scenarios shown in Figure 2, which can be directly transformed into PROMELA as shown below.

```
proctype env(chan in, out){
  do
  ::  out!system_on;
      do
      :: 1 ->
         if
         :: out!poti_tuned;
         :: out!button_pressed;
            out!button_released;
         fi;
      :: 1 -> break;
      od;
      out!system_off;
  :: 1 -> skip;
  od;
}
```

In this specification, the statements enclosed by *do..od* act like unconditional *while statements* in the C language; the statement repeats indefinitely as the *poti_tuned* signal or the *button_pressed* signal is generated non-deterministically.

This environment process *env* is composed with the *Comp_Spec* process, producing a system model for checking interaction consistency. The abstract Application component is verified to be consistent with its environment in this context using the SPIN verifier[4]; it took about 1 minutes and 521 M of memory for exhaustive verification, exploring 6×10^6 states and 1.7×10^7 transitions. Verification was performed on a PC with 2G Herz Pentium II processor and 2G bytes of memory.

5.2 Formalizing the Realization of the Application Component

The application component for the mirror control system is realized by a number of device drivers as illustrated in Figure 5(a). The realization behavior is specified in sequence diagrams defining which sub-components (device drivers) are used to realize a specific function provided by the Application component; an example is illustrated in Figure 5(b) for the *button* driver. Note that all the sub-components are designed as passive objects in this realization model and the Application component (the container of the driver components) acts as an active signal

[4] SPIN verifier provides an *invalid end-state* option which can be used to check the behavioral consistency between a component and its environment.

control center. Each signal passed to the Application component is identified with its source and handed to the corresponding driver depending on the source of the signal.

The next section of code shows a major part of the PROMELA specification for the realization model of the mirror control system directly translated from the realization diagrams.

```
 1: proctype ATMega(chan in1,out1, in2, out2, in3, out3, in4,
                    out4, in5, out5, tin, tout, sys_in,  sys_out){
 2:   mtype m;

 3:   hw_ready:
 4:       sys_in?m;
 5:       if
 6:       :: m == system_on ->system_state = system_on; goto driver_choice;
 7:       :: else -> goto hw_ready;
 8:       fi;

 9:   driver_choice:
10:     if
11:     :: in1?[m] -> goto button_driver;
12:     :: in2?[m] -> goto servo_driver;
13:     :: in3?[m] -> goto LCD_driver;
14:     :: in4?[m] -> goto potentiometer_driver;
15:     :: in5?[m] -> goto EEPROM_driver;
16:     :: sys_in?[m] -> goto sys_control;
17:     fi;

18:    sys_control:  ..
19:    button_driver:
20:       in1?button_pressed;          /* wait for button_pressed event */
21:       tin!set;                     /* set timer */
23:       in1?button_released;         /* wait for button_released */
24:       tin!reset; tout?m;           /* reset timer and get the timing info */
25:       if                           /* non-deterministic choice of action */
26:       :: 1 ->  out2!restore;
27:       :: 1 ->  out2!store;
28:       fi;
29:
30:     goto driver_choice;

31:   servo_driver: ..
32:   potentiometer_driver: ..
    }
```

Note that this realization model specifies the internal implementation of the system with detailed interactions among drivers and the controller; the $ATMega$ process is initially in the *hw_ready* state waiting for the *system_on* signal, which initiates a transition to the state *driver_choice* (line 9–17) from which $ATMega$ handles incoming messages and signals and chooses an appropriate driver. For example, if the signal is for the *button* driver, the $ATMega$ process transits to the *button_driver* state, where the signal is handled as specified in the realization behavior in Figure 5(b) (line 20–30).

We have composed the $ATMega$ process with the same environment model *env* illustrated in the previous section and checked for interaction consistency using SPIN to verify that the $ATMega$ process is a valid realization of the mirror control system. SPIN verifies the interaction consistency on this model within 4 minutes consuming 549 M of memory, after searching 1.5×10^7 states and 3.8×10^7 transitions.

5.3 Property Verification and Design Change

One of the major purposes of formalizing a design model is to identify and verify key properties of the design and address issues related to the key properties if the verification activity reveals design errors. Once the design is formalized, we can apply automated verification for various design properties. For example, we may want to make sure that each external event *button_pressed* followed by *button_released* always has an effect on the *Servo* component, either setting or restoring the position of the mirror. This property can be formally stated in temporal logic[5] as

$$button\_pressed \,\&\&\, !\,(servo\_set \,||\, servo\_restore) \rightarrow true\, U\, (servo\_set || servo\_restore)),$$

meaning that *"for all possible execution traces, if the button is pressed and the servo is currently neither set nor restored, then the servo will be set or restored sometime in the future."*

This property is proven to be false in our realization design by the SPIN verifier, which generates counter-examples showing various execution traces violating this property. For example, one counter-example shows that the system can stall without making any progress (process deadlock) if the *reset* signal to the timer gets lost in the middle of delivery. This is because the model is designed to lose additional messages if the channel is already occupied by another message. Another counter-example illustrates that there can be an infinite sequences of signals from *Potentiometer* that occupies the signal handler of the *ATMega* process all the time so that the handling of the *button_pressed* event is postponed indefinitely. This problem happens mainly because there is only one active process handling all the events and messages. If such a process is occupied by a hostile external component, the system cannot function as expected.

After careful review of the original design and the counter-examples, the realization model is redesigned to address the identified issues. We first introduce message buffers to make sure that there is no loss of messages directly affecting the system's behavior. We also make each driver component an active process that can handle events/messages on its own, instead of having one central message handler. The following illustrates how such a change in design is reflected in PROMELA;

```
proctype ATMega(chan in1,out1, in2, out2, in3, out3,
        in4, out4, in5, out5, tin, tout, sys_in, sys_out){
     ...
     run button_handler(in1, out1);
     run LCD_handler(in3, out3);
     run potentio_handler(in4, out4);
     ...
}

proctype button_handler(chan in, out, tin, tout){
     mtype m;
```

[5] Temporal logic can be considered as propositional logic with the notion of relative time. SPIN is equipped with an automated verification facility for properties written in temporal logic LTL [18].

```
      button_driver:
         in?button_pressed;
         tin!set;
         in?button_released;
         tin!reset; tout?m;
         if
         :: 1 -> out!restore;
         :: 1 -> out!store;
         fi;
         goto button_driver;
}
```

Note that the *ATMega* process is simplified containing only the statements initiating different drivers whose behavior is specified as an active and independent process. The drivers are connected to the *ATMega* process by wiring them with message channels. For example, when the *button_pressed* event arrives in channel *in*1, the *button_handler* process directly recognizes this event without going through the *ATMega* process and handles it as specified in the button driver. The behavioral specification for the *button_handler* process is the same as the one specified under the label *button_driver*(line 19–30) in the previous design. With this modified design, the same property is verified to be true[6].

6 Discussion

The use of formal methods in embedded systems has been an active research issue for almost a decade [14,23], but, unfortunately, we have not seen active practice of formal methods in industry, mainly due to lack of experience, supporting tools, and methodologies [13]. We believe this situation can be altered by integrating formal methods into existing development methodologies so that the application of a formal method can be seen as a routine task within the process. As demonstrated in this paper, the use of a structural methodology makes the application of formal methods simpler and easier by providing gradual yet seamless transitions from the early design to actual implementation. Our approach is partially automated by reusing the MARMOT-PROMELA prototype translation tool introduced in [4].

There have been other approaches that apply formal methods in embedded systems; for example, [20] uses a variation of Petri Net as the underlying formalism of a system model and translates it into PROMELA to use the SPIN verifier. Nevertheless, this approach and other related previous approaches [12,17,23] lack an association with development methodology. Several approaches have tried to address behavioral properties in system development [1,3,5,19,22,24], where some of them use model checking for checking properties of UML diagrams [1,24]; among them, [22] is the closest to our approach in the sense that their approach is closely coupled with a component-based system development process. Nevertheless, [22] takes a bottom-up approach by identifying properties for each component under environmental assumptions. Compositional verification is performed by cleverly assembling those properties of each sub-component that have

[6] It is verified to be true under fairness constraints that all the processes are executed infinitely often.

been already verified. On the other hand, our approach extracts environmental constraints from the internal behavior of the refined component, which is specified during the MARMOT refinement process, eliminating the need for manually identifying environmental assumptions [4].

Our approach emphasizes that the use of a structured development methodology such as MDD is necessary for achieving a high-quality system, but is not sufficient for it. While structured methods can cope with structural complexity using the well-known "divide-and-conquer" principle, the interaction complexity among decomposed parts of the system tends to get higher, which becomes a major problem. We tackle this particular problem with formal methods integrated into the development methodology.

We note that our approach presented in this paper is work in progress that requires further investigation on its practical aspects, especially with respect to usability and efficiency. We need more industrial case studies to claim that our approach is actually practical. There are other issues to be considered in the design of embedded systems, such as energy consumption, timing issues, and utilization of limited memory [13]. We plan to investigate such issues within the same verification frame in future work.

References

1. Adamek, J., Plasil, F.: Component composition errors and update atomicity: Static analysis. Journal of Software Maintenance and Evolution: Research and Practice (September 2005)
2. Atkinson, C., Bayer, J., Bunse, C., et al.: Component-based Product Line Engineering with UML. Addison-Wesley Publishing Company, Reading (2002)
3. Barros, T., Henrio, L., Madelaine, E.: Behavioural models for hierarchical components. In: International SPIN Workshop on Model Checking Software (August 2005)
4. Choi, Y.: Checking interaction consistency in MARMOT component refinements. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 832–843. Springer, Heidelberg (2007)
5. Engels, G., Kuester, J.M., Groenwegen, L.: Consistent interaction of software components. Journal of Integrated Design and Process Science 6(4), 2–22 (2003)
6. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
7. Object Management Group. UML2.0 superstructure specifications
8. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking: History, Achievements, Perspectives. Springer, Heidelberg (2008)
9. Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its PROMELA translation. In: 12th Asia-Pacific Software Engineering Conference (2005)
10. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall Software Series (1991)
11. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Publishing Company, Reading (2003)
12. Hsiung, P.-A.: Formal synthesis and code generation of embedded real-time software. In: 9th International Symposium on Hardware/Software Codesign (April 2001)

13. Johnson, S.D.: Formal methods in embedded design. IEEE Computer (November 2003)
14. Kern, C., Greenstreet, M.: Formal verification in hardware design: A survey. ACM Transactions on Design Automation of E. Systems (April 1999)
15. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in PROMELA/SPIN. In: Second IEEE Workshop on Industrial Strength Formal Specification Techniques (October 1998)
16. Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press, Cambridge (1999)
17. Naeser, G., Lundqvist, K.: Component-based approach to run-time kernel specification and verification. In: 17th Euromicro Conference on Real-Time Systems (2005)
18. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Foundations of Computer Science, pp. 46–57 (1977)
19. Reussner, R.H., Poernomo, I., Schmidt, H.W.: Reasoning about software architectures with contractually specified components. In: Component-Based Software Quality: Methods and Techniques, State-of-the-Art Survey (2003)
20. Ribeiro, O.R., Fernandes, J.M., Pinto, L.F.: Model checking embedded systems with PROMELA. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (2005)
21. Song, H., Compton, K.J.: Verifying pi-calculus processes by promela translation. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan (2003)
22. Xie, F., Browne, J.C.: Verified systems by composition from verified components. In: Proceedings of Joint Conference ESEC/FSE (2003)
23. Yang, W., Moo-Kyeong, Kyung, C.-M.: Current status and challenges of soc verification for embedded systems market. In: IEEE International Conference on System-On-Chip (2003)
24. Zimmerova, B., Brim, L., Cerna, I., Varekova, P.: Component-interaction automata as a verification-oriented component-based system specification. In: Workshop on Specification and Verification of Component-Based Systems (2005)

ESCAPE: A Component-Based Policy Framework for Sense and React Applications

Giovanni Russello, Leonardo Mostarda, and Naranker Dulay

Imperial College London, London SW7 2RH, United Kingdom
{russello,lmostard,n.dulay}@imperial.ac.uk

Abstract. Sense-and-react applications are characterised by the fact that actuators are able to react to data collected by sensors and change the monitored environment. With the introduction of nodes sporting actuators, Wireless Sensor Networks (WSNs) are being used for realising such applications. Sensor and actuator nodes are capable of interact locally. As a result, the logic that coordinates the activities of the different nodes towards a common goals has to be embedded in the network itself. In this scenario, the development of applications becomes more complex.

In this paper, we present a component-based framework that facilitates the development of sense-and-react applications promoting reuse of code. While applications components are used to implement basic functionalities (sense and reaction) our framework allows the specification of application-domain requirements. Our framework is composed of a Publish/Subscribe Broker, a component-based service layer and a Policy Manager. The broker manages subscriptions information and the service layer provides mechanisms orthogonal to publish/subscribe core (e.g., diffusion protocols, data communication protocols, data encryption, etc.). The novelty of our approach is the introduction of the Policy Manager where policies are enforced. Policies are rules that govern the choices and behaviour of the system. They can be used for specifying which services have to be associated with the broker operations. Moreover, policies can embed rules for coordinating the activities of the different sensors and actuators for reaching the common goals of applications.

1 Introduction

Early applications for WSN focused mainly on sensing the environment and sending the data to central sink devices with more computational power (e.g., PDAs and laptops) and therefore were able to coordinate the activities in the controlled environment. More recently, with the development of sensor nodes with more computational power and actuator nodes the *sense-and-react* application paradigm has emerged [1]. Sense-and-react applications are characterised by the fact that the data gathered by the sensing nodes can be used directly by actuator nodes that can react and change the sensed environment. Developing sense-and-react applications for WSNs is complicated by more complex interactions and the stringent limitations that characterise the devices where they are deployed. As a result, often applications developed for this scenario

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 212–229, 2008.

require ad hoc solutions optimised for specific environments. This compromises the flexibility, maintainability, and reusability of such applications [2,3].

Component-based software engineering (CBSE) can play a crucial role as it can be used for balancing the need of reusability with that of providing an efficient programming abstraction [4]. *Application components* encapsulate the functionality of the nodes. Components deployed on sensor nodes can be programmed for sensing data from the environment while components deployed on actuator nodes gather the data and react accordingly. However, to fully take advantage of CBSE it is necessary to provide a layer of abstraction to glue together the functionality of the different components. Components need to coordinate their functionality to achieve the global goals of the system without having their code tangled with details concerning OS and networking services. An effective solution to this problem needs to offer an appropriate level of abstraction to components without being too demanding in terms of resources. With its loosely-coupled, event-driven messaging services, the publish/subscribe paradigm offers to applications simple yet powerful primitives for communication.

Although the publish/subscribe paradigm is quite widely used, different aspects of its model concerning notification distribution, delivery and security can be implemented using different mechanisms. Each mechanism imposes requirements in terms of resources and each is more suited for specific class of applications. For instance, for certain critical applications it is acceptable to use a reliable delivery protocol even if it requires more resources in terms of energy consumption and computational overhead. Such aspects are not directly related to the basic functionality of the application and as such they can be referred to as *extra-functional concerns*. The *Separation of Concerns* (SoC) principle advocates that the basic functionality of an application should be specified in isolation from details regarding extra-functional concerns [5,6]. Because application code is not tangled with other details regarding extra-functional concerns, the code that results is less prone to errors, easier to maintain and far more reusable.

In this paper, we propose a component-based framework for programming WSNs realised through the publish/subscribe paradigm where extra-functional concerns are encapsulated in middleware components. Application developers specify which mechanisms have to be used in their applications in terms of *policies*. Policies are rules that govern the behaviour of a system and are an effective solution for separating the application functionality from low-level mechanisms [7]. Our framework supports an Event-State-Condition-Action Policy Environment (ESCAPE) where policies *connect* components orthogonally to the publish/subscribe paradigm. Policies define *stateful* interactions among components to *coordinate* their activities and reach system-wide goals.

The contributions of this paper are the followings. Our framework realises a flexible publish/subscribe system inasmuch as the needs of different application scenarios can be catered for by the different mechanisms that it can support. Application developers use policies to define which of these mechanisms are to be used. Because policies are defined outside the application code, application components (encapsulating application functionality) and middleware components

(encapsulating mechanism implementations) become the unit of reusability that can be deployed without modification in different scenarios. Moreover, policies represent also a unit of reusability. Once a specific behaviour is defined in a policy that policy can be deployed in other applications with similar characteristics. Finally, our framework is extendible since new mechanisms can be implemented as middleware components and deployed on the nodes.

The rest of this paper is organised as follows. In Section 2, we discuss the motivations and requirements of our approach. Section 3 provides a description of the architecture of our framework. Policy syntax and semantics are described in Section 4. To validate our approach, we present in Section 5 a case study and some of the policies used for its realisation. A brief evaluation of the implementation of our framework is presented in Section 6. In Section 7, we compare our approach to related research. We conclude by highlighting some future research direction in Section 8.

2 Motivations and Requirements

Sense-and-react applications represent a class of embedded control systems characterised by the realisation of a feedback-loop between a *sensing apparatus* and a *reacting apparatus*. Some examples of sense-and-react applications are heating, ventilation, and air conditioning (HVAC) [1], fire alarm systems, and burglar alarm systems. Nodes capable of sensing the environment provide readings of some parameters forming the sensing apparatus. Nodes equipped with actuators react to specific events and change the environment according to user preferences. As a software system, a sense-and-react application consists of two parts: the main *functionality* and the *control laws*. The main functionality represents the basic logic that is mapped into application components deployed in each sensor node. For instance, an application component deployed on a temperature node provides the functionality to obtain the readings from the node hardware and make it available to other nodes in the WSN. The control laws map the sensed data to specific actions that should comply with user preferences. When the functionality and control laws are not intertwined then it becomes possible to share the functionality of a node among different applications controlling the same environment. For instance, the functionality of a temperature node could be shared by both a HVAC and a fire emergency application. The two applications have different control laws however the functionality of the temperature node for both applications is the same, e.g. providing readings for the temperature of the environment.

From the above simple example, it emerges that the development of sense-and-react applications for WSNs is challenging not only for the constrains imposed by the physical devices but also for the complexity of the interactions that can be realised among different applications. In the following, we try to identify a set of requirements to define key aspects for facilitating the development of such applications.

Minimise functionality to maximise reuse. In our framework, the node functionality is encoded as an application component deployed on the node. In order

to maximise the reuse of such functionality, component functionality should be agnostic of the control laws enforced in the environment.

Coordination through middleware. WSNs can be seen as miniaturised distributed systems. The publish/subscribe model offers a very powerful abstraction for realising loosed-coupled distributed applications and middleware implementations have been already proposed in WSNs. In particular, the *reactive* style of interaction makes the model attractive for sense-and-react applications. Notifications sent by the sensing nodes can be used for trigger reactions by actuators. The underlying middleware is also the ideal place where the SoC principle should be realised. In particular, the middleware should provide to the application developers mechanisms that would allow the selection of different strategies implementing extra-functional concerns that can be subsequently enforced at runtime. For example, if a particular notification strategy is required, the middleware should offer such a strategy implemented as a component. Application developers specify which particular components have to be used with their applications in terms of policies. If necessary, new mechanisms can be developed and deployed as well, independent of the application functionality implemented by components.

Stateful policies. The control laws in sense-and-react applications typically represent *transitions* through different *states*. For instance, a sprinkler node that receives a temperature reading higher than a certain threshold has to check that smoke is detected before opening the water. This behaviour can be represented as two transitions: (i) from a normal state to a pre-alarm state when the temperature is above a safety threshold; and (ii) from a per-alarm state to an alarm state when the smoke detector provides a positive reading. To increase functionality reuse, control laws should be specified in isolation from application components. Policies represent ideal candidates for specifying the control laws of the system. In this case, policies need to be able to capture states and specify how transitions through different states must be executed.

Localised vs distributed computation. Sense-and-react applications are characterised by their capacity of reacting to stimuli coming from the surrounding environment. Because actuators can be in the proximity to where the data is generated, it is not necessary to flood the network with all readings. However, in certain case it is necessary that events have to be spread through the nodes present in the environment, such as fire alarms.

Multiple interaction patterns. Although reactive interactions characterised sense-and-react applications, there are still cases where *proactive interaction* should be preferred instead. This type of interaction is common in sense-only applications where data is proactively required by the consumers. In this way, it is possible to save the energy of the sensing nodes that are requested to generate the data only when it is needed.

In the following, we describe the architecture of our system to satisfy the identified requirements.

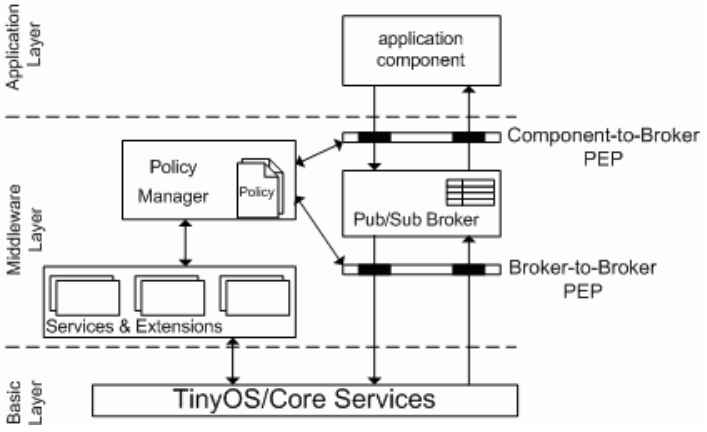


Fig. 1. Overview of our component-based architecture for a WSN node

3 Architecture

In this section, we discuss the main features of our approach. As shown in Figure 1, our approach presents a layered structure composed of an *application layer*, a *middleware layer* and a *basic layer*.

The application layer contains the basic functionality that is deployed on each sensor node. The functionality provided by a sensor node is encapsulated in application components. For instance, an application component deployed on a temperature sensor is responsible for providing temperature readings and acts as a publisher for this type of notification. Components deployed on actuator nodes are responsible for controlling and activating the actuator hardware according to the actual needs. In this case, actuator components act as subscribers of notifications representing the actual conditions of the environment.

The middleware layer consists of a **Publish/Subscribe Broker**, a **Policy Manager**, and a **Services & Extensions** described as follows.

Publish/Subscribe Broker. This module provides an API to the application layer and manages the subscription tables. In our approach, a notification is a tuple of (attribute,value)-pairs. A subscriber specifies its interest in a notification by issuing a `subscribe(notification)`. Although the subscriber cannot directly express constraints on the content of notification using the API, constraints can still be expressed in policies. For instance, if a subscriber should be notified only if the temperature value is higher than 50, then it is possible to write a policy that inspects the values of the temperature notifications and discards the notifications with values lower than 50 (more on this in Section 5). This decoupling of component functionality from subscription constraints increases the reusability of the components without sacrificing the expressivity of the publish/subscribe abstraction. In our framework content constraints are used for expressing control laws in the form of policies. A publisher advertises its notifications using

`advertise(notification)` and it publishes the data using `notify(notification)`. Subscription and advertisements can be withdraw using `unsubscribe(notification)` and `unadvertise(notification)`, respectively.

Policy Manager. One of the main features of our framework is that the publish/subscribe core is decoupled from mechanisms related to notification delivery, subscription distribution, and communication protocols. This design decision increases the flexibility of our approach inasmuch as our middleware is not bound to any specific mechanisms. Application developers can select the appropriate mechanisms that suit best their application needs. In contrast to the approach presented by Hauer et al. [8] where application components have to explicitly specify the mechanism to be used, in our framework components are completely agnostic of such specifications. Instead, we propose a policy-based approach where policies are used for such specifications. The enforcement of policies is done by a Policy Manager module. Policies can be specified to be enforced at specific points in our framework. Component-to-broker and broker-to-broker interactions are monitored via Policy Enforcement Points (PEP). Each time a message is sent through these interaction channels, the corresponding PEP intercepts the message and sends an event to the Policy Manager. The Policy Manager uses the events to trigger the policies available in its repository defined for that PEP (more on how policies are specified and enforced in Section 4). An important feature of our policy environment is that, the Policy Manager supports the deployment of new policies even during run-time without the need of taking the running application off-line. This feature increases flexibility of our framework and it makes particular appealing for WSN applications that required a high degree of availability.

Services & Extensions. This module provides hooks to the policy environment for invoking components that implement protocols and services outside the publish/subscribe core. Each service component is responsible for providing and consuming information required for fulfilling their tasks and if necessary to perform specific actions. Figure 2 shows some of the services components currently available. Components are organised in two sets: *Basic Pub/sub Components* and *Extension Components*. The Basic Pub/sub Components provide services that are necessary for realising the publish/subscribe paradigm and are described as follows:

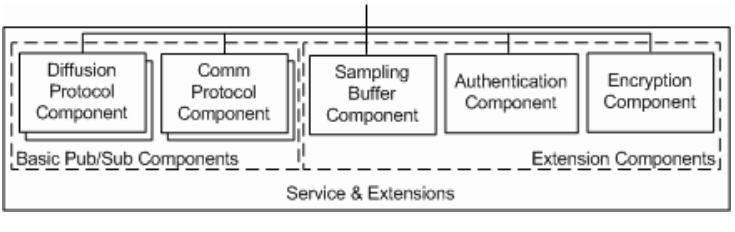


Fig. 2. A view of the Service & Extensions module

- A *Diffusion Protocol Component* (DPC) is responsible for routing data between publishers and subscribers. Initial work on diffusion protocols used a *two-phase pull* model [9], where subscriptions are distributed for the seeking of matching advertisements. Once a matching advertisement is found, the notifications are sent to the subscribers trying to find the best possible paths. This type of protocol is not suitable for all classes of application. In applications with many publishers that produce data only occasionally, the two-phase pull model is inefficient since it generates a lot of control traffic for keeping the delivery route updates. For this class of applications, the *push* diffusion protocol was proposed in [10]. According to the push diffusion protocol, the subscriptions are kept locally and the notifications seek subscribers. Other diffusion protocols have also be proposed, such as a *one-phase pull* protocol [10] (an optimised version of the two-phase pull), geographically scoped protocols [11], and rendezvous-based protocols [12,13]. Our current implementation provides a Push DCP (PsDCP) and a Pull DCP (PlDCP) implementing the push and one-phase pull protocols, respectively.
- A *Comm Protocol Component* (CPC) implements the delivery protocols of the messages generated by the publishers and subscribers with certain delivery guarantees. For instance, in certain cases subscriptions need to be updated frequently then a CPC that implements a probabilistic communication algorithm is acceptable. On the other hand, if an application requires a more reliable subscription distribution then a CPC that offers an algorithm that performs control traffic communication in the background can be used (at a higher costs in terms of resources). The former protocol is implemented by Probabilistic CPC (PCPC), while the latter is implemented by Reliable CPC (RCPC).

The Extension Components provide extra features to the paradigm. In the following we describe the components that have been implemented for the realisation of our case study discussed in Section 5.

- The *Sampling Buffer Component* (SBF) provides functionality for storing data samplings and computing certain predicates on the stored values. For instance, it could be used for calculating if a recent sampling differs more than a specified delta value from a stored sampling. Alternatively, it could just be used as a buffer that stores samplings frequently accessed or that requires a long time to be collected (i.e., audio signals).
- The *Authentication Component* and the *Encryption Component* are used for implementing *Trust Groups* of sensors. Trust groups are similar to secure multicasting groups [14]. Each trust group is associated to a secret key K_g. In this way, members of the same trust group are able to perform encryption and authentication within the group.The distribution of the K_g is done as an out-of-band bootstrapping process. For the encryption/decryption the component uses the Skipjack algorithm provided by the TinyOS core.

4 Event-State-Condition-Action Policy Environment (ESCAPE)

In this section we define the syntax and the semantic of our policy language. The syntax is defined by using a Backus-Naur form (BNF) while the semantic is described by defining the run-time behaviour of our policy manager.

```
<Policy> = policyName policyVariables <ESCAlist>

<ECSAlist> = "on" <Event> <SCAlist> | "on" <Event> <SCAlist> <ECSAlist>

<SCAlist> = currentState "-" newState "{"condition"}" "->" "{"action"}" <SCAlist>
 | currentState "-" newState "{"condition"}" "->" "{"action"}"

<Event> = <Qualifier> <pubSubEvent> | timeout <Timeout>

<Qualifier> = "B-C" | "C-B" | "B-extB" | "extB-B"

<pubSubEvent> = "notify(T)" | "advertise(T)" | "unadvertise(T)" | "unnotify(T)" |
"unpublish(T)" | "unsubscribe(T)"
```

Fig. 3. The syntax to our language for specifying ESCA policies

In Figure 3, we show our grammar used to define Event-State-Condition-Action (ESCA) policies. The notation $< symbol >$ is used define a non-terminal symbol, the character " is used to enclose language's keywords while words are terminals (i.e., symbols that never appear on the left side of a definition).

A policy is composed of a *policyName* followed by a *policyVariables* and an *ESCAlist*. The terminal *policyVariables* denotes variable declarations that can be used inside *condition* and *action* definitions. An *ESCAlist* is a list of event-state-condition-action each starting with the keyword **on** followed by an event and its state condition action list (i.e., SCAlist). A state-condition-action (SCA) is of the form *currentState-newState condition* $- >$ *action* where: (i) *currentState* is an integer that denotes the current policy state; (ii) *condition* is a predicate that must be true in order to apply the action *action*; (iii) *newState* is the new policy state after the action application.

In our approach, policies can be enforced when pub/sub operations are executed. Each of these operations is associated with a corresponding event of type **pub-sub** defined as following: $notify(T)$, $advertise(T)$, $subscribe(T)$, $unnotify(T)$, $unadvertise(T)$, $unsubscribe(T)$. T is represented by a tuple. With $T.x$ we denote the value of the parameter x in the tuple. For example, a policy defined on an event $subscribe(T)$ will be triggered each time the operation `subscribe(T)` is executed. However, our enforcement mechanism is able to capture the execution of an operation in 4 different points. This means that a pub-sub event associated to an operation can be generated in each of these points. In order to specify at which particular point a policy should be triggered, each pub-sub event must be always preceded by a qualifier that can be: (i) **C-B** specifying that the pub-sub event is sent from the component to its local broker; (ii) **B-C** specifying that the pub-sub event is sent from the local broker to a local component; (iii) **B-extB** specifying that a pub-sub event is sent from the local broker to an external one; (iv) **extB-B** defining an pub-sub event sent by

an external broker to the local one. Characterising a pub-sub event based on its local source and its local destination allows the description of flexible policy specifications. For instance policies can include component-broker interactions in order to filter pub-sub event content while broker-broker interactions allow the introduction of new features (e.g., security) without affecting the middleware basic mechanisms.

We also support **timeout** events for a node. A timeout event is of the form **timeout** t and is executed when for t seconds no event is observed. Generally speaking, a timeout is a way to perform actions when no pub-sub events are observed within a time interval.

Predicates can refer to event parameters, contain policy variables and invoke external libraries. Actions can modify policy variables, modify event parameters, execute any pub/sub operations (e.g., `notify(T)`, `subscribe(B)`, etc.) and call external libraries.

The action specification must always end either with the outcome **accept** or **discard**. Accept (discard) specifies that the pub/sub operation that triggered the policy must be completed (discarded) after the action execution terminates.

4.1 Policy Execution Model

In our model, the enforcement of policies is triggered by events. One event may trigger multiple policies at the same time. In the following, we define our policy execution model, that is the policy manager run-time behaviour. We denote with P the set of all policies and $p_1, \ldots, p_n$ are elements in P. A policy p in P is a set of events $\{e_1, \ldots, e_n\}$. Each event e has related a $SCAlist$ containing a sequence of elements of the form $(cs_i, ns_i, condition_i, action_i)$, representing the current state, the new state (after the action is executed), the condition, and action, respectively. In order to refer to one of these elements we prefix it with the event name followed by the symbol ".". For instance if e is an event then $e.condition_i$ and $e.action_i$ denote the action and the condition related to ith element in the $SCAlist$ of e.

Policies are executed by our policy manager that receives each event e and invokes the `execute` procedure, as shown in Figure 4. The `execute` procedure takes as an input an event e and defines a local list `Outcome` that will contain

```
1  void execute(event e)
2   Outcome[]={}; //the set of outcomes for each executed action
3   if no policy defines e then
4     accept;
5     return;
6   for each policy p that defines e do
7     let CS be the current state of the policy p
8     for i = 0 to p.e.SCAlist size do
9       if (p.e.cs_i == p.CS) and (p.e.condition_i) then
10        execute p.e.action_i;
11        add the outcome of p.e.action_i in Outcome[];
12        break;
13  select an outcome from Outcome[];
```

Fig. 4. Policy execution

the outcomes of all policies triggered by the event e. If no policy is triggered by the event e, then the operation that generated the event e is accepted (line 4) and the procedure terminates. On the other hand, for each policy p that defines e (line 6), then a $SCAlist$ element must be selected. An element in the $SCAlist$ is selected when the current state defined in the element ($p.e.cs_i$) is the same as the current state of the policy (CS) and the condition defined in the element ($p.e.condition$) is satisfied (line 9). In this case, the action corresponding to that element is executed and the outcome statement of the action (either accept or discard) is inserted in the `Outcome` set. When the actions of all policies defining the event e have been executed, the policy manager analyses the `Outcome` set. Because the same event may trigger several policies, the action outcomes that are inserted in the `Outcome` set are in conflict, i.e. both the statements *accept* and *discard* are present in the set. In our approach we allow policy writers to specify which of the two statements must be given priority per event. They can associate with each event e a default statement (either accept or discard) that is executed each time a conflict is detected. This choice allows us to have a fine-grain conflict resolution strategy at runtime that undertakes different statements for different types of events.

4.2 Tool and Policy Analysis

In this section, we describe the process that leads from policy definitions to code generation. This process is implemented by using two separate tools, i.e., an ESCA translator (shown in Figure 5) and the GOANNA tool [15] (shown in Figure 6). The ESCA translator parses each policy and translates it in a state-transition (a state machine). These state machines are input to the GOANNA tool that can show them in a graphical form, performs different semantic checks and generate the policy manager code. In the following we show how the ESCA translator translates a policy and we show the basic components of the GOANNA tool.

A translator checks that each policy p is syntactically correct and produces a state machine A_p. In particular for each event e the translator considers each state-condition-action (e.g, *currentState-newState condition* $- >$ *action*) and adds to A_p a transition that exits from the state *currentState*, is labelled with [*condition*] *e action* and enters in *newState*. In other words when the state machine is in the state current-state, the event e is observed and the condition is true than the state machine can move to *newState*. In Figure 6 we show the state machine related to the following policy:

```
1 TemperaturePolicy
2 on C-B advertise(Temperature,t_value)
3  0-1: true->{accept;}
4 on C-B notify(Temperature,t_value)
5  1-1: {t_value<50} -> {discard;}
```

State machines in this form can be loaded by the GOANNA tool that performs some semantic checks (possibly because of the state machine structure) and generates the policy manager code.

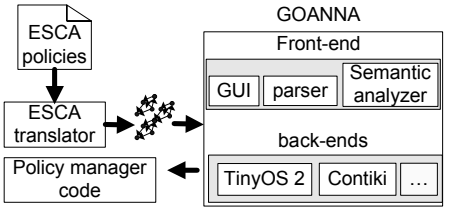


Fig. 5. Process of code generation

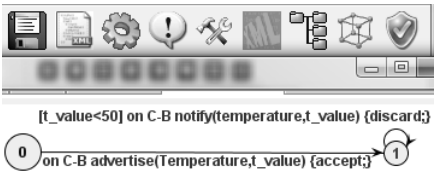


Fig. 6. GOANNA tool

GOANNA uses a front-end and a set of back-ends used for checks and code generation, respectively. The front end is composed of three main components: a *GUI*, a parser and a semantic analyser. The GUI implements a graphical tool to visualise the policy in a graphical form (i.e., a state machine based form). The parser and semantic controller take as input state machines and perform all syntactic and semantic checks, respectively. In the following we introduce the main semantic checks the tool performs, i.e., *state reachability*, *correct sequence* and *recursive event detection*.

State reachability ensures that each event-state-condition-action can be applied, i.e., all states inside a state machine definition can be reached. Correct sequence analyses the state machine definition and verifies correct ordering among system events, e.g., a notify of an event is preceded by a publication. Recursive event detection avoids policies leading to livelock. In the simplest case we can have a policy in which an event e is defined, the new state is equal to the current one (a transition that enters and exits in the same state) and its action define a notify of the same event e. In this case the policy can generate an infinite number of events e without making any progress. Generally speaking a policy can defines a chain of events that produces livelock. For instance a policy can define the events e_1 and e_2 and the state machine is such that: (i) e_1 changes the state from q_1 to q_2 and its action generates an event e_2; (ii) e_2 changes the policy state from q_2 to q_1 and its action generates the event e_1. Our tool tries to visit each state machine, detect possible livelock conditions and produce warnings. As future work we are adapting other well known state machines verifications to our particular context. For instance we are planning to apply checks defined over several state machines by defining composition among state machines and performing checks on it.

5 Case Study

This section presents a case study related to a cultural asset transportation service used to securely move cultural assets from one venue (museum) to another. The service was developed as part of the EU CUSPIS project [16].

In the transportation service, a lorry transports a set of packages each containing a cultural asset. As shown in Figure 7, the lorry is equipped with sensors and actuators on which several sense-and-react applications are deployed.

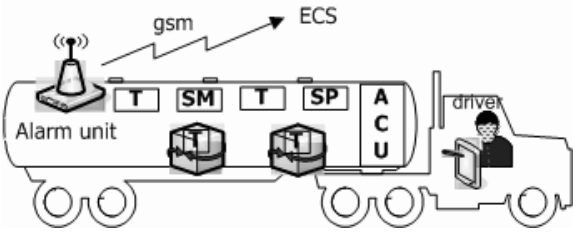


Fig. 7. An overview of the deployment of the sensors and actuators for the CUSPIS case study

During transportation, the lorry is monitored by the Emergency Central Station (ECS) that is in contact with police and emergency units (such as fire fighter stations). To send alarms to the ECS, the lorry is equipped with multiple alarm units that include a GSM transmitter and GPS sensor. The lorry driver can use a portable wireless computer, such as a PDA, to check sensor readings and be notified in case of any alarms.

The following sense-and-react application are deployed:

- *Fire Alarm Application* is responsible for detecting and taking initial actions against the fire inside the lorry. Temperature sensors provide readings for the actual temperature and smoke detectors are used for sensing the presence of smoke. If the temperature rises over a given threshold and the smoke detectors provide positive smoke readings then the water sprinklers must be activated and a fire alarm sent. Issuing a fire alarm activates the alarm unit that informs the lorry driver through the PDA and sends an alarm message to the ECS.
- *Air Conditioning Application* is responsible for maintaining temperature and humidity within the lorry to given values. Temperature sensors (shared with the Fire Alarm Application) and humidity sensors provide reading of the air quality in the lorry. An air conditioning unit (ACU) uses the readings from the sensors to increase or decrease the temperature and humidity to keep those values within the target values set by the driver.
- *Package Tampering Monitor* is responsible for the integrity of the packages containing the artifacts and to raise an alarm in case the packages are tampered with. Each package contains sensors that collect readings for temperature, humidity and light. An indication that a package was opened can be signaled when a reading deviates significantly from the previous values. For instance, when the package is opened the amount of light and temperature inside the package increases and such variation can be captured by the sensor. If this is the case, then the sensor notifies the driver's PDA and the alarm unit. The latter sends an alarm (together with the GPS position) to the ECS to summon the intervention of the police.

In the following, we discuss the policies used for specifying the control laws of the applications. For brevity reasons, we cannot present the complete set of

```
1  global target_t, delta_t;
2  TemperaturePolicy
3    on C-B notify((Temperature,t_value))
4     0-0: {t_value<50 && !SBC.deviateOrZero(target_t, delta_t,t_value)}
5                      -> {discard;}
6    on B-extB notify((Temperature,t_value))
7     0-1: {t_value>50} -> {PsDPC.notify((Temperature,t_value));
8                           RCPC.notify((Temperature,t_value));
9                           accept;}
10   on B-extB notify((Temperature,t_value))
11    0-0:  {SBC.deviateOrZero(target_t, delta_t,t_value)}
12                     -> {PlDPC.notify((Temperature,t_value));
13                     PCPC.notify((Temperature,t_value));}
14                     accept;
```

Fig. 8. The temperature policy defining the control laws for the Fire Alarm and Air Conditioning Applications

policies used for the case study but we have to limit our discussion to the most significant ones. We assume that to avoid the injection in the system of notifications generated from sensors outside the lorry, all the message exchanged use the authentication and encryption components for a trust group communication.

Content-based filtering. The application component deployed on the temperature sensors provides readings of the temperature inside the lorry. Each component acts as a publisher of the notification type `(Temperature,t_value)`. This type of notification is shared by the Fire Alarm and Air Conditioning Applications. Instead of flooding the network with every temperature sampling, only notifications with meaningful values should be allowed to leave the publisher node. In particular, for the Fire Alarm Application, only samplings with values over 50 should be allowed. For the Air Conditioning Application a different approach is used: a notification is published if the difference between the actual value is either more than delta from a target value or it is equal to zero. In the first case, this means that the Air Conditioning Unit (ACU) has to be activated to bring the temperature within the desired target; while in the second case the ACU can be switched off since the desired target is reached. This content-based filtering can be specified by a policy as shown in Figure 8. When the component sends a notification (line 3), the policy checks whether the value of the temperature is less than 50 and that the predicate `deviateOrZero`, provided by the Sampling Buffer Component (SBC), is not satisfied (line 4). In this case the notification is discarded.

However, when the notifications have to be delivered, two different delivery protocols must be used for the two applications. For the Fire Alarm Application, the notification should be spread as quickly and reliably as possible. In this case, the notifications are associated with the diffusion protocol component that implements the push model (PsDPC) using a reliable communication protocol component (RCPC) (line 6-9). On the other hand, for the delivery of the temperature notifications for the Air Conditioning Application (11-14) the pull model with the probabilistic communication protocol is used (implemented by the PlDPC and PCPC, respectively).

```
1  TamperingPolicy
2   on C-B advertise((PackageTemperature,t_value))
3    0-1: true -> {advertise((PackageAlarm));
4                  subscribe((PackageTemperature,t_value));
5                  accept;}
6   on C-B notify((PackageTemperature, t_value))
7    1-1: {!deviateDelta(t_value, delta_value)} -> {discard;}
8    1-2: {deviateDelta(t_value, delta_value)} -> {startTimer(3);
9                                                  accept;}
10  on B-C notify ((PackageTemperature, t_value))
11   2-2: {notification_id == this.node_id} -> {discard; \\ignore: this is my
         notification}
12   2-1: {node_id != this.node_id} -> {discard; \\false alarm: increase in
         temp in other sensors}
13  on timeout()
14   2-3: true -> {notify((PackageAlarm));
15                 discard;}
```

Fig. 9. The policy for setting off alarm notifications when the packages are tampered with

Localised computation. The Package Tampering Monitor is responsible for monitoring the integrity of the packages containing the artifacts and for raising an alarm in case the packages are tampered with. Each package contains a sensor that raises an alarm if the readings for temperature, humidity and light drastically change. For instance, when the package is opened the amount of light and temperature inside the package increases and such changes are can be captured by the sensor. However, care must be taken to avoid notification of false alarms. For instance, if the temperature in the package increases it could be an effect due to the increase of temperature inside the lorry (i.e., the air conditioning unit is not working properly). If this is the case, then the sensors in the other packages also register an increase in temperature. Therefore, before sending the alarm notification, the sensor that first registers an increase of temperature sets off a timer and waits for notifications from the sensors in other packages that signify an increase in temperature. If these notifications from other sensors arrive before the timer timeouts then no alarm is sent. Otherwise, the alarm notification is sent.

This behaviour can be codified using a policy as shown in Figure 9 (note that to improve readability we removed all details related to diffusion and communication protocols). This policy captures only the case for variations in temperature readings. The policy starts registering the node as a publisher for the `PackageAlarm` notification and as subscriber of the `PackageTemperature,t_value` notifications. The publishing of the temperature readings uses a "send-on-delta" approach, where a notification is published only if varies more than a given delta from the previous published notification. The predicate `deviateDelta` is used for checking whether the difference between the actual reading and the previous published one is grater than delta. If the difference in not more than delta, the notification is blocked (line 7). Otherwise, if the notification deviates more than delta, the notification is sent and a 3 second timer is started (line 8). At this stage, the following can happen:

Component type	Description	Code (bytes)	Data (bytes)
Temperature Component	Senses the temperature	4530	40
Broker	Maintains table	1234	21
Policy Manager	Instrumentation code	1120	58
Tampering policy	the policy code	870	12
PsDPC	Push diffusion	680	55
PlDCP	Pull diffusion	730	90

Fig. 10. Code and data size information

- a notification arrives but is the one that was just sent by the node itself (line 11). In this case, the state of the policy is not changed.
- a notification arrives from other nodes (line 12). This means that the increase of temperature is not local to this package but other sensors are registering it as well. Therefore this is a false alarm and should be ignored.
- the timer expires and a timeout event is sent (line 13). This means that no other sensors registered the increase in temperature. In this case a `PackageAlarm` notification is sent (line 14).

6 Implementation

We have used our approach to implement the requirements of our CUSPIS application. We have used the TinyOS operating system running on *Tmote Sky* motes. In particular, we have built basic monitoring components that only sense environmental data (i.e., temperature, smoke, light) and have added our framework to build CUSPIS functionalities.

In Figure 10 we show information about the code / data size of both application components and our framework. We emphasise that the policy manager size is independent from policy specifications and all other components. In other words the policy manager is a container that manages the policy life cycle (i.e., it loads, executes and deletes policies) so that its size is always the same. In our case the temperature components is bigger than the tampering policy since the temperature component must embed all code needed to sense and to manage the timer (the sensing is performed at each tick) while the policy only embeds few if statement and few variables to implement the state machines.

7 Related Work

The TeenyLIME middleware is specifically designed to address the requirements of sense-and-react applications for WSNs. TeenyLIME provides a programming model based on the tuple space paradigm. The tuple spaces in TeenyLIME represents a shared memory that is shared among sensors within a one-hop region. Although the TeenyLIME offers a simple but powerful abstraction, it lacks the flexibility of our approach. In fact, extra-functional mechanisms that are

provided with the middleware are fixed to specific hard-coded modules. For instance, in TeenyLIME tuples are distributed according to a communication protocol that supports only one-hop communications. In our case, notifications can be distributed using several diffusion protocols, according to the needs of the applications.

TinyCOPS [8] is a publish/subscribe middleware that uses a component-based architecture for decoupling the publish/subscribe core from choices regarding communication protocols and subscription and notification delivery mechanisms. The middleware can be extended with components that provide additional services (i.e., caching of notifications, extra routing information, etc.). The specification of which particular mechanism has to be used is done by means of metadata information that the application components have to provide through the publish/subscribe API. In our case, application components are agnostic of such extra-functional concerns since policies are used for specifying which mechanisms have to be used.

The Mires middleware [18] is also a publish/subscribe service that uses the component architecture of TinyOS 1.x. Like our approach, it uses a topic-based naming scheme. However, differently than in Mires, we can support content-based filtering by means of policies. Although in Mires it is possible to introduce new services (like aggregation) using extension components, the choice of the communication protocols is fixed.

MiLAN [19] is a middleware for WSNs that provides application QoS adaptation at run-time. The middleware continuously tracks the application needs and optimises network usage and sensor stacks for an efficient use of the energy. As such, MiLAN focuses more on a class of resource-rich wireless networks that can support well the impact of the monitoring overhead. In our approach, we concentrate more on sensors with limited resources, where optimisations are mainly performed at compile-time.

8 Conclusions and Future Work

In this paper, we have described a component-based framework for WSNs based on publish/subscribe paradigm where ESCA policies can be enforced. Component applications implement the basic functionality of the wireless nodes (sense and reaction capabilities) while policies implement all extra-functionalities that are domain specific. Policies are specified using our state machine language that includes variables and libraries in order to define complex policies. Policies are input in our tool that performs semantic checks and generate all needed code to execute them. We have applied our approach to a case study where a sensor network is deployed in lorries that transport cultural assets between museums. The system has been developed for the TinyOS2 operating system.

Our future work aims to optimise the code generation for TinyOS2 and to carry out a detailed evaluation of the run-time costs. Another direction to explore is to provide to application developers means for selecting the mechanisms that suit best the needs of their applications. Moreover, as the needs of the

applications changes after deployment, an autonomic approach that would select the best mechanisms for the actual needs of the application would be ideal. However, such an approach requires continuously monitoring of the activities that would incur in some overhead. In our future work, we want to study whether the monitoring and adapting overhead is sustainable compared to the overall gain in performance.

Acknowledgments

This research was supported by the UK EPSRC, research grants EP/D076633/1 (UBIVAL) and EP/C537181/1 (CAREGRID). The authors would like to thank our UBIVAL and CAREGRID collaborators and members of the Policy Research Group at Imperial College for their support.

References

1. Deshpande, A., Guestrin, C., Madden, S.: Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28(1) (2005)
2. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Transactions on Networking (TON) 11(1) (2003)
3. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: An acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1) (2005)
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: ASPL 2000. Proc. of the ninth international conference on Architectural support for programming languages and operating systems (2000)
5. Dijkstra, E.W.: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer, Heidelberg (1982)
6. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
7. Sloman, M., Magee, J., Twidle, K., Kramer, J.: An Architecture for Managing Distributed Systems. In: Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 40–46 (1993)
8. Hauer, J., Handziski, V., Kopke, A., Willig, A., Wolisz, A.: A Component Framework for Content-Based Publish/Subscribe in Sensor Networks. Wireless Sensor Networks, pp. 369–385 (2008)
9. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, Boston, MA, USA, August 2000, pp. 56–67. ACM, New York (2000)
10. Heidemann, J., Silva, F., Estrin, D.: Matching data dissemination algorithms to application requirements. In: SenSys 2003. Proc. of the 1st international conference on Embedded networked sensor systems, New York, USA (2003)
11. Yu, Y., Govindan, R., Estrin, D.: Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report TR-01-0023, University of California, Los Angeles, Computer Science Department (2001)

12. Braginsky, D., Estrin, D.: Rumor routing algorithm for sensor networks. In: Proceedings of the First ACM Workshop on Sensor Networks and Applications, Atlanta, GA, USA, October 2002, pp. 22–31. ACM, New York (2002)
13. Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., Shenker, S.: GHT: A geographic hash table for data-centric storage. In: Proceedings of the ACM Workshop on Sensor Networks and Applications, Atlanta, Georgia, USA, September 2002, pp. 78–87. ACM, New York (2002)
14. Rafaeli, S., Hutchison, D.: A Survey of Key Management for Secure Group Communication. ACM Computing Surveys 35(3), 309–329 (2003)
15. Mostarda, L., Dulay, N.: GOANNA: State machine monitors for sensor systems (2008), www.doc.ic.ac.uk/~lmostard/goanna
16. European Commision 6th Framework Program - 2nd Call Galileo Joint Undertaking. Cultural Heritage Space Identification System (CUSPIS) (2007), http://www.cuspis-project.info
17. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming Wireless Sensor Networks with the TeenyLIME Middleware. In: Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007), Newport Beach, CA, USA, November 26–30 (2007)
18. Souto, E., Guimares, S., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: A publish/subscribe middleware for sensor networks. Personal Ubiquitous Comput. 10(1) (2005)
19. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to support sensor network applications. IEEE Network 18(1) (2004)

Experiences from Developing a Component Technology Agnostic Adaptation Framework

Eli Gjørven[1], Frank Eliassen[1,2], and Romain Rouvoy[2]

[1] Simula Research Laboratory,
P.O.Box 134, 1325 Lysaker, Norway
eligj@simula.no
[2] University of Oslo, Dept. of Informatics,
P.O.Box 1080 Blindern, 0314 Oslo, Norway
frank@ifi.uio.no, rouvoy@ifi.uio.no

Abstract. Systems are increasingly expected to adapt themselves to changing requirements and environmental situations with minimum user interactions. A challenge for self-adaptation is the increasing heterogeneity of applications and services, integrating multiple systems implemented in different platform and language technologies. In order to cope with this heterogeneity, self-adaptive systems need to support the integration of various technologies, allowing the target adaptive system to be built from subsystems realized with different implementation technologies. In this paper, we argue that state-of-the adaptation frameworks do not lend themselves to ease technology integration and exploitation of advanced features and opportunities offered by different implementation technologies. We present the QUA adaptation framework and its support for technology integration and exploitation. Unlike other adaptation frameworks the adaptation framework of QUA is able to exploit a wide range of adaptation mechanisms and technologies, without modification to the adaptation framework itself. As a demonstration of this property of QUA, we describe the integration of an advanced component model technology, the FRACTAL component model, with the QUA framework. Our experience from this exercise shows that the QUA adaptation framework indeed allows integration of advanced implementation technologies with moderate effort.

1 Introduction

Increasingly dynamic computing environments require software developers to support a wider range of technologies with applications that need to handle continually evolving situations and environments. Well designed *component models* enforce *separation of concerns*, thus relieving application developers from having to address concerns, such as extensibility, distribution, and reconfiguration of the application, and letting them focus on business and application logic. In order to ease the tasks of system developers and administrators, separation of concerns can be supported by a *generic adaptation framework* for handling self-adaptation of applications and services [1,2,3]. Self-adaptation includes the ability to self-configure automatically and seamlessly according to higher-level policies. By the same approach, the application developer can model a set of components and their non-functional properties, and leave it to an underlying

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 230–245, 2008.

middleware to reason about changes in context and how these changes should impact and possibly reconfigure the application components to provide the optimal end-user satisfaction with the service. This way, adaptive behavior is developed separately from the application business logic.

However, a challenge for self-adaptation is the increasing heterogeneity of applications and services, integrating multiple systems implemented in different platform and language technologies [4,5,6]. In order to cope with this heterogeneity, self-adaptive systems need to support *technology integration*, which is the process of building a system from subsystems technologies. Successful technology integration includes overcoming three challenges: *i)* ensure that the *integrated subsystems are able to interoperate safely*, *ii) integrate into the adaptation framework the different technologies* used in the system to be adapted, and *iii)* whenever possible *exploit the specific features and opportunities* offered by the different implementation technologies used. The latter is generally preferable as it will reduce duplication of efforts when using advanced implementation technologies, such as state-of-the-art component platforms. Recently, much effort has been spent on interoperability, in particular in the area of *Service-Oriented Architectures* (SOA) [7] and web-services composition [8,9]. However, SOA focus on solving the first problem by standardizing the interaction between services, thus hiding the service implementation platforms. Consequently, SOA does not facilitate the exploitation of adaptation-related features that service implementation platforms may provide. Thus, adaptation techniques are limited to the specification and orchestration of workflows through dedicated languages and engines.

In this paper, we focus on the second and third problems, namely *technology integration* and *exploitation*, in the context of self-adaptive systems. In order to fully support technological heterogeneity, self-adaptive systems must support integration and exploitation also of adaptation-related technologies, while ensuring that the resulting system as a whole performs as expected. In order to be applicable to applications and adaptation mechanisms implemented with different technologies, the adaptation framework of a self-adaptive system needs to be *technology agnostic*. It must be able to adapt the behavior of applications and services without depending on knowledge of particular adaptation mechanisms, and application implementation technologies. In contrast, current adaptation frameworks are bound to particular adaptation mechanism technologies, such as component models, middleware, or communication infrastructures [2,3]. Integrating new technologies into these adaptation frameworks may require major changes to be made both to the integrated systems and the framework itself. Furthermore, the resulting system may not be able to exploit the specific capabilities of the integrated technology. Such a tight coupling between the adaptation framework and the adaptation mechanisms does not facilitate an easy technology integration.

This paper describes the QuA adaptation framework and its support for technology integration and exploitation. As a demonstration of the latter, we describe our experience with integrating an advanced component model technology, the Fractal component model [10], with the QuA framework. Unlike other adaptation frameworks, the adaptation framework of QuA is able to exploit a wide range of adaptation mechanisms and technologies, without modification to the adaptation framework itself. In order to establish a clear separation between the adaptation framework and the adaptation

mechanisms, we apply the *Dependency Inversion Principle* [11] to the QUA architecture. Under this principle, higher level policies do not depend on the modules implementing the policies, but rather on abstractions. Specifically, by expressing adaptation policies as *utility functions* [12], we enable the specification of adaptation policies that are independent from the technologies used to implement the adaptation actions enforcing the policies. Furthermore, rather than defining yet another component model, the QUA adaptation framework defines a concise, technology-agnostic, meta-model that abstracts over the various legacy component models, which can be plugged in to the adaptation framework.

In the remaining of the paper, we first study the requirements that self-adaptive systems must satisfy in order to facilitate easy technology integration, and we introduce the design principles that support these requirements (cf. Section 2). These principles are demonstrated through the QUA adaptation framework design (cf. Section 3), and the integration of the FRACTAL component model (cf. Section 4). We discuss the experiences made from this integration (cf. Section 5) before concluding and presenting our future work (cf. Section 6).

2 Technology Integration and Adaptation Frameworks

This section analyzes the challenges of designing an adaptation framework supporting technology integration and further motivates the need for clearly separating the adaptation concerns. Then, the main design principles adopted for achieving a better separation of adaptation concerns in the QUA framework are introduced.

2.1 Limitation of Technology Integrations in Self-adaptive Systems

Conceptually, a self-adaptive system consists of three parts: the *adaptation framework*, the *adaptation mechanisms*, and the *target adaptive system*.

The *adaptation framework* (also known as control loop) is responsible for controlling the ongoing adaptation processes. The adaptation framework constantly observes and analyzes the behavior of the target adaptive system, and instantiates, plans, and executes adaptations when necessary. The adaptation framework is based on adaptation policies, used to decide which adaptation to carry out in each situation. The adaptation framework depends on *adaptation mechanisms*, which perform adaptation related actions, such as collecting and processing information about the *target adaptive system* and its environment, evaluating alternative adaptation actions, and performing the selected ones. Examples of such mechanisms are context monitoring, component lifecycle handling, and reconfiguration mechanisms.

The *target adaptive system* represents the target of adaptation. The adaptive system spectrum covers application software, middleware infrastructure (*e.g.*, communication, transaction, persistence), lower level operating system modules (*e.g.*, scheduler, driver), or device resources (*e.g.*, screen resolution, network interface).

The current direction in self-adaptive software research is to isolate the adaptation concerns from the application logic using generic adaptation frameworks [2,3]. However, state-of-the-art adaptation frameworks and corresponding adaptation policy

specification languages are tailored to specific component models and platforms. The adaptation policy languages, such as SAFRAN [2] or PLASTIK [3], can be used to define both coarse-grained adaptations, such as replacing one component with another, and more technical and fine-grained adaptations, down to the level of setting the value of a component parameter. These adaptation frameworks impose a tight coupling between the adaptation policies—stating what adaptations should be carried out and when—and the adaptation mechanisms—implementing the corresponding adaptation actions. Typically, the adaptation policy refers directly to the adaptation actions themselves.

Actually, the integration of a new technology can have the following impacts:

→ integration of adaptive systems requires porting the target adaptive application or service to the technology platform of the adaptation framework, and to integrate associated adaptation mechanisms into the framework;
→ integration of new adaptation mechanisms requires updating the adaptation framework with knowledge about the new mechanisms;
→ updating the adaptation framework requires careful evaluation of the effects that the updates will have on other mechanisms and adaptive systems controlled by the adaptive systems.

Thus, a possible, and unfortunate, consequence of the above dependencies may be that adding a new component to a target adaptive system requires updating the higher level adaptation policies. In order to overcome the above challenge, design principles for building technology-agnostic adaptation frameworks are needed. Technology agnostic adaptation frameworks preserve the technological heterogeneity of the target systems, while exploiting adaptation-related features provided by their implementation platforms. We argue that to achieve the above, separation of adaptation concerns should be enforced when designing and implementing the adaptation behavior. In particular, by handling the three parts as separate concerns, we are able to reduce the dependencies between them, and thereby facilitate the integration of new solutions in each concern with less impact on the others.

2.2 Providing a Clear Separation of Adaptation Concerns

In the area of agile programming, the *Dependency Inversion Principle* (DIP) has been introduced as a fundamental design principle, which contributes to improving software maintainability and extendability [11]. The DIP can be applied to systems where higher level modules, containing the important policy decisions and business models of an application, controls lower level modules, containing the implementation of the higher level policies. Thus, according to this principle:

a) high level modules should not depend on low level modules. Both should depend upon abstractions;
b) abstractions should not depend upon details. Details should depend upon abstractions.

We apply the DIP to the case of an adaptation middleware consisting of a higher level module, the adaptation framework containing the adaptation policies and lower level modules, containing the adaptation mechanisms.

In [11], the author points out two consequences of applying the DIP. The first, and most obvious, consequence is that no implementation class should depend on another implementation class, but rather on abstractions. The second consequence is that the abstractions should be owned by the higher level policies, rather than the lower level implementations. From this, we formulate the following requirements for the design of the adaptation framework:

1. the *adaptation framework and the adaptation mechanism should depend on adaptation mechanism abstractions* (according to DIP *a)*),
2. the *adaptation mechanisms and the adaptation target should depend on adaptation target abstractions* (according to DIP *a)*),
3. the *adaptation mechanism abstraction should be owned by the adaptation framework* rather than the mechanisms (according to DIP *b)*),
4. the *adaptation target abstraction should be owned by the adaptation mechanism*, rather than the adapted system (according to DIP *b)*).

Figure 1 illustrates the design of an adaptation framework that satisfies the DIP principle. The modelling convention used here, and in the rest of the paper, is that closed arrows represent an implementation relationship from the implementing class to the interface, while open arrows represent associations and dependencies.

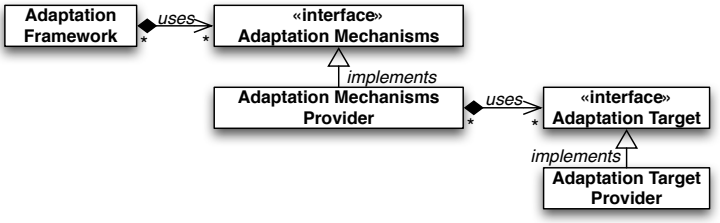


Fig. 1. Applying dependency injection principle to adaptation frameworks

Many adaptation frameworks satisfy the requirements 1, 2, and 4 [2,3,13]. They typically define adaptation-related interfaces that must be implemented by target systems in order to conform to the adaptation mechanisms. However, as discussed in Section 2.1, these frameworks use adaptation policies that tightly couple the adaptation framework and the mechanisms, making the adaptation mechanism abstraction not truly owned by the adaptation framework as specified by requirement 3. Below, we discuss how to design adaptation policies, making possible to fully satisfy the DIP.

2.3 Using Technology-Independent Adaptation Policies

The adaptation framework depends on an adaptation policy, which is applied to decide which mechanism to use in a certain situation. Many adaptation frameworks are based on variants of *rule-based adaptation policies* [2,3], where policies are specified using condition-action expressions. Rule-based approaches can be simple and practical, at least as long as the rule-set is small. However, adaptation rules do not separate

well between the adaptation framework and adaptation mechanism. Rules map adaptation conditions directly to detailed knowledge about the target adaptive system, and the available adaptation mechanisms. When integrating new adaptation mechanisms into the rule-set, at best, new rules have to be added to the rule-set. In order to keep the rule-set consistent, then the entire rule-set has to be checked for completeness (all conditions map to an action) and conflicts (conditions mapping to multiple actions that are contradictory). At worst, the policy language is not expressive enough for the new mechanism. For example, the policy language designed for supporting the capabilities of a given component model, may not be directly applicable to another component model. The essential problem is that the rule-based policy languages are owned by the mechanisms, producing dependencies that are difficult to handle when integrating new technologies.

Utility-based adaptation policies have been elaborated as an alternative to rule-based policies in self-managing systems [12]. Utility-based policies are expressed as functions assigning to each configuration alternative—including adaptation mechanisms necessary to implement the alternative—a scalar value indicating the desirability of this alternative. A utility-based adaptation framework discovers a set of configuration alternatives, computes their utility, and selects the one with the highest utility. This way, utility functions introduce a level of indirection between the adapted system and the mechanisms implementing a configuration, and its desirability. The utility value is calculated from metadata describing the functional and qualitative properties of a configuration, rather than the technical implementation knowledge. Thus, utility functions provide a higher-level, mechanism and technology independent adaptation policy language. The reader can refer to [14] for a detailed discussion about the characteristics of utility functions.

2.4 Reflecting the Target Adaptive System Properties

In order to be able to compute utility values, metadata about the functional and qualitative properties of configuration alternatives and adaptation mechanisms must be available to the adaptation framework. Thus, the adaptation framework depends on a technologically independent meta-model that is able to express information about the required properties.

In order for the adaptation framework to be independent of the existence of particular reflective capabilities provided by target technologies, the metadata must be provided by a separate module. A variant of traditional reflection, called *mirror-based reflection* [15], can be used to define reflective APIs suitable for technologically independent adaptation frameworks. In mirror-based reflection, the reflective capabilities are provided by separate objects called *mirrors*, instead of by the reflected objects themselves, as is common in traditional reflection. The reader can refer to [15] for a detailed discussion about the characteristics of mirror-based reflection.

3 Designing the QuA Adaptation Framework

This section introduces the design of the QuA adaptation framework, which proposes to improve the state-of-the-art adaptation middleware approaches by offering a modular support for reflecting, reasoning, and deploying services.

3.1 An Overview of the QuA Middleware

The QuA middleware supports middleware-managed adaptation, which means that the adapted system is specified by its behavior, and then *planned*, *instantiated*, and *maintained* by the middleware in such a way that its functional and qualitative requirements are satisfied throughout its lifespan. In order to be able to represent the adapted system from specification to termination, the unit of adaptation in QuA is a *service*, which we define as:

> A service describes a set of capabilities that are defined by *i)* a group of *operations* and their *input and output data*, and *ii)* a *contract* (explicit or implicit) describing the work done, as delivered output data, when invoking these operations with valid input data. The *service lifespan* encloses its specification of behavior, association with implementation artifacts, service instantiation, execution, and termination.

Thus, a service may be associated with implementation artifacts implementing its behavior, or running objects performing its behavior. Service implementation artifacts always require a particular *service platform*, which can be used to instantiate a service by interpreting the implementation artifacts. Finally, service implementations may depend on other services in order to implement the promised functionality.

A QuA client is typically a client application, using QuA to instantiate services, or service a development tool, using QuA to deploy service implementations and meta-data. QuA defines a programming API that can be used to invoke the QuA middleware services from tools or applications, and providing the following operations:

- *Publication of service implementations*: service implementations may include different types of implementation resources, such as implementation classes (Java classes or library modules), component descriptors (ADL or XML documents), interface definitions etc., depending on the type of technology used to implement the service.
- *Advertising service implementation meta-data*: Meta-data describing the static and dynamic properties of service implementations can be advertised to the middleware.
- *Instantiation of services*: service instantiation means evaluating, selecting, and instantiating service implementations, and perform initial service configuration. The resulting service will be maintained by the QuA middleware throughout its lifespan through adaptation.
- *Reflection on services*: the QuA middleware defines a reflective API, called the *Service Meta Object Protocol* (SMOP), used to inspect and manipulates services.

In contrast to other adaptation frameworks, which mixes the adaptation policies with the adaptation mechanisms, QuA identifies a clear separation between the three adaptation concerns described in Section 2.

Conceptually, we order the three adaptation concerns horizontally, as depicted in Figure 2. By applying the *Dependency Injection Principle* (DIP), we achieve an horizontal separation of concerns by establishing an ordering of module pairs where the higher level module always owns the interfaces shared with next lower-level modules.

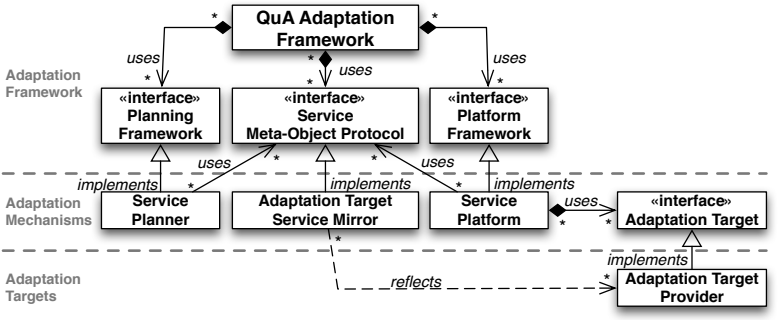


Fig. 2. Design of the QuA adaptation framework

The adaptation framework module define three abstractions; The Planning Framework is responsible for selecting service implementations, while the Platform Framework is responsible for managing service implementations during their execution. The Service Meta-Object Protocol (SMOP) can be used to inspect and manipulate services throughout their lifespan.

The planning and platform frameworks abstractions are implemented by concrete planning and implementation mechanisms. The planning framework is implemented by Service Planners that use metadata provided by the SMOP to find alternative service implementations, analyze their expected behavior, and select an alternative that match the service requirements. The platform framework is implemented by Service Platforms that enclose technology specific code and mechanisms supporting service instantiation and adaptation, including binding and rebinding of service dependencies. Such adaptation mechanisms typically define adaptation-related interfaces implemented by the target systems. Service Platforms are responsible for maintaining the causal connection between Service Mirrors implementing the SMOP, and the Adaptation Targets. In the adaptation target layer, we find the Adaptation Target Providers, which are the base level objects implementing the adaptation targets.

3.2 Reasoning Support: The Utility-Based Planning Framework

The planning framework applies *utility-based adaptation policies* [12] as a way to keep the adaptation framework independent of integrated technologies and mechanisms. Each service may be associated with a *utility function*, which is applied by the planning algorithm to metadata describing the qualitative properties of each alternative service implementations. Metadata about the qualitative properties of a service implementation can be expressed by *quality predictors*, which are functions of the run-time environment, and the quality provided by other services that the service depend on, if any. Such predictor functions are written by the implementation developer, and made available through the SMOP. By computing utility functions and quality predictors, the utility of a particular implementation can be calculated based on the desirability of alternative behaviors, rather than knowledge about the alternative implementations and mechanisms.

The planning process can be implemented by numerous algorithms. By applying the DIP also to the planning framework, the adaptation framework is protected from changes in the mechanisms used by the planning framework.

3.3 Technology Support: The Platform Framework

When an implementation has been selected by the planning framework, the platform framework is responsible for applying the correct mechanisms for instantiating the service. A service platform is able to interpret implementation artifacts of certain types, instantiate services from those artifacts, and provide a run-time environment for the instantiated services. For example, a Java Service Platform provides access to a *Java Virtual Machine*, and is able to instantiate Java objects hosted by that machine, from Java classes. The platform also defines the types and natures of service collaborations defined by the technology, such as component composition through component connectors, or specialized communication patterns, such as event-driven communication and data streaming.

Upon service instantiation, a service platform receives from the adaptation framework, metadata describing the required service, implementation artifacts that have been selected by the planning framework during initial planning, and services that the implementation depends on. Adaptation-aware platforms monitor their managed services, and when they find it necessary, trigger the adaptation framework for a re-planning. The result from the re-planning is a new set of metadata and implementation artifacts that can be used by the platform to perform an adaptation.

In order to hide the details of service instantiation and configuration from the adaptation framework, we apply the DIP to the platform framework. The adaptation framework invokes a service platform to instantiate a service with a package encapsulating the implementation artifacts, called *blueprint*, as a parameter. As the type of implementation artifacts used by a certain technology is highly technology specific, blueprints are black boxes to the adaptation framework. The blueprint may contain technology specific information related to different types of adaptation mechanisms, such as component replacement, component parametrization, insertion of interceptor or monitors, etc. The QuA adaptation framework does not define the format of a blueprint, nor does it ever inspect or manipulate its content. Blueprints are created by technology expert developers, and deployed to the QuA middleware using technology specific tools.

The platform depends on technology specific mechanisms in order to instantiate and adapt the service. Examples of such services are component factories, parsers for component descriptors, binders and configurators, resource managers, etc. These mechanisms may either be implemented as a part of the platform, or they may be deployed to the middleware as services that the platform depends on. Based on the simple abstractions described above, multiple adaptation techniques can be integrated and exploited through the platform framework concept [2,3,4,5,6,13].

3.4 Reflection Support: The Service Meta-object Protocol

The QuA middleware defines a service meta-object protocol that can be used to reflect on services in all phases of their life-cycle. The SMOP is based on a services

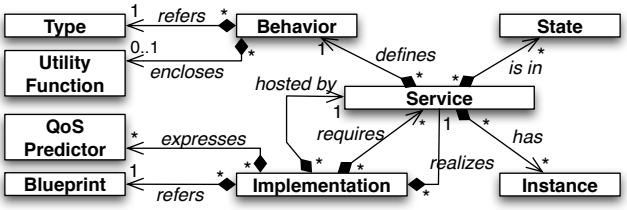


Fig. 3. The QUA service meta-model

meta-model, which is used to describe exactly the aspects of a service related to planning, instantiation, and execution of services managed by QUA—*i.e.*, its behavior (required or provided type, and utility function), implementation (including blueprint, required service platform, and implementation dependencies), and instances if any. Figure 3 describes the QUA service meta-model.

In order to conform to the DIP, the SMOP must provide the adaptation framework with a technologically independent reflective API. The QUA reflective API is based on mirror reflection, where the meta-level facilities are implemented separately from the reflected system as described in Section 2.4. Thus, mirror-based reflection does not require any changes to be made to the reflected system, and it allows the coexistence of technology specific reflective APIs required by service platforms and their mechanisms. In [16], we describe a comprehensive application scenario demonstrating the application of the service meta model, including examples of quality prediction and utility functions.

4 Implementing the QUA-FRACTAL Adaptation Middleware

In [14], we have shown that the framework is applicable to simple programming models, such as the Java programming languages, by designing lightweight component models based on this language. In order to confirm the ability of the QUA adaptation framework to integrate and exploit concrete adaptation technologies, we need to apply the framework to an adaptation technology that provide a rich set of features. To this end, we consider the FRACTAL component model [10] as an interesting candidate technology. The FRACTAL component model is a lightweight and hierarchic component model targeting the construction of efficient and highly reconfigurable middleware systems. FRACTAL has been used in several projects to implement advanced adaptive and self-adaptive behavior [17,2,18]. Thus, if we are able to successfully integrate and exploit the rich set of available mechanisms and tools provided by FRACTAL ecosystem, without coding FRACTAL-specific knowledge into the generic adaptation framework, it is a strong indication that the QUA adaptation framework has the expected capabilities with regards to supporting integration.

The work presented in this paper is based on integrating the powerful, expressive, and flexible component reconfiguration mechanisms provided by FRACTAL. In particular, the *FractalADL Factory* is a component factory that instantiates FRACTAL components and composites from architecture descriptions written in the FRACTAL *Architecture*

Description Language (FRACTALADL) [19]. The FSCRIPT engine interprets configuration scripts written in the FRACTAL-based configuration language FSCRIPT [2]. The FSCRIPT language includes primitives for standard FRACTAL component management, and can be extended in order to support more advanced configurations.

4.1 The FRACTAL Component Model

The reconfiguration capabilities are defined by controllers that defines the level of introspection and control of a component (life-cycle, attributes, bindings, interfaces, etc.).

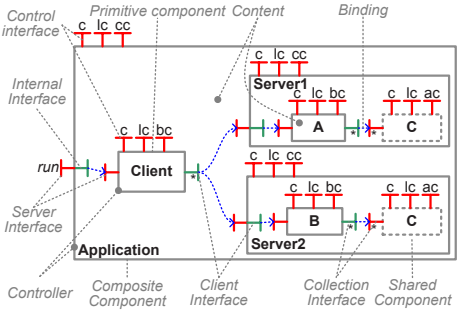


Fig. 4. Architecture of a FRACTAL component

Figure 4 illustrates the different entities in a typical FRACTAL component architecture. Thick black boxes denote the *controller part* of a component, while the interior of these boxes correspond to the *content part* of a component. Arrows correspond to *bindings*, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. External interfaces appearing at the top of a component represent reflective control interfaces, such as the *Life-cycle Controller* (lc), the *Binding Controller* (bc) or the *Content Controller* (cc) interfaces. The two dashed boxes (C) represent a *shared* component.

4.2 The QUA-FRACTAL Middleware

The QUA-FRACTAL middleware has been implemented as a *service platform*, Fractal Platform, that includes an implementation of the JULIA run-time, the FractalADL factory, and the FScript engine, as illustrated in Figure 5.

The FRACTAL platform instantiates services from FRACTAL blueprints, containing FRACTALADL descriptors, implementation classes, and FScript configuration scripts. It extracts ADL descriptors and implementation classes from the blueprint, and invokes the ADL factory to instantiate components from the descriptors. The ADL factory depends on the JULIA run-time to create the component instances from the implementation classes. Finally, FScripts are extracted from the blueprint, and the FSCRIPT engine is invoked to perform configuration based on the FScripts. FRACTALADL and FSCRIPT

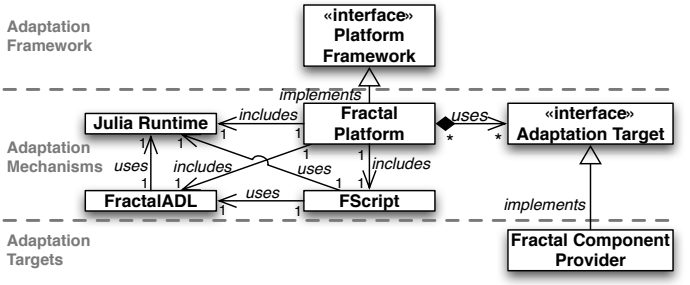


Fig. 5. Architecture of the FRACTAL platform

use standard FRACTAL controllers to perform component management tasks, such as binding, life-cycle management, and parameter configuration.

In order to be able to exploit different combinations of FRACTAL components, we have to enable the QUA planning framework to plan alternative FRACTAL components independently. For example, in the case of composite components, we want to be able to plan certain sub-components independently, in order to find the combination of components that best satisfy the service requirements. The recursive meta-model provided by QUA enables such a nested planning through the definition of implementation dependencies. Instead of publishing an ADL descriptor describing the complete composition, we extract ADL descriptions describing sub-components into separate FRACTAL blueprints. Thus, in the case where several implementations of a sub-component are available, the planner will select the one giving the highest utility.

4.3 The *Comanche* Application

Below, we illustrate our prototype application *Comanche*: a legacy web server developed by the FRACTAL community. *Comanche* is a lightweight web server implemented with the FRACTAL component model[1]. This implementation provides the core features of a web server as a proof of concept of the relevance of FRACTAL for building middleware systems.

In its initial version, *Comanche* is made of several components that identify the various concerns of a web server, as depicted in Figure 6.

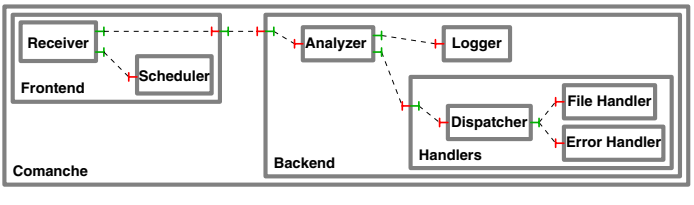


Fig. 6. Architecture of the *Comanche* web server

[1] *Comanche* tutorial: `http://fractal.ow2.org/tutorial`.

In particular, *Comanche* contains a component Scheduler that schedules the treatment of incoming requests (see Figure 6). The implementation of this component controls the allocation of dedicated activities for analyzing incoming requests. The initial implementation of the scheduler creates a thread per incoming request without controlling the number of activities created. In the SAFRAN project [2], an alternative to the scheduler proposes to use a pool of threads to control the number of threads used by the web server. However, the thread pool scheduler provides lower response times than constantly creating new threads.

```
ServiceMirror comancheMirror = QuA.createServiceMirror( Comanche );
comancheMirror.setUtilityFunction( ComancheUtility );
comancheMirror.setImplBlueprint( ComancheBlueprint );
comancheMirror.setImplQoSPredictor( ComanchePredictor );
comancheMirror.setImplDependencies( "s", Scheduler );
```

Listing 1.1. Service mirror reflecting the *Comanche* web server.

In order to be able to apply the QUA-FRACTAL middleware to the configuration of the *Comanche* web server, we had to deploy the *Comanche* application, including ADL descriptors, FScripts, and implementation classes, to the QUA middleware as a blueprint, and to advertise the necessary meta-data, including utility function and quality predictors, to the middleware (see the QuA-specific code for advertising metadata, represented as a service mirror in Listing 1.1).

```
<definition name="Frontend" extends="FrontendType">
  <component name="rr" definition="Receiver"/>
  <component name="s"  definition="Scheduler"/>
  <!-- Definitions of bindings -->
</definition>
```

Listing 1.2. FRACTALADL descriptor of the *Comanche* front-end.

The ADL for the *Comanche* front-end is depicted in listing 1.2. The utility of the *Comanche* server is expressed by a function that returns high utility values for low response times, and low utility values for high response times. Listing 1.3 contains the component replacement script *replace-scheduler*, used to replace one scheduler component with another. The script uses a number of primitive operators, such as *stop*, *bind*, and *remove*, in order to implement the routine that has to be followed in order to safely replace one component with another. These operators are mapped by FSCRIPT to invocations of standard FRACTAL controller interfaces.

```
action replace-scheduler(comanche, scheduler) {
  stop($comanche);
  var frontend = $comanche/child::fe;
  unbind($frontend/child::rr/interface::s[client(.)]);
  remove($frontend, $frontend/child::s);
  add($frontend, $scheduler);
  bind($frontend/child::rr/interface::s, $scheduler/interface::s);
  start($comanche);
  return $comanche;
}
```

Listing 1.3. FSCRIPT statements replacing component in *Comanche*.

5 Evaluating the QUA-FRACTAL Implementation

In order to evaluate the adaptation framework presented in Section 3, we have to consider to what degree the combined QUA-FRACTAL middleware was able to solve the challenges mentioned in Section 1, namely: *ii)*to *integrate into the adaptation framework different technologies* used in the system to be adapted, and *iii)* to whenever possible *exploit the specific features and opportunities* offered by the different implementation technologies used. With regard to *ii)*, we have managed to integrated the FRACTAL run-time and FRACTAL components into the QUA adaptation framework. The integration required an acceptable amount of work, given the availability of developers that have moderate knowledge about the QUA middleware, and some knowledge about the FRACTAL middleware. With regard to *iii)*, we have managed to exploit two FRACTAL specific adaptation mechanisms, namely the FRACTALADL factory and the FSCRIPT language.

In order to reflect the amount code in the resulting middleware that is technology agnostic, technology specific, and application specific, Table 1 presents the number of classes and the byte-code size of the different parts of the resulting middleware and application. As indicated by the table, the FRACTAL mechanisms contribute with by far the largest amount of files and byte-code. The QuA middleware consists of a rather small middleware core, which byte-code size is less than 10% of the size of QUA-FRACTAL platform. Furthermore, the number of QUA-specific files required in order to implement the QUA-FRACTAL platform was only 8. This number includes both the definition of the QUA-FRACTAL platform interface, the QUA-FRACTAL implementation classes, the QUA-FRACTAL blueprint used to encapsulate FRACTAL implementation artifacts, and an helper platform used to instantiate the QUA-FRACTAL platform itself, as a service.

Table 1. Distribution of code in QUA-FRACTAL

Concern	Number of class files	Byte-code size (Kb)	Distribution (%)
QUA middleware	53	276	7
QUA-FRACTAL platform	**8**	**76**	**2**
JULIA run-time	300	1,782	45
FRACTALADL factory	171	816	21
FSCRIPT engine	151	828	21
Utility classes	33	168	4
QUA-FRACTAL total	716	3,946	100
Comanche application	17	76	

The relatively small size of the QUA middleware is the result of keeping the responsibility of the QuA middleware small and concise, and independent of technology specific details and knowledge by the application of the DIP and utility functions. Due to these principles, we are able to control an advanced and comprehensive adaptation middleware technology from this small and generic adaptation framework.

6 Conclusions

Due to the growing heterogeneity of technologies used to implement nowadays distributed systems, existing adaptation middleware faces more and more difficulties to perform technology agnostic adaptations. This phenomenon is particularly true in the component-based software engineering community where most of the state-of-the-art approaches to adaptation suffer from their tight coupling to a particular component model [2,3]. This strong dependency restricts the integration of new technologies (*e.g.*, component models or middleware frameworks) to the fixed set of abstractions supported by the adaptation middleware, thus avoiding the integration of technology-specific adaptation capabilities.

The contribution of this paper is to present the implementation of a modular adaptation middleware, called QUA, whose design supports the integration of various technologies. This design combines the definition of a *Meta-Object Protocol* [15] and the application of the *Dependency Inversion Principle* [11]. The former is used to reflect the meta-data associated to technology artifacts, while a *utility-based planning framework* and a *platform framework* apply the latter to reason about the reflected metadata and perform adaptations, respectively. We validate this design by reporting the integration of the FRACTAL component model into the QUA middleware, and we illustrate the resulting adaptation middleware on the adaptation of a component-based application: the *Comanche* web server. By facilitating the integration of technologies, this approach clearly separates the adaptation concern from the application and the technology.

As a matter of perspective, we plan to extend the set of supported technologies and experiment the consistent adaptation of cross technology applications.

Acknowledgements

The authors thank the reviewers of the CBSE conference for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

References

1. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
2. David, P.C., Ledoux, T.: An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)
3. Batista, T.V., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-based Systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
4. Sun microsystems: Java Platform, Enterprise Edition (Java EE), http://java.sun.com/javaee
5. OSGi Alliance: OSGi Service Platform Release 4, http://www.osgi.org
6. Microsoft. Net: Microsoft. NET Framework 3.5, http://www.microsoft.com/net

7. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice-Hall, Englewood Cliffs (2005)
8. Erradi, A., Maheshwari, P., Tosic, V.: Policy-Driven Middleware for Self-adaptation of Web Services Compositions. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 62–80. Springer, Heidelberg (2006)
9. Kuropka, D., Weske, M.: Implementing a Semantic Service Provision Platform Concepts and Experiences. Journal Wirtschaftsinformatik – Special Issue on Service Oriented Architectures and Web Services 1, 16–24 (2008)
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java. Software Practice and Experience – Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11/12), 1257–1284 (2006)
11. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice-Hall, Englewood Cliffs (2002)
12. Kephart, J.O., Das, R.: Achieving Self-Management via Utility Functions. IEEE Internet Computing 11(1), 40–48 (2007)
13. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st International Workshop on Self-Healing Systems (WOSS 2002), pp. 33–38. ACM, New York (2002)
14. Alia, M., Eide, V.S.W., Paspallis, N., Eliassen, F., Hallsteinsen, S.O., Papadopoulos, G.A.: A Utility-Based Adaptivity Model for Mobile Applications. In: 21st International Conference on Advanced Information Networking and Applications (AINA 2007), pp. 556–563. IEEE, Los Alamitos (2007)
15. Bracha, G., Ungar, D.: Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In: 19th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 331–344. ACM, New York (2004)
16. Gjørven, E., Eliassen, F., Lund, K., Eide, V.S.W., Staehli, R.: Self-Adaptive Systems: A Middleware Managed Approach. In: Keller, A., Martin-Flatin, J.-P. (eds.) SelfMan 2006. LNCS, vol. 3996, pp. 15–27. Springer, Heidelberg (2006)
17. Bouchenak, S., Palma, N.D., Hagimont, D., Taton, C.: Autonomic Management of Clustered Applications. In: International Conference on Cluster Computing (Cluster 2006). IEEE, Los Alamitos (2006)
18. Roy, P.V., Ghodsi, A., Haridi, S., Stefani, J.B., Coupaye, T., Reinefeld, A., Winter, E., Yap, R.: Self-management of large-scale distributed systems by combining peer-to-peer networksand components. Technical Report18, CoreGRID - Network of Excellence (2005)
19. Leclercq, M., Özcan, A.E., Quéma, V., Stefani, J.B.: Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In: 29th International Conference on Software Engineering (ICSE 2007), pp. 209–219. IEEE, Los Alamitos (2007)

A Practical Approach for Finding Stale References in a Dynamic Service Platform

Kiev Gama and Didier Donsez

University of Grenoble, LIG laboratory, ADELE team
{Kiev.Gama,Didier.Donsez}@imag.fr

Abstract. The OSGi™ Service Platform is becoming the de facto standard for modularized Java applications. The market of OSGi based COTS components is continuously growing. OSGi specific problems make it harder to validate such components. The absence of separate object spaces to isolate components may lead to inconsistencies when they are stopped. The platform cannot ensure that objects from a stopped component will no longer be referenced by active code (a problem referred by OSGi specification as stale references) leading to memory retention and inconsistencies (e.g., utilization of invalid cached data) that can introduce faults in the system. This paper classifies different patterns of stale references detailing them and presents techniques based on Aspect Oriented Programming for runtime detection of such problems. We also present a fail-stop mechanism on services to avoid propagation of incorrect results. These techniques have proven to be effective in a tool implementation that validated our study.

Keywords: OSGi, stale references, dynamic services, memory leaks, runtime diagnostics, component validation.

1 Introduction

The OSGi service platform [1] is a framework targeting the Java platform, providing a dynamic environment for the deployment of services and modules (referred as *bundles* in OSGi terminology). The OSGi architecture provides a hot deployment feature by allowing modules to be dynamically added, updated or completely removed during application execution without the need to restart the JVM. OSGi is being used in a myriad of applications (e.g., desktop and server computers, home gateways, automobiles) and is becoming the de facto standard for modularized Java applications [2] [3] [4] [5]. A milestone of OSGi's acceptance in software industry is its adoption in the Eclipse Platform [6].

Although the OSGi platform has evolved and matured in several aspects, its runtime environment does not enforce the isolation of bundles. A certain level of isolation by means of class loaders is provided by the OSGi platform, but bundles are not truly isolated from each other under a memory perspective. There are no separate object spaces between bundles that would guarantee a safe and complete removal of a bundle from the platform. Bundles may freely exchange objects, but there is no mechanism to enforce that an object will not be referenced when its bundle stops.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 246–261, 2008.

Even with events notifying the departure of services and bundles, the current OSGi programming model is not trivial to follow and the handling of such events is error prone. Due to bundle programming flaws, object instances may be kept by a consumer bundle after the provider bundle stops. The usage of such objects leads to memory retention preventing the classes from stopped bundles to be unloaded from memory. Faulty components can be introduced in the system due to propagation of incorrect results (e.g., old or invalid cached data) that may result from calls to those stale objects.

The OSGi specification briefly describes this issue and refers to it as *Stale References*. Avoiding it is a matter of good programming practices since the environment cannot control or inspect it. Although there are mechanisms to minimize the occurrence of this problem, it is not possible to assure that every possibility of stale reference is being taken care of. This problem is difficult to detect in existing diagnostic applications (e.g., Eclipse TPTP, Netbeans profiler, Borland Optimizeit) because it is a consequence of particularities in the OSGi dynamic environment.

The market of OSGi based Commercial-Off-The-Shelf (COTS) components is rapidly growing [2]. Under the perspective of the OSGi dynamicity aspects that we have presented, existing tools or testing suites cannot guarantee or evaluate that OSGi based COTS components can be safely introduced in an OSGi platform without bringing any problems such as stale references upon OSGi life cycle events.

This paper proposes techniques that enable such type of validation for the OSGi environment. We go deeper in the stale references problem by classifying and detailing different patterns of stale references. We propose and validate diagnosis techniques that rely on Aspect Oriented Programming [7] to change OSGi framework implementations enabling them to provide information to detect those patterns during application runtime. We found that a static analysis approach may impose several constraints and it is not suitable to a dynamic environment such as OSGi. We also transparently introduce a fail-stop approach on calls to stale services to avoid the propagation of incorrect results.

Our detection techniques make possible to identify and visualize stale references, achieving an OSGi specific inspection feature that is not yet available in existing diagnostic tools. By identifying such problems it is possible to provide information that can help correcting bundle source code, allowing developers to guarantee the quality of their OSGi targeted applications and components.

All the techniques explained here were validated with the development of a diagnostic tool [8] that can be used to inspect OSGi targeted applications and components. The analysis of four open source OSGi based applications presented stale references after simulating life cycle (update, stop, uninstall) events.

The remaining sections of this paper are organized as follows: section 2 gives an overview of the dynamics in OSGi and its implications; section 3 details different patterns of stale references; section 4 explains the techniques for runtime detection of those patterns; section 5 presents the results of an experiment with 4 open source application as a part of the validation of our work; section 6 talks about related work; and, at last, section 7 presents the future work and conclusion.

2 OSGi Dynamics and Implications

The OSGi framework provides a straightforward service platform for the deployment of modules and services. The deployment unit in OSGi is called bundle, which is an ordinary compressed jar file with classes and resources. The jar file manifest contains OSGi specific attributes describing the bundle. A bundle can be dynamically loaded or unloaded on the OSGi framework and may optionally provide or consume services, which can be any Java object. Applications can take advantage of the dynamic loading feature to update software components without the need to stop the application. For example, a production system may have a bundle updated with a new version due to minor bugs fixed or other types of improvements.

Bundles can access the OSGi framework through a *BundleContext* object which becomes available in the bundle's activation process. Through that object they can register and retrieve services. In OSGi, services are ordinary Java objects that are registered into the framework service registry under a given interface name. The basic process to retrieve a service instance consists in two steps: it is necessary to ask the BundleContext for the desired interface, resulting in a *ServiceReference* object which holds metadata of a service. The next step is to use the BundleContext again to retrieve the service instance that corresponds to that ServiceReference object.

Upon service registration, modification or unregistration—either explicit or implicit when the defining bundle is stopped— the framework notifies the subscribers of the *ServiceListener* interface. Therefore, it is possible for service consumers to know when services become available (registered) or unavailable (unregistered).

Any OSGi targeted code should be written considering the arrival and departure of bundles and services. The code must release references appropriately upon such events. Service consumers must be aware that a service departure means that a service instance or its ServiceReference must not be used anymore. Any usage of the unregistered object may lead to inconsistency.

2.1 Bundles Isolation through Class Loaders

Whenever a bundle is loaded —either during startup or later during runtime— it is provided with its own class loader. Classes and resources from a bundle should be only loaded through its class loader. This individual class loader mechanism permits to unload from memory all classes provided by a given bundle when it is stopped.

The OSGi framework provides a basic level of isolation between bundles by means of that class loading mechanism. A bundle may choose which packages will be visible to other bundles by defining in its manifest an attribute with a list of exported packages. Only classes from exported packages (specified in the bundle manifest) may be instantiated by other bundles, which also need to explicitly specify in their manifest what packages they import. Whenever a bundle tries to reference a type, its class loader will enforce if the visibility rules are followed. Other mechanism that can be seen also as an isolation enforcement is the utilization of optional framework security permissions (AdminPermission, PackagePermission, and ServicePermission) which can provide a fine grained control to grant authority to other bundles perform certain actions, for example to retrieve a given service instance.

2.2 Isolation Limitations

Although there is some isolation level between bundles, this mechanism cannot ensure complete or safe removal of bundles from memory. During bundle active time objects can be exchanged freely between bundles. For instance, a service may receive a parameter object that comes from other bundle. If the bundle that provided the parameter object is stopped there is no guarantee that the service will stop referencing the object it received as parameter, even if the bundle of origin of that object uninstalled from the framework.

There is no security enforced communication channel (e.g., communication via proxy objects) that can be closed upon bundle departure, nor a protection domain (i.e, individual object spaces in memory) that enforces communication restrictions or other forms of application isolation.

The OSGi platform does not provide a true means of isolation between bundles. It mostly relies in a set of good programming practices to avoid the misreferencing of objects after bundles are stopped.

2.3 Stale References

The OSGi specification, release four, defines in the section 5.4 a stale reference as

> *"a reference to a Java object that belongs to the class loader of a bundle that is stopped or is associated with a service object that is unregistered"*

The utilization of such objects after the provider bundle being stopped leads to inconsistencies such as (1) incoherent operation results (e.g., stale services returning old data from stale caches) or erroneous behaviour due to the stale object's context (e.g.., network connections, binary streams) be released or de-initialized; (2) garbage collection obstruction of the retained object, its class loader, and the class loader's loaded types, leading to a memory leak.

Utilizing a ServiceTracker or an OSGi component model helps to minimize the occurrence of stale references. The ServiceTracker is a utility class in the OSGi framework for providing a transparent means for locating services but it is error prone since service consumers may not release the consumed service instances appropriately. OSGi Declarative Services (part of the OSGi R4 compendium specification), Service Binder [9], iPOJO [10] and Spring Dynamic Modules [11] are OSGi component models that provide the transparent handling of services arrival and departure. However, their usage would not avoid all possible types of stale references. Other patterns of stale references which are detailed in the next session may not be avoided by such mechanisms.

2.3.1 Propagation of Incorrect Results

The usage of an unregistered service may lead to inconsistent method calls. If a bundle unregisters a service, it is likely that the service needs to be disposed; therefore it may release internal resources (open file streams and database connections, etc) and calls on that object would produce erroneous behaviour. Exceptions may be raised

(e.g., access to a method that internally would try to use a closed connection) when methods of stale references are used. However if such method calls do not fail but produce incorrect results, there is a worst scenario where faulty components are introduced into the system with risks to propagate inconsistencies throughout the whole application. This can happen due to the stale object's internal state being invalid or stale (e.g., old cached data), which compromises the accuracy of operations involving that object. Such types of faults are harder to detect since the system would hide these issues and continue to work apparently without any problem.

A service failure mechanism, as presented in [12], currently is not enforced by the platform. A fail-stop strategy would be able to make the faults more explicit when using stale references. If any calls to stale references would result in a crash (an exception thrown) there would be no propagation of incorrect results, and bugs would be evident.

2.3.2 OSGi Specific Memory Leaks

While the previous problem may sometimes be identified due to exceptions thrown, memory retention is rather difficult to be seen. In addition, the retention of class loaders impedes OSGi to dynamically unload the classes from a stopped bundle.

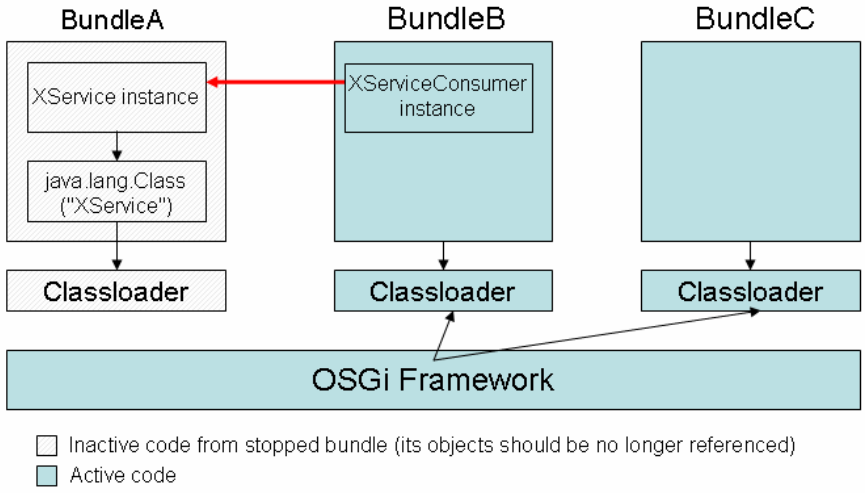


Fig. 1. The arrow from BundleB to BundleA illustrates a stale reference that prevents the appropriate unloading of BundleA from memory

According to the Java Language Specification [13], a class or interface reification (a java.lang.Class instance) may be unloaded if and only if its defining class loader may be reclaimed by the garbage collector. As long as an object from a stopped bundle is reachable (Figure 1) we will have a reference to that object's type as well, which references the bundle class loader which keeps all loaded types. Consequently, the classes can never be unloaded due to the presence of stale references.

3 Patterns of Stale References

As stated previously, the framework cannot guarantee that the objects provided by a bundle will no longer be referenced when the bundle stops. Neither the OSGi framework itself nor the mechanisms mentioned in section 2.3 can completely avoid stale references. In the current OSGi specification, the framework needs to share responsibilities with bundles. The bundle side is error prone as it depends on good programming practices to correctly handle the departure of services and bundles.

The correct handling mentioned previously will handle only a few patterns of stale references. We have classified three main patterns: (1) Stale services; (2) forwarded objects and (3) active threads from stopped bundles.

3.1 Stale Services

Stale services are a pattern of stale references that can be found when an unregistered service is still being referenced by active bundles. We considered that there are two levels of service referencing: reference to a service instance and reference to a ServiceReference instance. The former is the service object itself and the latter is a framework metadata object which is necessary to get a service instance. We kept references to ServiceReference instances as a simple case, but we classified a specialization of the reference to service instances as two possibilities: services from stopped bundles and services from active bundles. Therefore, we present the concept of stale services as three variations:

- Reference to an unregistered instance of a service whose bundle is still active (has not stopped);
- Reference to an unregistered instance of a service from a stopped bundled (update or uninstallation would lead to stopping the bundle as well);
- References to unregistered org.osgi.framework.ServiceReference objects.

The first case can happen during the active life-time of a bundle which may unregister a service due to an internal bundle change, for example. If after unregistration the service instance is retained by service consumers from other bundles we have a case of stale reference. In this case, the service can propagate incorrect results and it will also be prevented to be garbage collected.

The second pattern is rather similar to the first one, but now the propagation of errors is more likely because the bundle has been stopped and may have suffered some de-initialization code. In addition, the bundle class loader and classes would be prevented to be unloaded from memory.

The latter case of stale service (references to unregistered ServiceReference objects) does not prevent the unloading of bundle classes because there would be no reference to a bundle object, since the ServiceReference object is provided by the framework. Because of that, one may argue that this pattern does not fit the stale reference definition. However, this case has been added to our patterns because it may bring faults to the application and also characterizes the mishandling of service unregistration. When a ServiceReference is unregistered, subsequent calls to the framework using that ServiceReference object would return a null value, leading to a NullPointerException upon any method call attempt on the resulting value.

3.2 Forwarded Objects

Bundles may freely exchange messages between them by means of service method calls. Ordinary objects may be passed as method parameters across bundle boundaries without restriction. Also, there is no restriction for a service to retain an object received as a method parameter or to forward that object reference to objects from other bundles. If the bundle that provides that forwarded object is stopped, the same memory retention problem as the stale service pattern would happen. The same also applies when objects are registered in server object repositories (e.g., MBean server, RMI registry) and are not appropriately unregistered when bundles are stopped.

We have identified two variations of the forwarded objects pattern:

- Forwarding of ordinary (non-service) objects
- Forwarding of services as ordinary objects

Figure 2 shows an example of the forwarding of an ordinary object. Consider that the code on that example runs on an object from Bundle X, and foo.BarService is provided by an object from Bundle Y. Bundle X calls a method on a service from Bundle Y and sends a parameter, which is a local ordinary (non-service) object from Bundle X. That parameter will be retained as an attribute in the Bundle Y service. If Bundle X is stopped, uninstalled or updated, the object that was sent to Bundle Y's service will fit in the regular case of stale reference: impossibility to garbage collect the referenced object (localObj) and to unload the classes previously provided by Bundle X's class loader.

```
//Code on a BundleX retrieves a service from a BundleY
ServiceReference ref =
ctx.getServiceReference("foo.BarService");
BarService bar = (BarService)ctx.getService(ref);

//LocalObject is created in (and provided by) BundleX
LocalObject localObj = new LocalObject();

//service from BundleY will hold an object from BundleX
bar.setAttribute("anAttribute", localObj);
```

Fig. 2. Forwarding of an ordinary object

The second type of forwarded object pattern is detailed in Figure 3. It shows that the Bundle X uses a service instance from Bundle Z and forwards that instance to a service from a third bundle (Bundle Y). Bundle Y now references an object from Bundle Z without knowing that it is a service. Although at that time the foo.BarService service holds an instance of xyz.AService, most likely it would ignore the unregistration of xyz.AService, since the setAttribute method semantics does not expect a service. If Bundle Z (the provider of the xyz.AService "attribute service") is ever stopped, the foo.BarService will not release the reference to the xyz.AService

object. Bundle Y would point to a stale reference that prevents the unloading of classes from Bundle Z.

A significant difference between referencing an ordinary (non-service) object from a stopped bundle and referencing a service instance from a stopped bundle is the absence of framework events to notify the departure of ordinary objects. But if a forwarded service is treated as an ordinary object, notifications of service unregistration are ignored and do not help.

```
//Code on a BundleX retrieves a service from a BundleY
ServiceReference ref =
ctx.getServiceReference("foo.BarService");
BarService bar = (BarService)ctx.getService(ref);

//Code on a BundleX retrieves a service from BundleZ
ServiceReference anotherRef =
ctx.getServiceReference("xyz.AService");
AService servObj = (AService)ctx.getService(anotherRef);

//service from bundleY holds a service as an attribute
bar.setAttribute("anAttribute", servObj);
```

Fig. 3. Forwarding of a service instance as an ordinary object

3.3 Active Threads from Stopped Bundles

According to the OSGi specification, when a bundle is stopped it has to immediately stop all of its executing threads. Since there is no isolated bundle space in memory, the framework cannot cancel a bundle's set of executing threads. So, it must rely on good OSGi programming practices leaving that responsibility to the bundle developer.

Table 1. Summary of stale references

Referred object	**Memory Retention (bundle objects and class loader)**	**Incorrect Results**
Unregistered Service instance (Stopped bundle)	Yes	Yes
Unregistered Service instance (Active bundle)	Yes (but no class loader retention)	Yes
Unregistered `ServiceReference` instance	No	Yes (NullPointerException)
Active Thread (stopped bundle)	Yes	Yes
Forwarded object (stopped bundle)	Yes	Yes

If the thread is not stopped in such cases, the same stale reference issue is found: an object (the Runnable object) from a stopped bundle is still reachable in memory, preventing garbage collection of its class loader (the bundle class loader) and the loaded types of that bundle.

4 Detection Techniques

Information to track object references and diagnose stale references is not present in implementations OSGi of the framework. Several reasons have led us to think that changing the source code of OSGi implementation to add that information would not be adequate. It would be needed to inspect the registration and retrieval of services, class loader creation, etc. The custom code to track such objects would be scattered all over the OSGi framework implementation code. It is clear that a solution which customizes a given OSGi implementation would compromise the portability to other OSGi implementations. In addition, other problems such as tracking the creation of threads would concern bundles but not the framework. This would imply in changing bundle code as well, which we most likely don't have access in all applications.

The whole situation led us to choose the application of Aspect Oriented Programming (AOP) [7] techniques. Instead of adding a cross-cutting concern to the code of OSGi implementations, we left the tracking code as separate aspects. AOP would enable to weave those aspects into different OSGi implementations. The process would be the same for all of them: each implementation would have its bytecode changed resulting in a composed implementation capable of providing information to identify stale references.

The reference tracking techniques presented here rely on a special type of reference provided by the Java programming language, called *weak reference.* Weak references are different than ordinary (strong) references. They do not prevent a referred object to be reclaimed from memory and are able to tell if an object has been garbage collected.

4.1 Point Cut Definitions

AOP introduces the concept of *joint points*, which are well defined points in the program flow (e.g., method call, constructor call). *Point cuts* are elements that pick one or more specific join points in the program flow. We have defined two different sets of point cuts. One was responsible for aspects that would be applied to the framework, for example tracking service registration and retrieval, bundle start up, class loader creation, etc. The other set of join points was responsible for the aspects on bundles, which so far were limited to the creation and start up of threads.

The code that is injected into point cuts during the weaving process is called *advice* in AOP terminology. The portions of code defined in the advices are executed during method interception. In the techniques that we have developed and tested, the advices contained the calls to the code that enabled the tracking of objects.

4.2 Detection of Stale Services

With AOP, service registration can be intercepted and each ServiceReference object tracked with weak references. Our technique consists also in track the garbage collection of each instance provided by a ServiceReference. Multiple service instances can be served by the same ServiceReference when the service provider is a *ServiceFactory*, which can provide one service instance per bundle.

In order to verify the existence of stale services, it is necessary to analyze tracking information relative to unregistered ServiceReference objects. There are two straightforward manners to know the existence of stale services. One is checking if the unregistered ServiceReference object has not been garbage collected, and the other is to verify if all service instances of each unregistered ServiceReference have been garbage collected. The former would characterize the pattern of a reference to an unregistered ServiceReference. The latter identifies the pattern of a reference to an unregistered service instance.

4.3 Detection of Active Threads from Stopped Bundles

The detection of thread creation and its start up in bundle code is necessary in order to have more information about them. Instead of weaving the framework, this approach implies in weaving the bundles. Two options are possible: static weaving or dynamic (runtime) weaving. The same aspects are reusable in both approaches.

The static weaving is easier to perform but adds the step of externally weaving the bundles before loading them into the platform. The dynamic approach is more flexible but adds the overhead of weaving while loading the bundles in runtime. It is also necessary to add code in the framework, by AOP as well, to intercept the loading of bundles and dynamically weave them.

The information on thread point cuts allows establishing a bundle-thread relation that can be stored for later inspection. Running threads that are in the bundle-thread map can have their metadata inspected (e.g. the class loader of the bundle that started the thread) and compare it with logged information of the bundle that started the thread. It is possible to identify if the bundle that started the thread has been update, stopped or uninstalled.

4.4 Identifying Forwarded Objects

Identification of forwarded objects was found to be more difficult and depends on the inspection of dumps of memory, as the one provided by tools such as jmap which comes with the Java 6 SDK. It is necessary to inspect a memory dump and verify if there are reachable objects whose class loader belongs to a stopped bundle. Jhat is a tool also available in the Java 6 SDK which allows performing queries o memory dumps. Its API can be integrated into applications that can programmatically perform queries on memory.

Establishing a relation between runtime information and memory dump information is difficult. An object's id in the heap is a sort of JVM private information that is not available to the runtime objects via a Java API. User intervention constructing ad-hoc queries has proven to be more precise some times. This happened due to the fact that automated inspection extracted runtime information of private attributes by

means of reflection and compared it with results from queries on memory dumps. The results most of the times would return a list of suspects that would need to go through a manual inspection by the user.

5 Validations and Experiments

The techniques to detect the patterns of stale references presented here were developed, tested and validated. We have developed a diagnostic tool called Service Coroner [8] which examines the "dead" objects from stopped bundles. Our work comprises the implementation of the aspects to track the code, the classes to perform the queries, the tool that visualizes the problems and a fail-stop mechanism to avoid calls on stale services. The latter was developed as a side experiment that we detail in the end of this session.

Aspects were developed and weaved with AspectJ [14] and each technique was initially validated by bundles that were intentionally developed with errors that would present stale reference problems. A series of life cycle events (stop, update or uninstall) would lead to stale references that were diagnosed by the tool.

The diagnostic tool and the results of an initial experiment are presented in [8]. We have extended that experiment by adding two other open source applications and also analyzing stale threads. The tool is able to inspect OSGi applications and diagnose the patterns presented in this paper.

5.1 Portability Across OSGi Implementations

Althought the process of weaving an OSGi implementation may be seen as intrusive due to the changes it performs in the bytecode, the techniques that we have developed as separate aspects where easy to be applied to different OSGi implementations. As part of the validation, we have achieved to weave the diagnostics aspects into the three main implementations of the OSGi specification, Release 4: Equinox [15], Felix [16] and Knopflerfish [17]. All of the weaved platforms were successfully tested with our bundles that present the stale references patterns.

From a source code point of view there was no need to change any of the implementations. The process of aspect weaving was the same on all of the three platforms, and consisted on a simple build process that basically compiles the Service Coroner tool, the aspects and then weaves the aspects into the OSGi implementation.

5.2 Experiment on Open Source OSGi Applications

We have validated the diagnostic tool in an application scenario where errors would not be intentional like in our test bundles. Four open source applications constructed on top of OSGi were inspected with the Service Coroner tool: JOnAS[1] 5.0.1 [18], SIP Communicator Alpha 3 [19], Newton 1.2.3 [20] and Apache Sling [21]. JOnAS is a JEE application server; SIP Communicator is a multi-protocol instant messenger application; Newton is a distributed component framework that provides an

[1] We have also inspected Apache Geronimo and Glassfish V3 JEE servers, however analyzing them would not bring significant results since they do not use the OSGi service layer.

implementation of the Service Component Architecture (SCA) standard [22]; and Sling is a web framework that uses a Java Content Repository. All applications are of significant size, especially JOnAS, whose core is about 400 000 lines of code but comes to over 1 500 000 when the other components are taken into account.

Table 2 presents an overview of the experiment that was run on a Sun HotSpot JVM 1.6.0u4. All OSGi implementations utilized have been previously weaved with the aspects that we have developed. The line IV of table 2 shows that JOnAS, SIP Communicator and Sling are partially developed with component models for the OSGi Platform: iPOJO, Service Binder and Declarative Services, respectively. Nevertheless, Newton which provides an implementation of SCA has not been developed with a component model.

Table 2. Overview of the experiment. Lines VIII to XI present the results.

I	Application	**JOnAS**	**SIP Comm.**	**Newton**	**Sling**
II	Version	5.0.1	Alpha 3	1.2.3	2.0 incubator snapshot
III	OSGi Impl.	Felix 1.0	Felix 1.0	Equinox 3.3.0	Felix 1.0
IV	Bundles using Component Models	20 iPOJO [10]	6 Service Binder [9]	0[2]	18 Declarative Services [1]
V	Lines of Code	Over 1 500 000	Aprox. 120 000	Aprox. 85 000	Over 125 000
VI	Total Bundles	86	53	90	41
VII	Initial No. of Service Refs.	82	30	142	105
VIII	No. of Bundles w/ Stale Svcs.	4	17	25	2
IX	No. of Stale Services Found	7	19	58	3
X	No. of Stale Threads	2	4	0	0
XI	Stale Services Ratio (IX/VII)	8.5 %	63 %	40.8%	2.8%

The tool was capable of executing scripts that could simulate life cycle events (update, start, stop, uninstall). A script executed by the tool simulated the update of components during runtime by performing calls on the update method of bundles that provide services (except for bundles related to the OSGi framework or component models). We used a standard 10 seconds interval between each bundle life cycle method call. With Newton and Sling we had to adapt the script because of exceptions being raised during bundle update. Instead of the update method, we performed a call to the stop and start methods with the standard interval between each call.

[2] Actually the whole Newton implementation is an SCA constructed on top of OSGi, but its bundles did not use an OSGi component model like the other analyzed applications did.

5.3 Fail-Stop Calls on Stale Services

A crash-only principle, as provided in [12], could be adapted to services in the OSGi environment. We have implemented this fail-stop approach to avoid the propagation of incorrect results when calling methods on stale services. Any method call on stale services would throw an exception. Actually such calls were being done through a proxy object dynamically generated.

We have added another point cut to intercept the calls of the getService method in the BundleContext. Whenever a service instance was requested, the result would be a proxy object that wrapped the service instance. The proxy would receive the calls and delegate them to the actual service. Upon service unregistration, the proxy object had its state invalidated. Subsequent calls to the invalidated proxy would throw a runtime exception. Proxies were cached to avoid creating multiple proxies for the same service instance if it was requested multiple times.

The experiment presented previously did not utilize the fail-stop services. We have successfully tested it in a controlled environment where we developed all bundles deployed in the framework. Other adjustments would be necessary to make our implementation more robust and usable in other scenarios. This strategy could be taken further to minimize the impact of stale services, the strategies to handle such exceptions would allow the auto correction of applications that upon such crashes could react trying to retrieve a valid service or aborting the operation if no valid instance of the service is found.

5.4 Limitations and Drawbacks

Some drawbacks have been found regarding the implementation of the techniques presented here. The first one is regarding the OSGi optional security layer when using digitally signed jars files. Since we have utilized bytecode weaving, the resulted jar file will be different from the original one. Thus, the loading of the changed framework jar file will imply in security errors that will impede the start up of the OSGi platform. This could be found with Equinox [14] version 3.3.2 which provides the digitally signed jars feature, a security feature whose objective is to ensure that jars contents are not modified. In order to utilize our tool, such security option would have to be disabled. We have achieved to turn that off on Equinox by removing all information about security on the manifest and the jar file.

The second drawback was found when doing inspections of memory dumps using the jhat API integrated to our tool. The process of reading memory dumps consumes a large amount of memory and occasionally would lead to out-of -memory errors. An alternative would be using such tools as a parallel auxiliary tool instead of trying to integrate it with the running application.

Although we have removed the propagation of incorrect results produced by stale services and made their utilization explicit by throwing exceptions, generally the proxy solution of our fail-stop approach has two limitations. It does not completely solve the memory retention problem. Upon service unregistration the proxy can free the reference to the actual service, but the service class loader (and all java.lang.Class objects it has loaded) would still hang in memory.

6 Related Work

Our work addresses a problem which is a consequence of code isolation limitations in a specific Java-based middleware for services and components. The same issues apply in environments with similar modularity approaches based on the concepts of OSGi, such as the upcoming Java Module System [23]. Thus theses techniques could be adapted to detect the same problem when that system becomes available.

We focus on the dynamic diagnosis of OSGi applications, evaluating OSGi specific problems during runtime. There are other mechanisms partially addressing this problem in OSGi and in other platforms as well. OSGi component models [9], [10] and [11] provide mechanisms that automate service location and handle service departure but do not avoid all patterns of stale references, as previously mentioned.

A formal model was built on [24] for OSGi verification. By doing that formal analysis they were able to check and identify stale references problems. However their solution was coupled to a specific OSGi implementation (Knopflerfish) and constrained by the limitation of the environment that was used for formal verification. Only applications with a maximum of 10 000 lines of code could be analyzed. They proposed three different solutions to avoid stale references. On each solution the services would have to extend from a default service superclass that provides a lock object. All solutions would depend on synchronization on that object in order to acquire a lock to access the service.

A service failure approach [12] presents a fail-stop solution to handle faults in the composition of services in SOA environments where consumers of a service must anticipate that any service provider will crash from time to time. Another work [25] presents, like ours, a proxy-based service solution to deal with fault tolerance. However, their approach to is different and does not prevent the stale service from being called. Their proxy implementation is responsible for dynamically locating the best service implementation, and in case of faults it tries to locate another service.

Concerning isolation mechanisms, other environments such as .NET [26] have concepts like application domains which resemble lightweight processes isolated from one another and can even be terminated without interfering in the other domains execution. Communication across application domains is done in an RPC fashion and objects are sent via marshalling. Application domains can be dynamically loaded but have limitations in being unloading.

In Java, an effort on JSR 121 [27] provides an environment where applications can be isolated from each other by means of *Isolates*, which are application units which resemble lightweight processes. Applications are isolated in different object spaces but they can share some resources like runtime libraries. Communication between isolates can be done through Java RMI (remote method invocation) based mechanisms which imply in marshalling objects across contexts.

7 Conclusions and Future Work

The OSGi service platform is a dynamic environment for modules (bundles) and services, but it still does not provide a completely isolated environment where services and bundles may be transparently removed during runtime without the risk of

having their objects still being referenced by active code. Memory instrumentations tools currently available (e.g., Eclipse TPTP, Netbeans profiler, Borland Optimizeit) do not consider such particularities of the OSGi framework such as bundle life cycle. The problem of *stale references* described in the OSGi specification may happen if misprogrammed bundles do not handle correctly services unregistration and bundles unavailability. The utilization of stale references introduces memory leaks and faulty components into the system due to the propagation of incorrect results (e.g., a stale service that provides invalid cached data).

This paper presents different patterns of stale references, techniques to diagnose them and a fail-stop mechanism to minimize inconsistent results due to the utilization of stale services. The runtime diagnosis techniques presented here were implemented and validated in a tool called Service Coroner, and were effectively tested against four open source applications. Our detection techniques provide a solution that is portable across different OSGi implementations, without needing to change their corresponding source codes. We rely on AOP to keep the tracking code as separate aspects that can be weaved into different OSGi implementations. Weak references were used to identify which tracked objects have been garbage collected or not.

In a COTS market that targets OSGi application it would be necessary to somehow measure the quality of the components. For example, if they are able to be updated in the system without leaving any weak references or if they would not provoke such problems in the system.

The diagnostics tool that is part of our work addresses OSGi specific issues not covered by currently available tools. Our techniques have proven that it is completely feasible to analyze large OSGi applications and components during runtime, allowing to detect the presence of implementation flaws that lead to stale references. We were able to evaluate if the applications' components are ready to handle some dynamic characteristics of the OSGi platform like being able to cope with module updates.

The initial fail-stop mechanism that we provided invalidates any method call on stale services, avoiding the propagation of incorrect results and facilitating to know where stale services are being used in the application. Some improvements need to be done in that mechanism in order to run it in any type of OSGi application.

In our future work, we also plan to provide a more automated test approach by wrapping the script execution on unit tests. A wider range of OSGi based applications should be tested. It would also be important to adapt the presented techniques for providing the runtime inspection of the Eclipse platform's extension points (although constructed on top of OSGi, Eclipse has its own dynamic plugin mechanism).

References

[1] OSGi Alliance, `http://www.osgi.org`

[2] OSGi Alliance. About the OSGi Service Platform, Technical Whitepaper Revision 4.1, `http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf`

[3] Delapp, S.: Industry Use of OSGi Continues to Increase (retrieved April 9, 2008), `http://www.infoq.com/news/OSGi-Use-Increases`

[4] Chappel, D.: Universal Middleware: What's Happening With OSGi and Why You Should Care (retrieved April 9, 2008), `http://soa.sys-con.com/read/492519_3.htm`

[5] Desertot, M., Donsez, D., Lalanda, P.: A Dynamic Service-Oriented Implementation for Java EE Servers. In: 3th IEEE International Conference on Service Computing, Chicago, USA, pp. 159–166 (2006)
[6] Gruber, O., Hargrave, B.J., McAffer, J., Rapicault, P., Watson, T.: The Eclipse 3.0 platform: Adopting OSGi technology. IBM Systems Journal 44(2), 289–300 (2005)
[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
[8] Gama, K., Donsez, D.: Service Coroner: A Diagnostic Tool for Locating OSGi Stale References. In: Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications, Parma, Italy (2008)
[9] Cervantes, H., Hall, R.S.: Automating Service Dependency Management in a Service-Oriented Component Model. In: Proceedings of the 6th International Workshop on Component-Based Software Engineering, Portland, USA (2003)
[10] Escoffier, C., Hall, R.S., Lalanda, P.: iPOJO: An extensible service-oriented component framework. In: IEEE International Conference on Service Computing, Salt Lake City, USA, pp. 474–481 (2007)
[11] Spring Dynamic Modules for OSGi™ Service Platforms, http://www.springframework.org/osgi
[12] Hobbs, C., Becha, H., Amyot, D.: Failure Semantics in a SOA Environment. In: 3rd Int. MCeTech Conference on eTechnologies, pp. 116–121. IEEE Computer Society, Montréal (2008)
[13] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn., pp. 330–331. Addison-Wesley, Reading (2005)
[14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)
[15] Equinox Framework, http://www.eclipse.org/equinox/framework
[16] Apache Felix, http://felix.apache.org
[17] Knopflerfish OSGi, http://www.knopflerfish.org
[18] JOnAS Open Source Java EE Application Server, http://jonas.objectweb.org
[19] SIP Communicator, http://www.sip-communicator.org
[20] Newton Framework, http://newton.codecauldron.org/
[21] Apache Sling, http://incubator.apache.org/sling/
[22] Service Component Architecture Specifications – Open SOA Collaboration, http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
[23] JSR 277: Java Module System, http://jcp.org/en/jsr/detail?id=277
[24] Chen, Z., Fickas, S.: Do No Harm: Model Checking eHome Applications. In: Proceedings of the 29th Intl. Conference on Software Engineering Workshops, Minneapolis, USA (2007)
[25] Ahn, H., Oh, H., Sung, C.O.: Towards Reliable OSGi Framework and Applications. In: Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France, pp. 1456–1461 (2006)
[26] Escoffier, C., Donsez, D., Hall, R.S.: Developing an OSGi-like service platform for. NET. In: Consumer Communications and Networking Conference, Las Vegas, USA, pp. 213–217 (2006)
[27] JSR 121: Application Isolation API Specification, http://jcp.org/en/jsr/detail?id=121

Towards a Systematic Method for Identifying Business Components

Antonia Albani[1], Sven Overhage[2], and Dominik Birkmeier[2]

[1] Information Systems Design,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
a.albani@tudelft.nl
[2] Component and Service Engineering Group,
Business Informatics and Systems Engineering Chair,
University of Augsburg,
Universitaetsstrasse 16, 86159 Augsburg, Germany
{sven.overhage,dominik.birkmeier}@wiwi.uni-augsburg.de

Abstract. The identification of business components, which together define a *modular* systems architecture, is a key task in todays component-based development approaches for the business domain. This paper describes the Business Component Identification (BCI) method which supports a systematic partitioning of a problem domain into business components. The method allows the designer to state preferences for the partitioning process and uses them as the basis to produce an optimized balance between the business components' granularity on the one hand and their context dependencies on the other hand. It makes use of business domain models specified during the definition of system requirements and can be integrated into the early design phase of a component-based development process. The paper also shows how the produced partitioning can easily be refined into an architecture specification and thus can be used as a starting point for the technical design of a software system and/or its business components.

1 Motivation

Modern component-based approaches allow developers to realize software systems in business domains by partitioning a problem space into a set of proper business components, developing or discovering suitable candidates, and assembling them to obtain the aspired solution [1,2,3]. This modular way of systems development promises to bring many advantages, among which especially a reduced time to market, the increased adaptability of systems to changing requirements and, as a result, reduced development costs are of key importance for the IT strategy of todays enterprises [1,4].

A prerequisite for the envisioned breakthrough of component-based approaches in practice, however, is to better support the underlying modular development paradigm with specialized methods and tools. Although the modular paradigm sounds rather straight-forward at a first glimpse, it introduces a variety of methodological challenges when being analyzed more closely. As a consequence, both the partitioning as well as the composition process continue to pose research questions. Compared to the composition process, where a lot of research is ongoing and for which methods to browse, adapt,

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 262–277, 2008.

as well as to assemble components in a predictable way have already been proposed [5,6,7,8,9], especially the question of how to partition a problem space into modular components still remains to be addressed.

In line with this observation, component identification strategies found in literature are usually limited to basic guidelines or general advices. The established partitioning principle of *maximizing cohesion and minimizing dependencies* between components, e.g., states that contextually related functions and data should be grouped together and ideally constitute a single component [10,11,1]. This principle, however, does not make a statement about an optimal component granularity. Consequently, it might be conceivable to design coarse-grained components containing all required functionality and having no context dependencies at all. Because this leads to redundant implementations of supporting functions and makes components more difficult to maintain, an alternative is to outsource supporting functions into separate components and opt for a better reuse grade. In practice, designers will have to strive for an *optimal balance between self-containedness and implementation reuse* [1]. This means that even with the advices and guidelines from literature taken for granted, a component-based system can well be partitioned into parts of varying size and context dependencies. To date, there only exist generalized approaches that show how a grouping of functions can technically be realized (see e.g. [12]) and discussions about different aspects that have to be taken into account (e.g. selected aspects of scale and granularity presented in [1]). The partitioning itself is still left completely to the designer and his or her personal skills, though.

In this paper, we present the *Business Component Identification (BCI)* method, which systematically supports the partitioning process and helps designers to find an optimized set of business components. The presented method takes results from the requirements definition as input and forces designers to make their partitioning preferences explicit. Based on these preferences, it generates an optimized partitioning of a problem space into business components, which provides a basis for further refining. It allows designers to make use of a rational, unequivocal partitioning procedure and validate the stability of the result against modified preferences. In doing so, the BCI method contributes to evolve the partitioning of component-based systems from handcrafting to an engineering process. The key research questions addressed in this paper are a) how the information modeled in business domains can be used to identify business components in a formal way and b) which optimization methods are suitable for the identification of business components leading to better results than existing solutions. Principally, the introduced partitioning algorithm is not limited to business domains, since it uses process and concept models as inputs which are being created in many application domains. To date, however, we have only applied BCI in business domains.

In section 2, we firstly discuss how to integrate BCI into the component-based development process. This discussion will also elaborate on the input that can be taken as a basis for the partitioning as well as the output that has to be generated by the BCI method. In section 3, we will then describe the BCI method in detail and present the algorithms used for the generation as well as the optimization of a partitioning. Section 4 briefly presents related approaches. We conclude the paper with a discussion of additional aspects that will be taken into consideration in the future and the work that has been done to validate the results of BCI in practice.

2 Systems Development and Component Identification Process

The partitioning of a problem space into components is a core part of the component-based development process and has a significant impact on the quality of both the resulting software system as well as its constituent components. Component-based development process models presented in literature therefore typically either comprise an explicit component identification phase before the actual design starts or at least include this task as an early step of the system design phase [2,13,12]. The extent, to which a partitioning has to be made from scratch, of course, depends on the availability of components that eventually can be reused.

With mature component markets in place and components preferably being reused instead of being newly developed, the partitioning process during the design of a software system needs to be driven by two determinants: the *predefined architecture* imposed by reusing existing components and the *conceptual models* created during the requirements definition. The conceptual models describe processes and information of the problem domain which have to be managed. There are various process models that can be used to develop component-based systems *with reuse*, among which the *Reuse-Oriented and Reuse-Driven Development* approaches [2] as well as the *Assemble Route* of Catalysis [13] are the more prominent ones. In such a reuse-oriented scenario, the partitioning of a problem domain into components also is an important step during the so-called development *for reuse*, which provides reusable components for the development of systems. Reusable components are usually not being developed in isolation, but in so-called domain engineering approaches in which entire problem spaces are being partitioned.

The before-mentioned reuse-based development has repeatedly been described as an ideal component-based software engineering scenario in literature. Using a component-based development approach, however, even is able to bring substantial benefits where component markets and in-house reuse are not established, since modular systems with easily replaceable parts better support managing changes [12,1]. Cheesman and Daniels have presented a process model that supports component-based systems development without a special focus on reuse [12]. In this case, the partitioning process can begin *from scratch*. It solely depends upon domain-oriented conceptual models that have been created during the requirements definition. Notably, however, is the fact, that none of the process models mentioned in this chapter describes how to achieve a good partitioning in detail. Instead, all of them are limited to giving very heuristic advice or to introducing technical means which merely help to capture relationships and dependencies between parts of the domain models. To advance the state of the art, we integrate a rational partitioning procedure, namely the BCI method, into the component-based development process.

The integration is demonstrated for the *UML Components* process model introduced by Cheesman and Daniels [12], which we have chosen for several reasons: firstly, mature component markets today are rather the exception than the rule and especially the development of business systems can not yet systematically include the reuse of existing components. Furthermore, the UML Components process model is well-established, easily applicable in practice, and – thanks to its close relationship to Catalysis as well as to other approaches [12, p. xv] – the transfer of our results to different process models is rather straightforward.

The UML Components process already includes an explicit component identification phase. It is part of the system specification and follows immediately after the requirements engineering (see fig. 1). The goal of the component identification phase is to come up with an initial specification of the system's architecture and its constituent components, which is then refined during the next design steps. The system partitioning is driven by the description of the problem domain and – following established software engineering principles – separates the discovery of system components (the front-end side) from the discovery of business components (the server side providing the business functions).

In this paper, we focus on the discovery of business components, which is based upon the *business concept model* and the associated *business processes*. The business concept model documents the information which is being processed in the application domain. It consists of *information objects* (concepts) and identified structural relationships between them. The business processes describe workflows of the business domain which have to be supported by the software system. They contain *business functions* (modeled as process steps) as well as the temporal relationships between them.

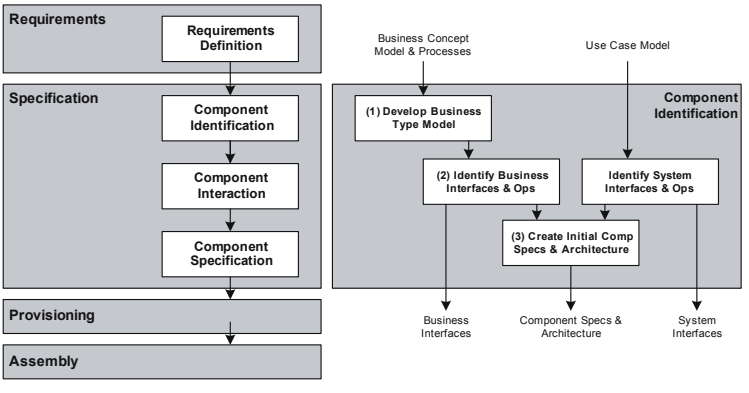


Fig. 1. UML Components process and component identification stage (cf. [12])

To identify business components, Cheesman and Daniels recommend to formalize the business concept model into a more detailed and technical *business type model* (see fig. 1 (1)). The next step is to identify so-called *core business types*, which represent information that can stand alone in the business domain. For each core business type, a business interface has then to be created (see fig. 1 (2)). A business interface has to manage the information represented by an independent core type and thus is a candidate to constitute a *business component*. The so identified business components finally have to be specified in detail and, together with identified system components, can be formed into an initial *systems architecture* (see fig. 1 (3)).

While this procedure may serve as a very heuristic approach to get to an initial set of business components, it has a variety of drawbacks. Firstly, even information that can stand alone in the business domain may likely have relationships to other information objects. Cheesman and Daniels acknowledge this and recommend to convert such

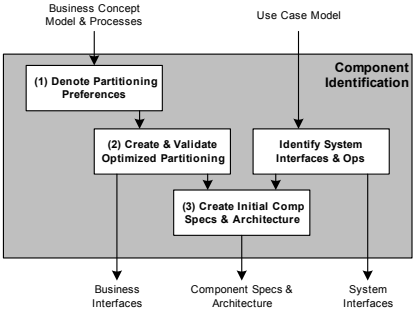


Fig. 2. Integration of the BCI method into the UML Components process

relationships into component dependencies. This might, however, not lead to an optimal partitioning, e.g., when components have to be easily replaceable and dependencies have to be kept at a minimum. Furthermore, the procedure is focused solely on a modularized management of information objects. Most business systems, however, will also have to include business components that manage entire processes or parts of business workflows [3]. These business components will then automatically have dependencies to all the business components governing relevant information in a process. Designers will hence have to take business functions *and* information (with their respective relationships to each other) into account when partitioning a problem space [10].

With the BCI method, the procedure proposed by Cheesman and Daniels can be replaced (see fig. 2). In line with their procedure, BCI also takes a business concept model and specified business processes as input to create a partitioning. In a first step, the designer will have to denote his partitioning preferences (see fig. 2 (1)). Thereafter, an optimized partitioning with respect to the given preferences is derived and has to be validated (see fig. 2 (2)). The resulting partitioning clusters process steps and information objects to form a set of business components. The identified business components are then to be refined, technically specified, and, together with the required system components, formed into a systems architecture (see fig. 2 (3)).

3 The Business Component Identification Method

The set of domain models and the defined metrics of maximizing cohesion and minimizing dependencies constitute the basis for the identification of business components. The identification is strongly dependent on the underlying domain models [14,15]. Only a domain model reflecting the business in a concise, complete and comprehensive way can lead to an adequate component model and therefore to a corresponding application system. In this paper we will not discuss the advantages and disadvantages of domain modeling methodologies. Instead, we will show how the information modeled in business domains can be used to identify business components in a formal way using the BCI method. Data from the domain of *Strategic Supply Network Development (SSND)* is used in the figures below to better visualize the identification process. The example domain comes from the area of strategic purchasing, where networks of suppliers are

analyzed and selected in order to define an adequate purchasing strategy. It is not our intention to explain the SSND example in this paper, we rather focus on the formal method for identifying business components using the data of the SSND example. For details about the SSND domain we refer to [16]. In the following, the single BCI process steps – (1) Denote Partitioning Preferences, (2) Create and Validate Optimized Partitioning, and (3) Create initial Component Specification and Architecture – introduced in fig. 2 will be described.

3.1 Denote Partitioning Preferences

The BCI method uses the information objects from the concept models and the process steps from the process diagrams of the business domain, including their relationships. One can distinguish between three types of relationships necessary for the identification of business components: the relationships between single process steps, the relationships between information objects, and the relationships between process steps and information objects. A relationship type distinguishes between subtypes expressing the significance of a relationship. E.g., a relationship between single process steps expresses – based on their cardinality constraints – how often a process step is executed and therefore how close two process steps are related to each other in that business domain. The relationships between information objects define how loosely or tightly the information objects are coupled, and the relationships between process steps and information objects define whether a corresponding information object is, e.g., used or created while executing the respective process step. All types of relationships are of great relevance and build the basis for the BCI method. In order to apply a formal method for the identification of business components, we map the domain models to a weighted graph. As the nodes represent information objects and process steps, the edges characterize the relationships between the nodes. Weights are used to define the different types and subtypes of relationships. They build the basis for assigning process steps and information objects to components. The mapping of information objects and process steps from the domain models to nodes in the weighted graph is straightforward. Whereas, the definition of the relationship subtypes and the assignment of weights to corresponding edges is heavily dependent on the importance of the relationships in the underlying domain models. Therefore, domain knowledge and know-how is required for this step and the designers need to denote their partitioning preferences by introducing relevant relationship subtypes and assigning weights to them. The relationship subtypes as well as the weights may therefore differ dependent on the domain models and the preferences specified by the designers.

The BCI-3D Tool was developed to support the application of the BCI method. Due to display reasons the weighted graph is visualized in a three-dimensional representation having the process steps and information objects arranged as nodes in circles. The nodes representing the information objects are shown on top of fig. 3, and the nodes representing the process steps are shown on the bottom of fig. 3. The edges representing the relationships between information objects connect the top nodes to each other, the ones representing the relationships between process steps connect the nodes on the

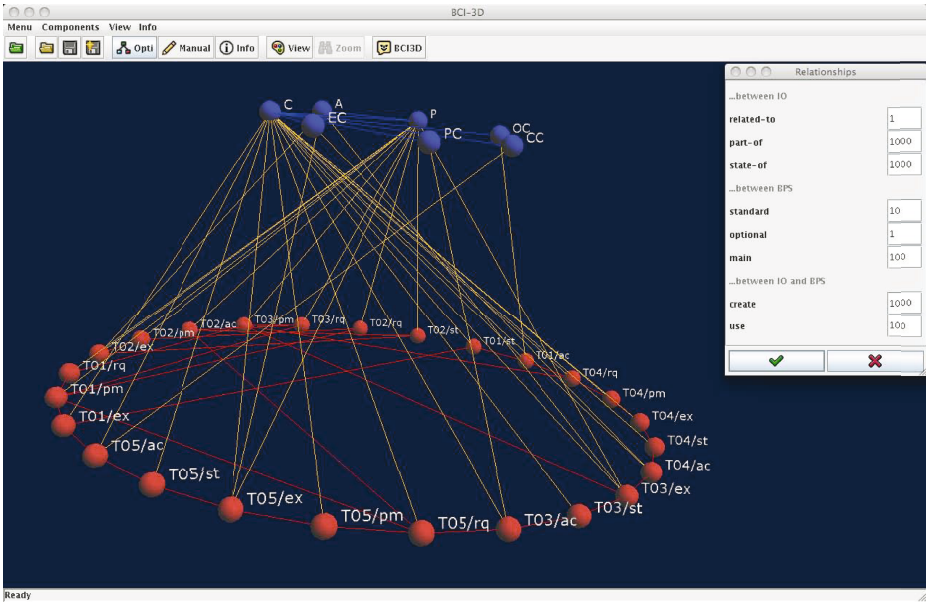


Fig. 3. BCI – defined preferences

Table 1. Assignment of process step names to shortcuts

process steps name	shortcut	process steps name	shortcut
request offering	T01/rq	state exploration	T03/st
promise offering	T01/pm	accept exploration	T03/ac
produce offering	T01/ex	request evaluation	T04/rq
state offering	T01/st	promise evaluation	T04/pm
accept offering	T01/ac	produce evaluation	T04/ex
request engineering	T02/rq	state evaluation	T04/st
promise engineering	T02/pm	accept evaluation	T04/ac
produce BoM explosion	T02/ex	request conclusion	T05/rq
state engineering	T02/st	promise conclusion	T05/pm
accept engineering	T02/ac	produce concluded contract	T05/ex
request exploration	T03/rq	state conclusion	T05/st
promise exploration	T03/pm	accept conclusion	T05/ac
produce contract	T03/ex		

bottom and the edges representing the relationships between information objects and process steps connect the nodes on top with the nodes on the bottom, shown in fig. 3. The weights assigned to the relationship subtypes are listed in a separate window on the right of fig. 3. Shortcuts are used to describe the process steps and information objects. The full names can be found in table 1 and table 2.

Table 2. Assignment of information object names to shortcuts

information object name	shortcut
Product	P
Assembly	A
Contract	C
Evaluated Contract	EC
Potential Contract	PC
Concluded Contract	CC
Offered contract	OC

3.2 Create and Validate Optimized Partitioning

For the identification of business components, as implemented by the BCI method, the weighted graph needs to be partitioned by assigning information objects and process steps to single components. The grouping should satisfy the defined metrics of maximizing cohesion and minimizing dependencies and should take all domain information into account which has been mapped to the weighted graph .

The problem of partitioning a graph $G = (V, E)$, with vertices V and edges E, into subsets of nodes of a defined size is known to belong to the class of NP-complete problems [17]. A clustering of the given example with 32 nodes into, e.g., three components of approximately equal size can be achieved in over 10^{12} different ways. Therefore, a direct calculation of the best solution by comparing all combinations is unreasonable, but heuristics can be used to find a best possible solution in suitable time. BCI applies an opening heuristic first, that gives a starting partition, and enhances this partition with an improving heuristic.

In general, a better starting partition is more likely to lead to better optimization results. We achieved the best results with the *Start Partition Greedy* heuristic shown in fig. 4. This is a greedy graph partitioning algorithm. In each iteration the most promising step is taken [18, p. 127]. The Start Partition Greedy utilizes a priority queue (PQ) to order the edges $e \in E$, whereby a higher priority is assigned to higher weighted edges. In the case of edges having equal weights, the weights of all edges adjacent to the end nodes are added. This allows for a fine-grained ordering. Beginning with unmarked vertices $v \in V$, the heuristic sequentially takes the edges in the PQ, and examines their end nodes. In the case of two unmarked nodes, a new component is generated. Whereas, in the case of one unmarked node, it is added to the marked node's component. Finally, all remaining unmarked nodes are collected in a last new component.

An advantage of our opening heuristic is, that there is no need to define the number of components in advance. It is determined by the Start Partition Greedy algorithm, depending solely on the given domain models and based on priority ordering of the edges. The idea is to cluster nodes, that are highly connected to their neighbors into one and the same component. An evaluation of different starting heuristics on various models, emphasized this method as leading to the most promising starting solutions.

After obtaining a primary solution for the optimization problem, various heuristics can be used to further improve the component structure. In 1970, Kernighan and Lin developed an algorithm for the enhancement of a given clustering of a graph into

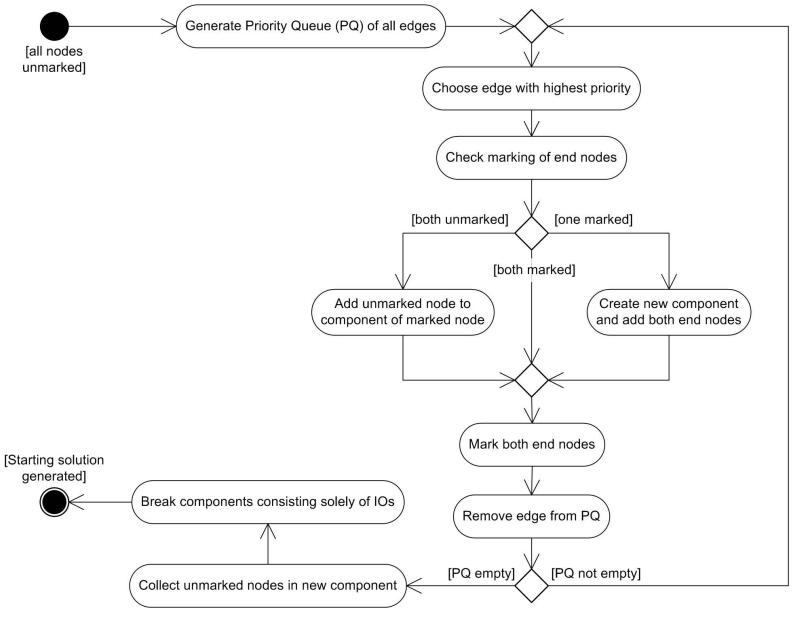


Fig. 4. UML Activity Diagram of the Start Partition Greedy heuristic

equal sized subgraphs [19]. Numerous variations of this method where proposed since then and all are based on the same concept (cf. [20,21,22]). We adopted the original Kernighan-Lin heuristics to improve the starting solution. This method does not consider all components at once, but rather a pair of two components in each step. At the beginning, all pairs are unmarked. In each step an unmarked pair is picked at random and the components are optimized, with respect to the heuristics. If any changes are made, all pairs are going to be unmarked again. Whereas, if no action is taken, the pair will be marked. This is repeated until no unmarked pairs are left and the component structure is optimized.

In order to compare different component structures and to be able to optimize them, we defined the cost $C(A, B)$ of a partitioning into components A and B as $C(A, B) = \sum_{a \in A, b \in B} w_{(a,b)}$, where $w_{(a,b)}$ is the weight of the connection between the single nodes $a \in A$ and $b \in B$. The goal is to minimize the cost of the partitioning for each pair of components (A, B). Furthermore, we defined internal $I(a)$ and external $E(a)$ costs of a node a according to Kernighan and Lin [19]:

$$I(a) = \sum_{x \in A, x \neq a} w_{(a,x)}, \quad E(a) = \sum_{b \in B} w_{(a,b)}$$

Moreover, the D-value of a node is referred to as the difference between its external and internal costs, $D(a) = E(a) - I(a)$. The gain $g(a, b)$ of exchanging the nodes a and b between the components A and B is then calculated by $g(a, b) = D(a) + D(b) - 2w_{(a,b)}$. The basic procedure of a two-component optimization step corresponds to Kernighan-Lin and is shown in fig. 5.

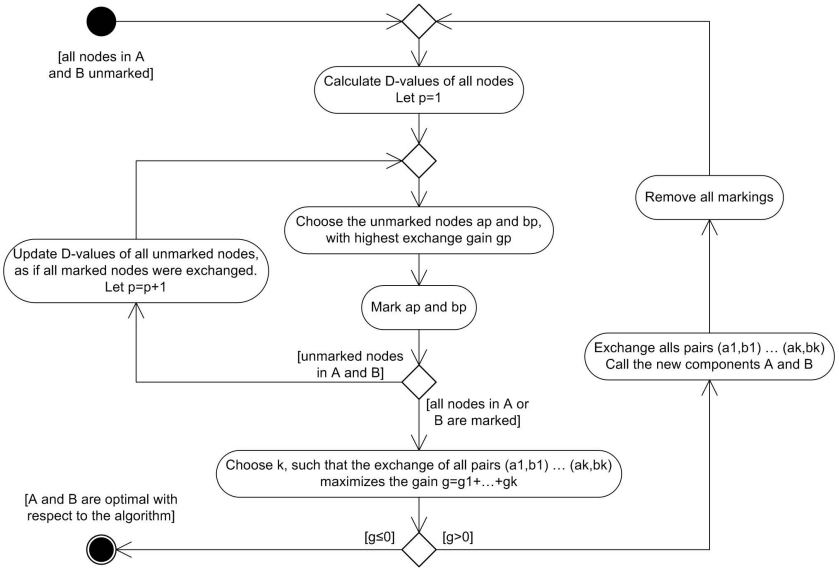


Fig. 5. UML Activity Diagram of the adopted Kernighan-Lin algorithm

The process of identifying business components by applying the BCI method and satisfying defined metrics is an iterative process. The business components resulting from BCI need to undergo a sensitivity analysis check before taken for granted. In analyzing the process steps and information objects assigned to the resulting components, inconsistencies and errors in the underlying domain models can be identified and corrected correspondingly. Additionally, the resulting component model should remain stable even if the weights in the weighted graph are slightly changed. By changing weights of the relationships and reapplying the BCI method, the stability of the resulting component model can be analyzed.

Applying the BCI method to the graph introduced in fig. 3 results in the following graph clustering (see fig. 6). The figure shows the identified business components and the dependencies between them. Additionally, the window on the right lists the single process steps and information objects as assigned to the identified components by BCI.

3.3 Create Initial Component Specification and Architecture

Two business components can be identified immediately. While looking at the process steps and information objects clustered within the components, the designer can identify the business functionality of the two business components: one containing the business tasks related to *Product Management* and one containing the business tasks related to *Contract Management*.

From fig. 6, the services provided and required by each component can be derived. We distinguish between two types of services: *inter-component services* and *information services*. Inter-component services are services, which are required by another component in order to provide a specific functionality. The inter-component services

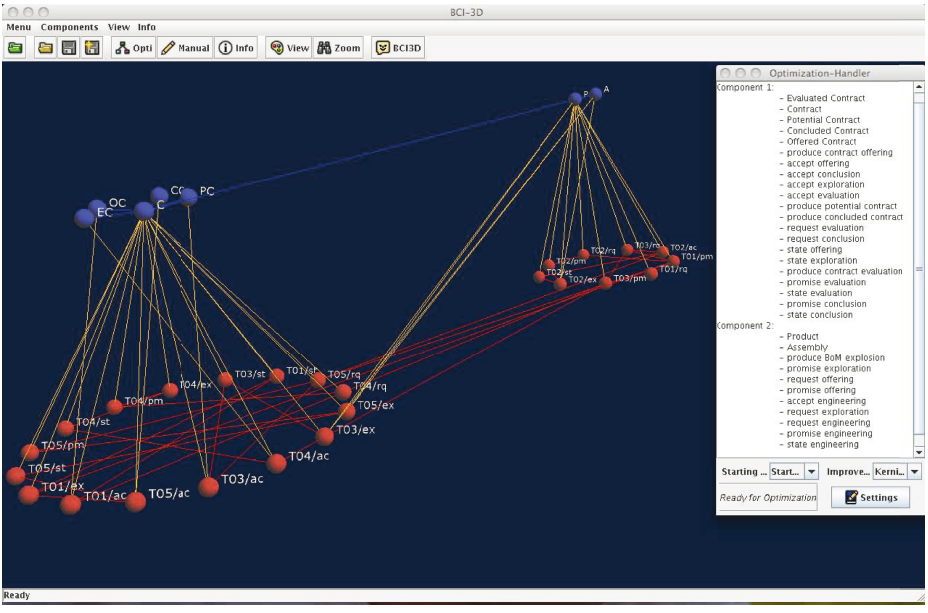


Fig. 6. BCI – optimized partitioning

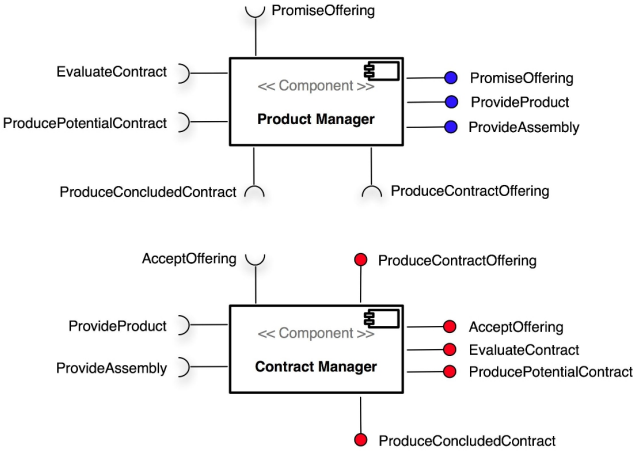


Fig. 7. UML Component Model of the identified components

are apparent in fig. 6 as the edges connecting two process steps, each located in a different component. E.g., the edge connecting the T01/pm (promise offering) and T05/rq (request conclusion) defines an inter-component service. The service provided by the Contract Manager component relates to the conclusion of the contract, and is therefore called ProduceConcludedContract. Which business component requires or provides that service becomes clear when looking at the process step diagrams of the relevant

business domain. The identified business components with their required and provided services are shown in fig. 7.

The second type of services gained from the business components identified and visualized in fig. 6 are information services. While information objects are created and updated by the responsible business component, other components need to request the values of required information objects through services. By analyzing the edges that connect process steps of one component with information objects of another component the services are identified. In fig. 6 we have e.g., T03/ex (produce potential contract) connected by an edge with A (Assembly). This means that the process step of producing a potential contract needs information about the assembly information object. In this case the Product Manager component needs to provide the service ProvideAssembly, while the Contract Manager component requires that service (see fig. 7).

4 Related Work

The identification of business components and their services is a primary research problem that needs to be addressed. Today, there is still little research contributing to the development of systematic approaches which support designers in finding an optimized set of business components. In accordance with the classification introduced by [23], mainly three different types of business component identification techniques can be distinguished: *Domain Engineering* based methods, *CRUD (Create, Read, Update, Delete) matrix* based methods and *Cohesion-Coupling based Clustering Analysis* methods.

A key issue in the design phase of the domain engineering process is "the generation of components that represent conceptual, functional and technological aspects of the domain, and their organization within a domain architecture" [24]. Given that fact, Domain Engineering based methods for component identification usually focus on the reusability of the domain architecture and the adaptability of constituent components, based on defined criteria. E.g., the Feature-Oriented Reuse Method (FORM) [25] captures commonality selectable for different applications as an AND/OR graph, where AND nodes indicate mandatory features and OR nodes indicate alternative features. This graph is used to define parameterized reference architectures and reusable components instantiable during application development. Another approach aims at gathering components that intensively exchange messages in a unique artifact, and defining an architecture element referred to as components grouping [24]. It uses defined criteria for the grouping of components based on four different aspects: domain context, process component, components interfaces, and the component itself. Domain Engineering based methods, however, rarely use formal approaches to obtain reusable components and are highly dependent on the experiences of the designers.

CRUD matrix based methods focus on the semantics of business elements, which is contained in domain models, to merge closely related elements into business components. They use the relationships between behavioral business elements (e.g., process steps) and static business elements (e.g., information objects) to define how closely the elements are related to each other. Four relationship types – Create (C), Read (R), Update (U) and Delete (D) with the priority C>D>U>R – are used to specify the semantic relationship between the behavioral and the static business elements. The relationships

are visualized in a matrix. CRUD matrix based methods aim at transforming the matrix by given rules in order to identify blocks in which behavioral and static business elements with C and D relationships are merged to form single components. Examples of CRUD matrix based methods are [26,27]. The disadvantage of CRUD matrix based approaches is that additional information available in the domain models is not used for identifying business components. E.g., the relationships between static elements and the relationships between behavioral elements are not considered at all.

With Cohesion-Coupling based Clustering Analysis methods, researchers try to cluster business models according to high cohesion and low coupling principles, and encapsulate each cluster in a component. The main idea of those methods is to first transform the domain models into the form of weighted graphs, in which business elements are nodes, the dependencies between single business elements are edges and semantic dependency strengths are represented as weights. In a second step, the graph is clustered using graph clustering or matrix analysis techniques that satisfy the metrics of high cohesion and low coupling. E.g., [28,29] are implementing such clustering analysis methods in order to identify components. Both approaches assume that UML models are available describing the business domain. The disadvantage of such approaches is that they are often based on technical concepts defined, e.g., in UML instead of focusing on the semantics of the corresponding business domain.

The BCI method directly contributes to the research area of identifying business components. According to the classification of business components identification methods by [23], the BCI method combines Cohesion-Coupling based Clustering Analysis and CRUD matrix based methods. The advantage of BCI is that the method uses all relevant dependencies of business domain models, including relationships between behavioral business elements, between static business elements, and those between behavioral and static business elements. It therefore extends CRUD matrix based methods with two additional types of dependencies. Additionally, BCI maps those business elements and their mentioned dependencies, independently of the notation used to model the business domain and its technical concepts, into a weighted graph. This graph is then used to apply the Cohesion-Coupling based Clustering Analysis methods implemented in BCI for identifying business components. With BCI, we thus satisfy Wang's recommendation of *combining* current component identification methods in order to achieve better results [23].

5 Conclusions and Future Directions

In this paper, we described the BCI method and have shown how to integrate it into the UML Components development process. The BCI method creates a partitioning of a problem space into business components which are optimized to satisfy the designers' partitioning preferences. In doing so, we advance the current state of the art and contribute to establish a more systematic approach to partition business systems into components, a key task in component-based systems development.

The BCI method was created in a perennial research project and has been continually improved to reach the scope of operation presented in this paper. It already has been *validated* in complex case studies that confirm its appropriateness for the development of

component-based business systems in practice [30,31,32]. While the algorithms used in the current method and the derived partitioning results have proven to be mature, several approaches to further the scope of operation are currently under development. Among others, it is the plan to empirically evaluate the BCI method versus the other component identification methods described in this paper in order to show that the approach presented is superior to alternative approaches. Additionally, the initiative to integrate the BCI method into a tool that covers the domain modeling process is ongoing. With the partitioning as a final result, the tool may lead over to a component-based system design as well as to a service-based development approach. More technically motivated research initiatives include an automatic derivation of *component orchestrations* as well as the generation of parts of the components' internal structure.

In future, we plan to extend the BCI method to support *reuse-driven development approaches*, in which existing components will be considered. To reuse existing components during the partitioning process, we require conceptual models of process steps and information objects managed by those components. These models are either derived from technical specifications or already available when building upon more holistic specification approaches like, e.g., the Unified Specification of Components approach [33]. Existing components will then be represented as clusters of process steps and information objects that are marked to remain unchanged during the partitioning.

Our research initiatives centered around the BCI method are part of a longer-term goal to provide a mature methodical support of the partitioning process, just as it will become available for the complementary composition process. Only with an appropriate support of *both* processes, component-based development will lead to a component-based software engineering process.

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software. Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Harlow (2002)
2. Sametinger, J.: Software Engineering with Reusable Components. Springer, Heidelberg (1997)
3. Herzum, P., Sims, O.: Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise. John Wiley & Sons, New York (2000)
4. Brown, A.W.: Large-Scale, Component-Based Development. Prentice Hall, Upper Saddle River (2000)
5. Zaremski, A.M., Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology 4(2), 146–170 (1995)
6. Seacord, R.C., Hissam, S.A., Wallnau, K.C.: Agora: A Search Engine for Software Components. Technical report CMU/SEI-98-TR-011, Software Engineering Institute, Carnegie Mellon University (1998)
7. Yellin, D., Strom, R.: Protocol Specifications and Component Adaptors. ACM Transactions on Programming Languages and Systems 19(2), 292–333 (1997)
8. Wallnau, K.C.: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institue (2003)
9. Reussner, R.H., Schmidt, H.W.: Using Parameterised Contracts to Predict Properties of Component-Based Software Architectures. In: Crnkovic, I., Larsson, S., Stafford, J. (eds.) Workshop on Component-Based Software Engineering, Lund (2002)

10. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
11. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall, Englewood Cliffs (1997)
12. Cheesman, J., Daniels, J.: UML Components. A Simple Process for Specifying Component-Based Software. Addison-Wesley, Upper Saddle River (2001)
13. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML. The Catalysis Approach. Addison-Wesley, Upper Saddle River (1999)
14. Albani, A., Dietz, J.L.: The benefit of enterprise ontology in identifying business components. In: IFIP World Computing Conference, Santiago de Chile, Chile (2006)
15. Albani, A., Dietz, J.L., Zaha, J.M.: Identifying business components on the basis of an enterprise ontology. In: Konstantas, D., Bourrieres, J.P., Leonard, M., Boudjlida, N. (eds.) Interoperability of Enterprise Software and Applications, Geneva, Switzerland, pp. 335–347. Springer, Heidelberg (2005)
16. Albani, A., Müssigmann, N., Zaha, J.M.: A Reference Model for Strategic Supply Network Development. In: Reference Modeling for Business Systems Analysis, Idea Group Inc. (2006)
17. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified np-complete problems. In: STOC 1974: Proceedings of the sixth annual ACM symposium on Theory of computing, pp. 47–63. ACM, New York (1974)
18. Jungnickel, D.: Graphs, Networks and Algorithms, 3rd edn. Springer, Berlin (2007)
19. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell system technical journal 49, 291–307 (1970)
20. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: DAC 1982: Proceedings of the 19th conference on Design automation, Piscataway, NJ, USA, pp. 175–181. IEEE Press, Los Alamitos (1982)
21. Dutt, S.: New faster kernighan-lin-type graph-partitioning algorithms. In: ICCAD 1993: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design, pp. 370–377. IEEE Computer Society Press, Los Alamitos (1993)
22. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proceedings of the 1995 ACM/IEEE conference on Supercomputing. ACM, New York (1995)
23. Wang, Z., Xu, X., Zhan, D.: A survey of business component identification methods and related techniques. International Journal of Information Technology 2, 229–238 (2005)
24. Blois, A.P.T., Werner, C.M., Becker, K.: Towards a components grouping technique within a domain engineering process. In: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005) (2005)
25. Kang, K.C., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
26. Lee, S., Yand, Y.: Como: A uml-based component development methodology. In: Proceedings of the 6th Asia Pacific Software Engineering Conference, pp. 54–63 (1998)
27. Somjit, A., Dentcho, B.: Development of industrial information systems on the web using business components. Computer in Industry 50, 231–250 (2003)
28. Kim, S.D., Chang, S.H.: A systematic method to identify software components. In: 11th Asia-Pacific Software Engineering Conference (APSEC), pp. 538–545 (2004)
29. Jain, H., Chalimeda, N.: Business component identification - a formal approach. In: Proceedings of the Fifth International Enterprise Distributed Object Computing Conference (EDOC 2001). IEEE Computer Society, Los Alamitos (2001)
30. Albani, A., Bazijanec, B., Turowski, K., Winnewisser, C.: Component framework for strategic supply network development. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 67–82. Springer, Heidelberg (2004)

31. Selk, B., Kloeckner, S., Bazijanec, B., Albani, A.: Experience report: Appropriateness of the bci-method for identifying business components in large-scale information systems. In: Turowski, K., Zaha, J.M. (eds.) Component-Oriented Enterprise Applications, Proceedings of the Conference on Component-Oriented Enterprise Applications (COEA 2005), Bonn, Köllen. Lecture Notes in Informatics, vol. 70, pp. 87–92 (2005)
32. Eberhardt, A., Gausmann, O., Albani, A.: Case study automating direct banking customer service processes with service oriented architecture. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2006 Workshops. LNCS, vol. 4277, pp. 763–779. Springer, Heidelberg (2006)
33. Overhage, S.: UnSCom: A Standardized Framework for the Specification of Software Components. In: Weske, M., Liggesmeyer, P. (eds.) NODe 2004. LNCS, vol. 3263, pp. 169–184. Springer, Heidelberg (2004)

Life-Cycle Aware Modelling of Software Components

Heiko Koziolek[1], Steffen Becker[3], Jens Happe[2], and Ralf Reussner[2]

[1] ABB Corporate Research
Wallstadter Str. 59, 68526 Ladenburg, Germany
[2] Chair for Software Design and Quality
Am Fasanengarten 5, University of Karlsruhe (TH), 76131 Karlsruhe, Germany
[3] FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
{koziolek,sbecker,happe,reussner}@ipd.uka.de

Abstract. Current software component models insufficiently reflect the different stages of component life-cycle, which involves design, implementation, deployment, and runtime. Therefore, reasoning techniques for component-based models (e.g., protocol checking, QoS predictions, etc.) are often limited to a particular life-cycle stage. We propose modelling software components in different design stages, after implemenatation, and during deployment. Abstract models for newly designed components can be combined with refined models for already implemented components. As a proof-of-concept, we have implemented the new modelling techniques as part of our Palladio Component Model (PCM).

1 Introduction

Methods for model-based reasoning about component-based software architectures shall enable software architects to assess functional properties (e.g., correctness, interoperability, etc.) and extra-functional (e.g., performance, reliability, etc.) properties already during design [20]. By composing individual component specifications and running different analysis and simulation tools, the properties of the whole system shall be evaluated based on the properties of its individual parts. These methods shall avoid the implementation of designs which exhibit insufficient functional or extra-functional properties.

During component-based system design, software architects specify new components and incorporate existing components in their architectures [4]. To support this mixed (i.e., top-down and bottom-up) development process, modelling and analysis methods must account for different stages in the component life-cycle. A step-wise refinement of component specifications is desirable as components progress from the design to implementation stage.

Existing models for component-based systems support different stages in the component life-cycle only insufficiently [9]. Industrial component models, such as EJB [6], COM [5], or CCM [14], only refer to component implementations, but not to component designs. Furthermore, their support for functional and extra-functional analysis is limited.

We propose modelling software components during different design stages and allow combining coarse specifications of new components with refined specifications of

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 278–285, 2008.

already implemented components to improve functional and extra-functional analysis. When the development of a component-based system progresses, coarse models of individual components can be refined with additional information thereby increasing the accuracy of analysis methods. Our approach improves functional and extra-functional reasoning for component-based software architectures, as it better reflects the different life-cycle stages of a software component than existing approaches.

The contributions of this paper is a component type hierarchy that enables modelling software components at different design stages. We meta-modelled both concepts (which should be included into other component models) and added them to our Palladio Component Model (PCM) [2]. To illustrate the benefits of our approach, we model the components of a business reporting component during different stages of their life-cycle in this paper.

This paper is organised as follows. Section 2 discusses related work, before Section 3 sketches the component-based development process with separated developer roles. Section 4 introduces our concepts for modelling software component during different development stages. Section 5 shows how the new concepts were implemented in the PCM. Finally, Section 6 concludes the paper.

2 Related Work

We compare the component type hierarchy proposed in this paper with common definitions of software components (i.e., [20,4,9]), to different realisations of component definitions in current software component models, and to ADLs [11].

Common Component Definitions: Szyperski's well-known definition of software components [20] does not explicitly distinguish between component type and implementation. It mainly states how a component should be specified with provided and required interfaces, but does not refer to the whole life-cycle of a software component.

Cheesman et al. [4] distinguish four different stages in the component life-cycle: component specification, component implementation, installed component, and component object. Our component type-hierarchy supports modelling components and reasoning on their properties in the first two of these stages (i.e., with complete component types, basic components). The PCM also contains a context model (not described in this paper) to describe installed components, and component objects. In addition to Cheesman's viewpoint, we distinguish between different stages of component specification.

Lau et al. [9] distinguish between component design, deployment, and runtime as the stages of component life-cycle. Lau's view does not include reasoning for mixed architectures of software components modelled at the design or implementation stage.

Software Component Models: The following briefly analyses UML, industrial component models, and component models from research [9]. The UML [13] supports modelling software components with component diagrams. With additional UML profiles (e.g., UML SPT [12]), designers may also specify QoS attributes to reason about extra-functional properties. However, the UML does not explicitly support modelling different life-cycle stages of a software component.

Component models used in industry, such as EJB [6], COM [5], and CCM [14], target the implementation of component-based systems, and do not explicitly support early reasoning about component-based designs. Fractal [15] is a component model targeting the runtime stage of software components. There is no type hierarchy for Fractal components, as it is assumed that an implementation of each component is available. SOFA [16] does not distinguish between different design stages. ROBOCOP [3] targets performance prediction for embedded, component-based software architectures. There are no different design stages for software components in ROBOCOP.

Architecture Description Languages: Medividovic and Taylor have provided a classification and comparison framework for ADLs [11]. While all ADLs differentiate between component types and component instances, only a few of them provide facilities for refining component specifications according to their life-cycle. For example, Aesop [7] allows component subtypes and enforces preservation of component behaviour. C2 [10] supports different subtyping relationships for interfaces, behaviours and implementations. However, these approaches are tied to object-oriented inheritance relationships and do not explicitly distinguish between discrete component life-cycle stages.

3 Component-Based Development Process

The component-based development process involves several developer roles with specific responsibilities. The following roles are particularly relevant in our setting [8]:

- **Component Developers** specify and implement software components either from scratch or using existing components. They develop components for a market as well as per request. They make as few as possible assumptions about a specific deployment environment to ensure broad reuse.
- **Software Architects** lead the development process for a component-based application. They design the software architecture and delegate tasks to other involved roles. For the design, they decompose the planned application's specification into single component specifications.
- **QoS Experts** collect QoS-relevant information from the different developer roles and assess the extra-functional properties of the system.

In practice, the process has to consider the desired reuse of components as well as new requirements. Software architects can use existing components from repositories or specify new ones for specific requirements, which shall be implemented by component developers. During the specification of a software architecture, some of the used components are already specified and implemented while others are only sketched. As a consequence, the component-based development process does not follow a strict separation into classical top-down (i.e., going from requirements to implementation) and bottom-up (i.e., assembling existing component to create an application) categories. Instead, it is a mixture of both approaches.

Any software component model should account for the mixed top-down and bottom-up development process. It is beneficial as it allows software architects to reason about the properties of their architecture during early development stages when some components are not yet implemented, but at the same time allows to rely on refined models

of already implemented (e.g., third-party) components. However, at this stage it is less costly to change design decisions.

4 Component Design

To support model-based reasoning about component-based designs in a mixed top-down and bottom-up development process, it is necessary to model components in different development stages. It must be possible to successively refine components from early development stages. There are at least three different stages of component specification as depicted in Fig. 1 and described in the following from top to bottom.

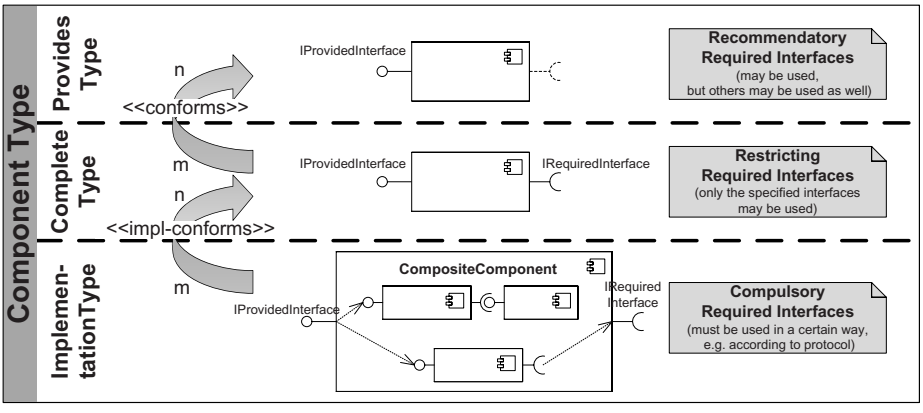


Fig. 1. Component specification in different development stages

4.1 Bundling Provided Services into Components

At the first stage, software architects specify components based on their desired functionality using provided interfaces. The architects might be unsure, which other components are required to provide this functionality, but nevertheless they want to include the desired functionality in their model for early reasoning.

We call such component specifications, which include provided interfaces, but only optionally include required interfaces *provides component types*. These components are merely stubs for reasoning and can for example contain estimated QoS-annotations (e.g., execution times, failure probabilities) for early QoS predictions. Fig. 2 shows the example of a component specified for business reporting functionality and includes estimated execution times as QoS-annotations.

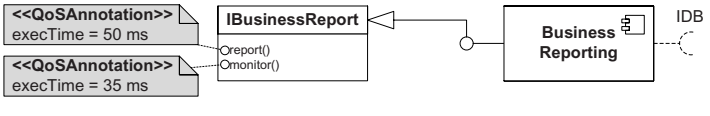


Fig. 2. Example of a Provides Component Type

4.2 Full Specification of Required Services

At the second stage, the software architect refines components with interfaces needed to provide a certain functionality. In this stage, the implementation of the component is still unknown and there are multiple possibilities to realise a component conforming to the specified interfaces. For example, component developers can use different algorithms and data structures behind the same interfaces. The specified required interfaces in this stage can (but need not) be used by component developers implementing the component. However, they must not use additional required interfaces in order to remain type-conform.

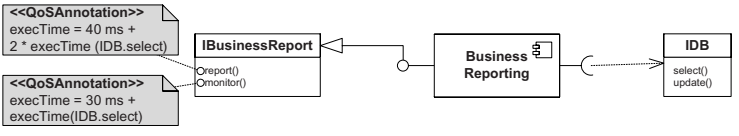


Fig. 3. Example of a Complete Component Type

We call such component specifications, which include provided interfaces and required interfaces, *complete component types*, as all their interfaces are known. Software architects can pass these component specifications to component developers as requirements specifications. A complete component *conforms* to a provided component (and thus can substitute it), if and only if it provides at least the services specified in the provided type. With *complete component type*, functional and extra-functional analysis can be refined, as for example estimated QoS-annotations can now also refer to required services, as depicted in Fig. 3.

4.3 Modelling Component Implementations

At the third stage, a component specification has been implemented, and a model of the implementation (with refined information) should be included in the architectural design model to improve the accuracy of analyses. Developers can either assemble other components to implement a component (i.e., a so-called *composite component*) or directly implement them (i.e., a so-called *basic component*).

The models of these component implementations can be refined with service effect specifications (SEFF) (cf. Fig. 4), which are a high-level abstractions of the behaviour of component services and model how provided services of a component call the required interfaces. SEFFs are useful for many kinds of functional and extra-functional analysis (e.g., protocol checking [17], reliability prediction [18], performance prediction [2], testability [19]).

A basic or composite component *impl-conforms* to a complete type (and thus can substitute it) if and only if it provides at least the services specified in the provided interfaces of the complete type and it requires at most the services specified in its the required interfaces of the complete type. This principle is known as contra variance [20]. The *conforms* as well as the *impl-conforms* are n:m relations. Each basic or composite

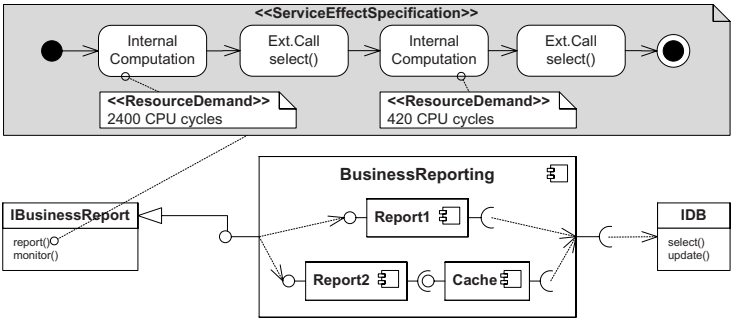


Fig. 4. Example of a Composite Component

component can conform to multiple complete types and each complete type can be implemented multiple times.

5 Implementation

Fig. 5 shows the realisation of the formerly described component type hierarchy in the PCM meta-model. Abstract meta-classes are colored in light grey. We have introduced an explicit abstract class for the concept of providing and requiring an interface, as it is common for all types of components.

PCM `Interfaces` mostly follow the syntax and semantics of CORBA IDL [14], therefore we omit the full meta-model for interfaces for clarity. PCM `Interfaces` are first-class entities and may exist independently from components. The specification of a `RequiredRole` to an `Interfaces` has different semantics according to the underlying component type (i.e., recommended, restricted, or compulsory as described above).

Meta-classes for QoS annotations (e.g., for provides and complete component type) have been omitted for brevity. Component developers specify QoS properties of

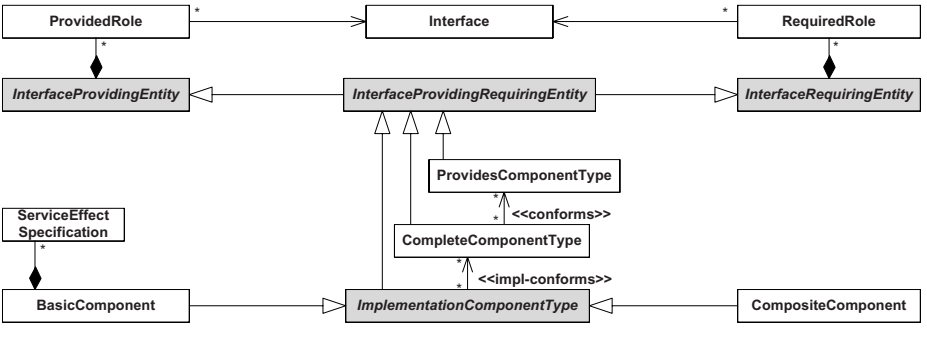


Fig. 5. PCM Component Type Hierarchy (Meta-Model)

BasicComponents using ServiceEffectSpecifications. Their metamodel is extensively described in [2]. Tools can compute the QoS properties of CompositeComponents by combining the ServiceEffectSpecifications of the included BasicComponents.

6 Conclusions

We have proposed a refined modelling of component types during different development stages to improve early analysis of functional and extra-functional properties. The different component type levels allow reasoning on the properties of software architectures with already implemented and only designed components. This reflects the typically mixed (top-down and bottom-up) development process of component-based systems.

During component deployment, the PCM supports modelling contextual information for each component instance, such as the binding to other components, the allocation to hardware resources, and the usage of the components. This context model has been detailed in [1]. For the future, we plan to extend the PCM's context model to hold more refined contextual information for QoS predictions with higher accuracy. Furthermore, modelling specifics of the runtime stage of components would allow even more kinds of predictions.

References

1. Becker, S., Happe, J., Koziolek, H.: Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R., Szyperski, C., Weck, W. (eds.) Proceedings of the Eleventh International Workshop on Component-Oriented Programming (WCOP 2006) (June 2006)
2. Becker, S., Koziolek, H., Reussner, R.: Model-based Performance Prediction with the Palladio Component Model. In: Proceedings of the 6th International Workshop on Software and Performance (WOSP 2007), February 5–8, 2007, ACM Sigsoft (2007)
3. Bondarev, E., de With, P.H.N., Chaudron, M.: Predicting Real-Time Properties of Component-Based Applications. In: Proc. of RTCSA (2004)
4. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-based Software. Addison-Wesley, Reading (2000)
5. Microsoft Corp. The COM homepage (last retrieved 2006-10-30), http://www.microsoft.com/
6. Sun Microsystems Corp., The Enterprise Java Beans, homepage (Last retrieved 2008-01-06) (2007)
7. Garlan, D., Allen, R., Ockerbloom, J.: Exploiting style in architectural design environments. SIGSOFT Softw. Eng. Notes 19(5), 175–188 (1994)
8. Koziolek, H., Happe, J.: A Quality of Service Driven Development Process Model for Component-based Software Systems. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 336–343. Springer, Heidelberg (2006)
9. Lau, K.-K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering 33(10), 709–724 (2007)

10. Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: Using object-oriented typing to support architectural design in the c2 style. In: SIGSOFT 1996: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, pp. 24–32. ACM, New York (1996)
11. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1), 70–93 (2000)
12. Object Management Group (OMG). UML Profile for Schedulability, Performance and Time (January 2005)
13. Object Management Group (OMG). Unified Modeling Language Specification: Version 2, Revised Final Adopted Specification (ptc/05-07-04) (2005)
14. Object Management Group (OMG). CORBA Component Model, v4.0 (formal/2006-04-01) (2006)
15. Object Web. The Fractal Project Homepage (Last retrieved 2008-01-06) (2006)
16. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering 28(11), 1056–1076 (2002)
17. Reussner, R.H.: Automatic Component Protocol Adaptation with the CoCoNut Tool Suite. Future Generation Computer Systems 19, 627–639 (2003)
18. Reussner, R.H., Schmidt, H.W., Poernomo, I.: Reliability Prediction for Component-Based Software Architectures. Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes 66(3), 241–252 (2003)
19. Stafford, J.A., McGregor, J.D.: Top-down analysis for bottom-up development. In: Proc. 9th International Workshop on Component-Oriented Programming (WCOP 2004) (2004)
20. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. ACM Press and Addison-Wesley, New York (2002)

A Component Selection Framework for COTS Libraries

Bart George, Régis Fleurquin, and Salah Sadou

VALORIA Laboratory, University of South Brittany, 56017 Vannes, France
{george,fleurqui,sadou}@univ-ubs.fr
http://www-valoria.univ-ubs.fr/

Abstract. Component-based software engineering proposes building complex applications from COTS (Commercial Off-The-Shelf) organized into component markets. Therefore, the main development effort is required in selection of the components that fit the specific needs of an application. In this article, we propose a mechanism allowing the automatic selection of a component among a set of candidate COTS, according to functional and non-functional properties. This mechanism has been validated on an example using the *ComponentSource* component market.

1 Introduction

Component-Based Software Engineering allows developers to build a system from reusable pre-existing commercial off-the-shelf (COTS) components. The two immediate potential benefits for such an approach are reduced development costs and shorter time-to-market [1]. For this reason, more and more software applications are built using COTS rather than being developed from scratch, as this is something that fewer and fewer companies can afford [2]. However, due to the intrisic nature of COTS as "black-box" units put into markets by third party publishers, software development life-cycle must be rethought in depth [3,4]. In fact, COTS-based software development leads to constant trade-offs between requirement specification, architecture specification and COTS selection [5]. In this context, it becomes impossible to specify requirements without asking if the marketplace provides COTS that can satisfy them. And one cannot specify an architecture without asking if there are COTS to integrate it.

In such a context, COTS selection becomes particularly important [6]. So important that a bad requirements definition associated to a poor selection of COTS products can lead to major failures [7]. There are also extra costs due to the investigation of hundreds of candidates disseminated into several different markets and libraries, not to mention the diversity of components' description formats. Finally, this phase can become so time-consuming that it may annihilate the initial promise of cost and time reductions [6]. Therefore, the only solution to maintain these gains is to have a selection process [1] that would be well-defined, repeatable, and as automated as possible.

In this paper, we propose a mechanism that allows application designers to select, among a vast library of candidates, the one that best satisfies a specific

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 286–301, 2008.

need, modeled by a virtual component called a "target component". Section 2 will detail existing approaches, as well as their limits. In section 3 we will present our own approach. Then, before concluding, in section 4 we will present a validation of this approach using *ComponentSource* [8] as a component marketplace.

2 COTS Selection Techniques

The issue is the following one: given a vast number of COTS components from all origins, disseminated in several different markets, how can the one that will best satisfy an application's specific need be chosen ?

2.1 Presentation of Current Selection Processes

Works in the field of component selection are trying to answer this fundamental question. C. Güngör En and H. Baraçli [6] listed many of these works. This study shows that most selection processes provide at least the three following phases: evaluation criteria definition, prioritization of these criteria, and COTS candidates' evaluation according to these criteria. Usually, in order to achieve these phases, selection processes use multi-criteria decision making techniques (MCDM). The most used MCDM techniques are Weighted Scoring Method or WSM [9] and Analytic Hierarchy Process or AHP [10]. WSM consists in using the following formula: $score_c = \Sigma_{j=1}^{n}(weight_j * score_{cj})$, where weight $weight_j$ represents the importance of j-th criterion compared to the n-1 other evaluation criteria, and local score $score_{cj}$ evaluates the satisfaction level of the j-th criterion by candidate c. Thus, total score $score_c$ represents the global evaluation value for candidate c. Therefore, the best candidate is the one that has the highest total score. AHP is a technique that organizes the definition and prioritization of evaluation criteria. It consists in decomposing a goal in a hierarchical tree of criteria and sub-criteria whose leaves are available candidates. Inside each criteria-node, the importance of each sub-criterion is estimated compared to others. For example, the criterion "performance" can be divided in two sub-criteria "response time" and "resource consumption", the first sub-criterion having a weight twice higher than the second one. Then, one can use a formula such as WSM to evaluate each candidate c by aggregating local scores $score_{cj1}, ..., score_{cjn}$ inside each node j, and propagating all these sums to the root of the tree to get c's total score.

Now, let us take a look at the contributions made by main works in the field of component selection. OTSO [11] is considered as one of the first selection processes dedicated to COTS components. In addition to the three phases described above, it adds other ones such as pre-selection of COTS to identify potentially relevant candidates and limit their number (therefore it acknowledges the difficulty to manually evaluate too many candidates). PORE [7] is a selection process that pleads in favor of a progressive selection. Candidates are filtered and their number decreases while the description of the needs becomes more accurate. DEER [12] is aimed at selecting single components, or assemblies

of components, which satisfy requirements while minimizing costs. Other processes propose new steps in order to facilitate selection. For example, PECA [13] adds an extra phase : evaluation planning. It consists in choosing the people responsible for the evaluation of candidates and the techniques they will use. Other approaches focus on the definition of evaluation criteria [14]. For example, STACE [15] proposes taking into account "socio-technical" criteria. Such criteria can be, for instance: product quality, product technology, business aspects (for example, supplier reputation on the marketplace), etc... BAREMO [16] adapts AHP to COTS by defining a set of specific criteria and sub-criteria dedicated to these kind of components. COTSRE [17] proposes to create reusable "criteria catalogs". And CAP [18] proposes specific non-functional criteria inspired by ISO-9126 quality standard [19].

2.2 Limits of These Approaches

The main inconvenience of these processes is their lack of automation. Even if candidates' total scores are calculated with an automated formula such as WSM, local scores $score_{cj}$ are estimated manually by evaluators for each candidate c. Coming back to the example of sub-criteria "response time" and "resource consumption", it is clear that if we want a precise evaluation, it would be much better to measure them automatically with the help of metrics instead of letting a user enter arbitrary local scores. Furthermore, even if we limited ourselves to only one market, or a particular section of a market, we would face more than one hundred candidates anyway. For instance, the single *ComponentSource*'s "internet communication" section [8] contains more than 120 candidates. Therefore, it is important to automate local score measurements as much as possible. It is not only a matter of precision, but also an efficient way to deal with a huge amount of information. A high number of candidates becomes quickly fastidious in the case of a manual evaluation [20].

All local score calculations are not automatable the same way, though. Preselection phase usually uses a small number of general criteria, such as keywords. In this case, local score calculations are simple, but they can apply to a large number of candidates. Component search and retrieval techniques [21], whose goal is to formulate a specific query and then retrieve all the components matching this query, provide adapted algorithms for this kind of local scores, for instance, keyword search or facet-based classification [22]. However, during detailed evaluation phase, there can be many complex criteria, each one concerning a specific property (signature matching, metric value comparison...). In this case, local score calculations are much more complex because they require the aggregation of many values of different nature, but they apply to a smaller number of candidates. Fine-grained comparisons such as signature subtyping [23] fit this kind of comparison. And non-functional properties, in order to be evaluated, can be described with techniques such as quality of service contracts [24,25,26] or COTS-based quality models [27].

The mechanism we propose allows for the automation of these local score calculations by taking into account the need for flexibility on the criteria detail

level. The originality of our approach consists in automating COTS selection by using existing works from other domains. All these techniques cohabitate into a unique concept: target component.

3 Component Selection

Our selection approach takes place in a component-based software development context, as defined in [3]. In this context, a component-based application is built incrementally. When a component is added into the application, it brings its own constraints. Then, its required interfaces become part of the new requirements that must be satisfied by the next component to be integrated. Therefore, the application's current requirements are dictated, among other things, by components currently integrated in it.

We chose to model the application's requirements by virtual "target" components. A target component represents the "ideal" answer for a specific need, and has to be replaced by the closest "concrete" candidate component. Evaluation criteria are components' functional and non-functional properties. Such a representation allows the designer to have criteria that are closer to the application's true needs. It also allows for the use of many techniques dedicated to automatic component search and comparison. However, such a mechanism implies two problems: i) the choice of a description format for candidate components as well as target ones; ii) the definition of a comparison function for such component descriptions to measure their "similarity". In this section, we will successively present the solutions we propose to address these two problems.

3.1 Component Description Format

Nowadays, there is no consensus on component description format. Each market has its own way to document its components, often developed from several different models. For instance, *ComponentSource* stores ActiveX, JavaBeans or .NET components. However, all candidates must be compared to a target component according to a same description format. Furthermore, this format must be abstract enough to encompass concepts that are common to most existing models. This is why we defined our own format dedicated to COTS components. It is described with a UML model (figure 1), whose elements will be presented in the following pages.

Architectural artifacts. Three kinds of artifacts have been selected: components, interfaces, and operations. Components contain two sets of interfaces (provided and required), and interfaces are constituted by a set of operations. This representation is inspired by the standard definition used by many models such as UML 2.0 [28]. As COTS components are represented as "black-boxes", we will not take into account "composite" components.

For each (target) operation, we associate several signatures. It is useful to anticipate many to improve performance during the search for a specific service. A signature $S = ParamTypes \rightarrow ResultType$ details parameters' types,

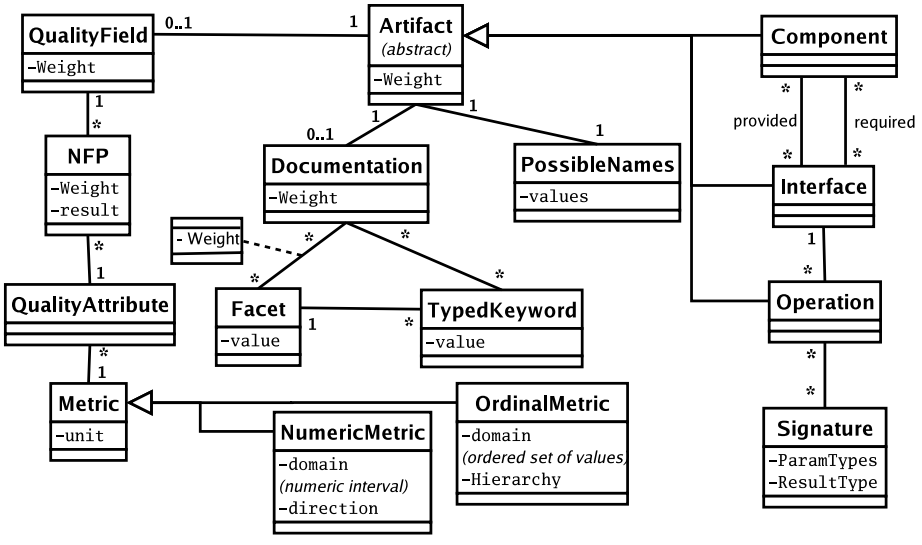


Fig. 1. Description format for COTS components

denoted $ParamTypes = (\tau_1, ..., \tau_n)$, and the result's type denoted *ResultType*. Let us take the example of an operation dedicated to folder creation. It could have a signature $string \rightarrow void$, like the *MakeDirectory* operation provided by *PowerTCP* FTP component, which can be found on the *ComponentSource* website. However, another signature for a folder creation operation could be $string \rightarrow boolean$, like for the *CreateDirectory* operation provided by the *FTPWizard* component, which can be found at the same place.

Information associated to artifacts. Two sets of information are common to all artifacts: a set of its possible names, and a documentation. The first set is here because a same artifact can be proposed under several different names. For example, a download operation can be named *Download* or *GetFile*. This is the case, respectively, for *ComponentSpace*'s FTP component and *Xceed*'s, both being available on *ComponentSource*'s website. The second set of information represents the artefact's documentation. Each one of the information elements included in this documentation is called "typed keywords". A keyword is typed because it positions its value in a specific interpretation domain called facet. For example, a component developed in EJB whose publisher is NBOS Inc. can be documented with two typed keywords: i) one that will have "Publisher" as facet, and "NBOS Inc." as value; ii) one that will have "Technology" as facet, and "EJB" as value.

It is possible to associate other information to artifacts, in particular behavioral information such as pre- and post-conditions. However, the primary goal of our approach is to bring a concrete answer to an industrial concern. To do so, consider the context of COTS component markets such as *ComponentSource*.

Unfortunately, in such markets, the documentation of components' behavior is very poor. This is why we currently do not address these aspects.

3.2 Non-functional Properties Associated to Artifacts

Each artifact can have a "quality field", i.e. a set of non-functional properties (NFP in figure 1). The idea that every architectural artifact can have non-functional properties is inspired by quality of service description languages such as QML [24] and QoSCL [26]. A non-functional property represents the result ($result_P$) obtained by measuring the "level" of a quality attribute on an artifact. This measure is made by a metric. Such a structure is inspired by quality models dedicated to COTS components [27,29]. Such models extend ISO-9126 quality standard [19] by associating quality attributes and metrics to its characteristics and sub-characteristics. We chose metrics to represent and compare non-functional properites, because contrary to other methods focusing on one specific property or family of properties [26], metrics seem to be the simplest evaluation tool for quality in the largest sense.

There are several standards for metrics, such as IEEE 1061-1998 [30], for which a same quality characteristic or sub-characteristic can be measured by several metrics, and conversely. However, there is a problem when a same quality attribute is measured by metrics of a different kind from one quality model to another. Let us take the example of two different quality models: Bertoa's and Vallecillo's model [29] and CQM [27]. Sometimes, both models associate the same quality attributes to ISO-9126's sub-characteristics, but measure them with different metrics. For instance, the *Controllability* attribute, associated to sub-characteristic *Security*, is measured by a percent value in Bertoa's and Vallecillo's model, whereas it is measured by a boolean in CQM. But even though metrics measuring the same attribute may have the same type, it does not mean they are semantically comparable. And there are no systematic methods allowing one to compare values obtained for a same quality attribute with different kinds of metrics. Consequently, we will consider for our description format that one quality attribute can be measured by only one metric, even though a same metric can measure several quality attributes. We can use attributes and metrics from one existing quality model. It can either be an academic one, or one provided by a component market such as *ComponentSource*. In any case, it must be the same for all components.

A metric can be numeric or ordinal. This distinction is inspired by the CLARIFI project [31]. The domain of a numeric metric is a subset of real numbers (integers, percent values...). As the quality models we surveyed do not propose metrics with negative values, we take as a hypothesis that the domain of a numeric metric is always positive. About the domain of an ordinal metric, it is a finite and totally ordered set. A numeric metric has a supplementary attribute called "direction". This direction allows for the interpretation of a metric's result. Available directions are *increasing* and *decreasing*. This distinction is inspired by quality of service contract languages [24,26]. An *increasing* (resp. *decreasing*) direction means that the higher (resp. the lower) the metric's value, the better the corresponding quality. For example, an operation's execution time has a

decreasing direction. An ordinal metric has one supplement attribute called "hierarchy". It gathers and ranks all the metric's possible values by associating a key to each one of them. This key, or rank, defines the total order relation on the metric's domain. When a value is "better" than another, the first one's rank is strictly superior than the second one's rank in the associated hierarchy. For example, if an ordinal metric M has {*very bad*, *bad*, *average*, *good*, *excellent*} as a domain, corresponding hierarchy is: $Hierarchy(M)$=[(0, *very bad*), (1, *bad*), (2, *average*), (3, *good*), (4, *excellent*)].

3.3 Satisfaction Index between Components

Using the same description format given above for candidate and target components, we can address the problem of component comparison. In selection processes and multi-criteria decision making techniques, after total score calculations are performed, the candidate with the highest total score is selected as the best one. This is why we chose to define a satisfaction index based on the same principle. This index allows one to determine how much a candidate component fits the target one. That means, how many functional and non-functional properties this candidate has in common with the target component. First, we will present the principle and the general formula for this satisfaction index, before giving the details for some elements of the description format.

General definition. A careful analysis of description format allows us to distinguish a hierarchical description. In this format, a component is described by a tree whose root is a component artifact and child nodes are potentially: $Interface$, $Documentation$, $PossibleNames$ and $QualityField$. Among them, an $Interface$ node can have the following child nodes: $Operation$, $Documentation$, $PossibleNames$ and $QualityField$. Therefore, the satisfaction index must compare recursively two nodes from different trees by comparing their respective child nodes pair by pair, then "aggregate" the result of sub-nodes' comparisons to measure similarity score. This calculus is the same whatever the nature of the compared nodes is (as long as they are both of the same nature). Therefore, we will give a generic description of this calculus independently of their nature.

To each node, we associate a type and a weighting function. For example, a node can have $Interface$, $Operation$ or $Documentation$ as a type. Only two nodes of a same type can be compared, otherwise the satisfaction index between them will return 0. On the opposite, the maximum value for a satisfaction index is fixed to 1, which means the candidate element completely fits the target one. Weighting function $Weight(E)$ allows the designer to associate to each node E a numeric value called "weight", which gives its importance compared to other nodes. General satisfaction index calculus for a target node E0 and a comparable candidate node E1 is described in figure 2. For each node e0, child of E0, we measure satisfaction indices with each child node of E1 that is comparable to e0 (respectively, index1, index2 and index3). The best result is the highest satisfaction index among them. This measurement is repeated for all E0's other child nodes. Finally, all these best indices, with their corresponding weight, are added

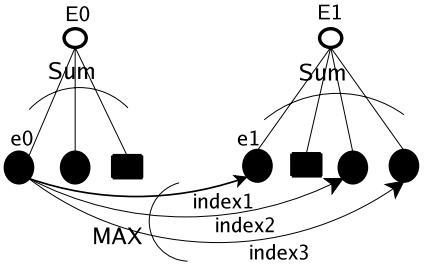


Fig. 2. Satisfaction index between two elements

to obtain a total satisfaction index between E1 and E0. In order to compare leaf nodes, we use a specific function to each kind of them.

Formally, the satisfaction index between a candidate element E_1 and a target one E_0, denoted $Index$, is defined as follows:

$$Index(E_1, E_0) = \begin{cases} \bullet\ 0 \text{ if } Type(E_1) \neq Type(E_0). \\ \bullet\ Comp(E_1, E_0) \text{ if } E_0 \text{ is a leaf node.} \\ \bullet\ \Sigma(\{Weight(e_0) * MAX(\{Index(e_1, e_0) \mid e_1 \in E_1\}) \mid e_0 \in E_0\}) \\ \text{if } E_0 \text{is an inner node.} \end{cases} \tag{1}$$

Selection is performed by calculating satisfaction indices between each available candidate component and the target one, then choosing the candidate whose satisfaction index is the highest one.

Chosen weighting and comparison functions. Now that the general satisfaction index formula has been defined, we have to detail comparison functions we have chosen for each type of leaf node (NFPs, sets of possible names, typed keywords and operation signatures), as well as the weighting function we have chosen for every type of node.

Comparison function between NFPs: Let A_0 be a target artifact, P_0 be an NFP belonging to A_0's quality field, A_1 be a candidate artifact having the same type as A_0, and P_1 be an NFP belonging to A_1's quality field. P_1 is comparable to P_0 only if they measure the same quality attribute. In this case, the metric they both use will be denoted M. If M is numeric, comparison function will measure the similarity of P_1's result value with P_0's, with respect to M's direction. If M is ordinal, the comparison function will measure the similarity of P_1's result's rank with P_0's result's rank, with respect to M's hierarchy.

Formally, the comparison function between P_1 and P_0 is defined as follows:

$$Comp(P_1, P_0) = \begin{cases} \bullet\ 0 \text{ if } P_1 \text{ and } P_0 \text{ do not measure the same quality attribute.} \\ \bullet\ Comp_{inc}(P_1, P_0) \text{ if } M \text{ is numeric with an } \textit{increasing} \text{ direction.} \\ \bullet\ Comp_{dec}(P_1, P_0) \text{ if } M \text{ is numeric with a } \textit{decreasing} \text{ direction.} \\ \bullet\ Comp_{ord}(P_1, P_0) \text{ if } M \text{ is ordinal, and if } rank(result_{P_0}) > 0. \\ \bullet\ 1 \text{ if } M \text{ is ordinal, and if } rank(result_{P_0}) = 0. \end{cases} \tag{2}$$

With:

$$Comp_{inc}(P_1, P_0) = MIN(\frac{result_{P_1}}{result_{P_0}}, 1) \quad (3)$$

$$Comp_{dec}(P_1, P_0) = MIN(\frac{result_{P_0}}{result_{P_1}}, 1) \quad (4)$$

$$Comp_{ord}(P_1, P_0) = MIN(\frac{rank(result_{P_1})}{rank(result_{P_0})}, 1) \quad (5)$$

Let us suppose that P_0 and P_1 both measure the "quality level" of a particular attribute with an ordinal metric whose domain is {*very bad*, *bad*, *average*, *good*, *excellent*}. If P_0's result value is *good*, its rank is 3. If P_1's result value equals *bad*, its rank equals 1 and the comparison function between P_1 and P_0 gives 1/3.

Comparison function between sets of possible names: For simplicity reasons, we consider that for one candidate set of possible names N_1 and one target set possible names N_0, $Comp(N_1, N_0)$ equals 1 if one of N_1's names is contained in N_0 (i.e. there is at least one common element between N_1's values and N_0's), regardless of differences between uppercases and lowercases. In any other case, $Comp(N_1, N_0)$=0.

Comparison function between typed keywords: For a candidate typed keyword K_1 associated to a facet F_1 and a target typed keyword K_0 associated to a facet F_0, $Comp(K_1, K_0)$=1 if F_1's value equals F_0's (unless F_0's value is "", in this case facets are not compared) and if K_1's value equals K_0's. In any other case, $Comp(K_1, K_0)$=0.

Comparison function between operation signatures: To automate comparison between operation signatures, we chose to use signature subtyping, in particular contravariance and covariance rules [23]. Therefore, we consider that a candidate signature S_1=$ParamTypes_1 \rightarrow ResultType_1$ is subtype of a target signature S_0=$ParamTypes_0 \rightarrow ResultType_0$ if $ParamTypes_0$ is subtype of $ParamTypes_1$ and if $ResultType_1$ is subtype of $ResultType_0$. Consequently, $Comp(S_1, S_0)$=1 if and only if S_1 is subtype of S_0. Otherwise, $Comp(S_1, S_0)$=0.

Let us consider, for example, target signature S_0: $float \rightarrow int$. If $S_1 = float \rightarrow float$, it will be a subtype of S_0 and the comparison function will give 1. However, if $S1$: $boolean \rightarrow int$, the comparison function will give 0.

There are other existing techniques to compare operation signatures, in particular signature matching as defined by A. Zaremski and J. Wing [32] or S. Sadou *et al.* [33,34]. However, all matching rules are not fully automatable, because they require a collaborative approach.

Weighting function: For every type of node, we will use "weighting by distribution". It consists, for the application designer, in giving a percent weight and sharing the totality of each node's weight between its direct child nodes, from the root (the component whose weight is 1) to the leaves. For example, an interface will share its weight between its set of possible names, its documentation, its operations and its quality field, so that the sum of all these nodes' weights will equal 100% of the interface's weight. The only exception is the set of possible

signatures for an operation. As we look for only one correct signature among all the ones we propose, all of them will count for one. Let us suppose we described a target operation with three possible signatures and a quality field. If the quality field takes 40% of the interface's weight, then each signature will have a weight equal to 60% of the interface's weight.

4 Selection in *ComponentSource*

In this section, we present an experiment conducted on a concrete component market. This experiment shows practical feasability and interest of an automatic, multi-level, selection approach. We consider the following context: a designer needs for her/his application a component dedicated to FTP (*File Transfer Protocol*) among all the candidates available in the *ComponentSource* component market's "internet communication" section[1]. For each candidate we produced a description in our format. In particular, the quality model we used corresponds to the non-functional information available on *ComponentSource* web pages for each of its components. Then, we tested the selection mechanism on these translated descriptions, and produced the results presented in this section. Translation from original components' descriptions to our description format has been done using model transformation techniques. Because of article size, this work will not be presented in this paper, but it will be the subject of a future publication.

4.1 Non-negotiable Requirements

Let us consider the development of a component-based application. Some constraints are specific to the development context. They are often imposed and non-negotiable. For example, if the chosen development tool is *Visual Studio*, then the candidate components must have *Visual Studio* as a compatible container[2]. Therefore, we must filter candidates to keep only the ones that fit these technological constraints. We can model this filter with our framework. To do so, we can use this constraint on compatible containers as a property whose values are either "True" or "False". Then, we can specify a target component with an NFP corresponding to the compatibility with the *Visual Studio* container. Therefore, with only one constraint whose value is "True" or "False", the possible satisfaction index values are 1 or 0. Only the candidates whose satisfaction index equals 1 will be pre-selected. Thus, only 35 "compatible" candidates from *ComponentSource* remain. In the following pages, measurements will be performed only on these compatible candidates.

4.2 Initial Requirements

The application being in development, its current "concrete" architecture has requirements. That means, the designer needs to find a FTP component that

[1] For more information: http://www.componentsource.com/index.html

[2] There may be other non-negotiable requirements and filters, but for reasons of simplicity and clarity, we will use only this one as an example.

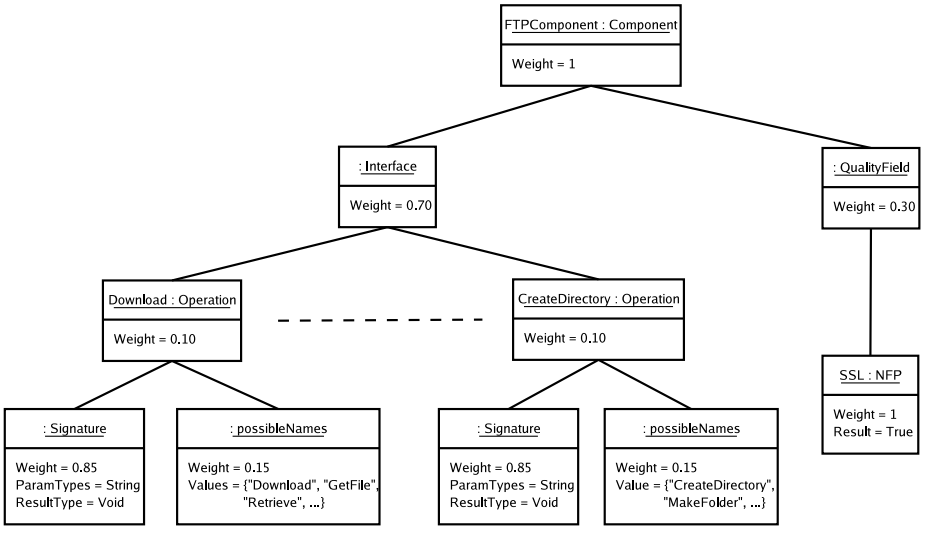


Fig. 3. Example of target component

provides operations whose signatures are required by "concrete" components already included in the architecture. Moreover, the application has security requirements, which imposes that the FTP component enables SSL protocol. It corresponds to quality attribute $SecuritySSL$, measured by an ordinal metric with a boolean domain (see figure 1). The target component that models all these requirements is shown in figure 3. From a functional point of view, it provides one interface containing 10 operations. Each one of them is dedicated to a specific FTP task (folder creation, login, download...), and has a signature imposed by the application's concrete architecture. However, it can have many possible names. From a non-functional point of view, the target component's quality field contains a unique NFP concerning the need of SSL protocol. This NFP represents quality attribute $SecuritySSL$ with value $True$. We estimate that the provided interface takes 70% of the component's weight, while the quality field takes the remaining 30%. Therefore, provided interface and quality field weights equal respectively 0.7 and 0.3 (see figure 3). We also consider that all 10 operations are of equal importance, so each one's weight equals 0.1 in the context of the provided interface. Of course, this is only an example of possible weighting: other weights can be estimated according to the context of the application.

4.3 Results for Initial Requirements

Satisfaction indices have been measured for each of the 35 candidates on our tool *Substitute*[3]. This tool takes as parameters XML files describing the chosen

[3] Because of article size, we cannot give details of *Substitute* tool. However, interested readers can download it at the following address: http://www-valoria.univ-ubs.fr/SE/Substitute/

Table 1. First satisfaction index measurements

Candidate name	Operations	NFP SSL	**Total**
IP*Works! SSL v6 .NET	0.73	1.0	**0.81**
IP*Works! SSL v6 ActiveX/VB	0.73	1.0	**0.81**
IP*Works! SSL v6 .NET Compact Framework	0.73	1.0	**0.81**
IP*Works! SSL v6 ASP/.NET	0.73	1.0	**0.81**
PowerTCP SSL for ActiveX	0.65	1.0	**0.75**
IP*Works! v6 .NET	0.73	0.0	**0.51**
IP*Works! v6 ActiveX/VB	0.73	0.0	**0.51**
IP*Works! v6 .NET Compact Framework	0.73	0.0	**0.51**
IP*Works! v6 ASP/.NET	0.73	0.0	**0.51**
PowerTCP FTP for ActiveX	0.65	0.0	**0.45**
Aspose Network .NET	0.65	0.0	**0.45**
IP*Works! SSL v6 C++	0.14	1.0	**0.39**
SocketTools Secure Visual Edition	0.12	1.0	**0.38**
SocketTools Secure .NET Edition	0.12	1.0	**0.38**
Xceed FTP Library	0.52	0.0	**0.36**

quality model and the set of all the component descriptions that will be used (the target component, and candidate ones). For target components, only needed properties are specified and weighted. Properties that are not specified take implicitly a null weight in our calculus. Once all XML component descriptions are loaded, *Substitute* returns satisfaction indices between each candidate of the library and the target component. Not only global indices, but also local ones for child nodes (interfaces, operations, NFPs...).

Table 1 shows the measurements for the 15 best candidates. First, there are 5 secure (SSL) and 4 non-secure (without SSL) versions of a FTP component provided by the *IP*Works!* component suite. These different versions are identified according to their language (C++), their framework (.NET, ActiveX) or the context they were developed for. For example, the .NET Compact Framework is made specifically for mobile phones, while the ASP/.NET version is better suited for Web applications. Then, there are the secure and non-secure ActiveX versions of a FTP component provided by *PowerTCP* component suite. There are also other FTP components provided by component suites *Aspose Network*, *SocketTools* and *Xceed*. At first, we ignored the candidates' development context. However, the first four candidates all have the same satisfaction index. The reason is that they all represent the same secure FTP component, but for different development contexts (ActiveX, .NEt...). Thus, they provide the same operations with the same signature. Then, we must consider a precise context, such as the development of a client/server application based on ActiveX. In order to choose between the remaining candidates, a more in-depth analysis of them is necessary.

Table 2. New satisfaction index measurements

Candidate name	Operations	NFP SSL	NFP TD	**Total**
IP*Works! SSL v6 ActiveX/VB	0.73	1.0	1.0	**0.81**
PowerTCP SSL for ActiveX	0.65	1.0	1.0	**0.75**
IP*Works! v6 ActiveX/VB	0.73	0.0	1.0	**0.61**
PowerTCP for ActiveX	0.65	0.0	1.0	**0.55**
Xceed FTP Library	0.52	0.0	0.88	**0.45**

4.4 Requirement Evolution

By focusing on a precise development context, thus removing the versions of a same component made for a different context, further exploration becomes conceivable on a remaining candidate. Thus, we can notice they have some properties which were not considered first, but may be very interesting. A good example of such unexpected properties is the tests performed on these components before they were brought to the market. As the application has security requirements, it would be better if the candidates were tested before being integrated. Indeed, *ComponentSource* provides for each component some information about the tests performed on it: installation test, uninstall test, antivirus scan, sample code review, etc... Therefore, it would be interesting to check the "test degree" of each candidate, i.e. the number of tests, among the eight ones recognized by *ComponentSource*, which were performed on it.

This new requirement leads to a modification of the target component. Its quality field now contains a new NFP representing test degree and asking for the maximal value, 8. Provided interface and quality field weight do not change. However, inside the quality field, the weight of NFP representing SSL enabling must decrease a bit. It will equal 0.665 (two thirds of the quality field's weight), while the new NFP representing test degree will take the remaining 0.335.

4.5 Results for New Requirements

Satisfaction indices for the remaining candidates (i.e. those that are developed for ActiveX) have been calculated with *Substitute*. Table 2 shows the new measurements of satisfaction indices for the five best candidates. The new column, *NFP TD*, shows the results for NFP representing test degree. All the tests recognized by *ComponentSource* were performed on the *IP*Works!* and *PowerTCP* components in their secure and non-secure versions (index for NFP TD equals 1). The last candidate did not pass some of these tests, which decrease its satisfaction index. Finally, it seems obvious that, for the specified requirements, secure ActiveX/VB version of *IP*Works!* FTP component is the best candidate.

Usually, when we try to select the "best" candidate for an application's current requirements, we are limited by our initial knowledge. With a way to navigate through the library, it is possible to discover properties offered by the components which we did not originally think about. This is typically one of the trade-offs

predicted by L. Brownsword *et al.* [5] between requirement specification and COTS selection. Our easy-to-use way to specify requirements and select good candidates makes this navigation possible.

5 Conclusion and Future Work

We proposed an approach that allows us to automate the component evaluation phase, including: i) a description format for COTS components' functional and non-functional properties; ii) a satisfaction index that measures the similarity level between a candidate component and a target one. This approach has been validated on *ComponentSource* component market with the help of a tool that measures local and global satisfaction indices for a whole library of candidate components. This study showed that an automated comparison improves the performance of selection process. It also showed the importance of weighting.

Such an automated mechanism is adapted to an incremental construction of a component-based software, because "back-tracking" is possible. Each target component's specification depends on the components already integrated into the application. So if the situation is blocking (i.e. there is no candidate that can satisfy current target component), we can go back to previous ones and choose other candidates for them. It will lead to modified requirements, which may be better satisfied by candidates.

This paper follows and improves a previous work [35,36], whose goal was to find how a component could substitute another one. At that time, we considered no particular context. Since then, we adapted and improved our framework by considering an industrial problem, such as selection in COTS markets. Therefore, the work we present in this paper is better suited to a concrete component-based development context. Considering this context, our description format is inspired by what we do (and do not) find in documentation provided by COTS publishers. For this reason, we have not yet dealt with some component properties, particularly behavioral ones. Because of the importance of these aspects, we plan to take them into account in future versions of our framework. However, in order to achieve this goal, these properties should be documented more explicitly in component markets.

References

1. Voas, J.: COTS software - the economical choice? IEEE Software 15 (3), 16–19 (1998)
2. Ye, F., Kelly, T.: COTS product selection for safety-critical systems. In: Proc. of 3rd Int. Conf. on COTS-Based Soft. Systems (ICCBSS), pp. 53–62 (2004)
3. Crnkovic, I., Larsson, S., Chaudron, M.: Component-based development process and component lifecycle. In: 27th International Conference on Information Technology Interfaces (ITI), Cavtat, Croatia. IEEE, Los Alamitos (2005)
4. Tran, V., Liu, D.B.: A procurement-centric model for engineering CBSE. In: Proc. of the 5th IEEE Int. Symp. on Assessment of Soft. Tools (SAST) (June 1997)

5. Brownsword, L., Obendorf, P., Sledge, C.: Developing new processes for COTS-based systems. IEEE Software 34 (4), 48–55 (2000)
6. En, C.G., Baraçli, H.: A brief literature review of enterprise software evaluation and selection methodologices: A comparison in the context of decision-making methods. In: Proc. of the 5th Int. Symp. on Intelligent Manufacturing Systems (May 2006)
7. Maiden, N., Ncube, C.: Acquiring cots software selection requirements. IEEE Transactions on Software Engineering 24 (3), 46–56 (1998)
8. ComponentSource: Website (2005), http://www.componentsource.com
9. Mosley, V.: How to assess tools efficiently and quantitatively. IEEE Software 8 (5), 29–32 (1992)
10. Saaty, T.: How to make a decision: The analytic hierarchy process. European Journal of Operational Research 48, 9–26 (1990)
11. Kontio, J.: A case study in applying a systematic method for COTS selection. In: Proceedings of International Conference on Software Engineering (ICSE) (1996)
12. Cortellessa, V., Crnkovic, I., Marinelli, F., Potena, P.: Driving the selection of COTS components on the basis of system requirements. In: Proceedings of ACM Symposium on Automated Software Engineering (ASE) (November 2007)
13. Comella-Dorda, S., Dean, J., Morris, E., Oberndorf, T.: A process for COTS software product evaluation. In: Proc. of 1st Int. Conf. on COTS-Based Soft. Systems (ICCBSS), Orlando, Florida, USA, pp. 46–56 (2002)
14. Carvallo, J.P., Franch, X., Quer, C.: Determining criteria for selecting software components: Lessons learned. IEEE Software 24 (3), 84–94 (2007)
15. Kunda, D., Brooks, L.: Applying social-technical approach for COTS selection. In: UK Academy for Information Systems Conf. (UKAIS 1999) (April 1999)
16. Lozano-Tello, A., Gómez-Pérez, A.: Baremo: How to choose the appropriate software component using the analytic hierarchy process. In: Proc. of Int. Conf. on Soft. Eng. and Knowledge Eng (SEKE), Ischia, Italy (July 2002)
17. Martinez, M., Toval, A.: COTSRE: A components selection method based on requirements engineering. In: Proceedings of the 7th Int. Conf. on COTS-Based Soft. Systems (ICCBSS), February 2008, pp. 220–223 (2008)
18. Ochs, M., Pfahl, D., Chrobok-Diening, G., Nothelfer-Kolb, B.: A COTS acquisition process: Definition and application experience. In: Proceedings of the 11th European Software Control and Metrics Conference (ESCOM), pp. 335–343 (2000)
19. ISO International Standards Organisation Geneva, Switzerland: ISO/IEC 9126-1:2001 Software Engineering - Product Quality - Part I: Quality model (2001)
20. Ncube, C., Dean, J.: The limitations of current decision-making techniques in the procurement of COTS software component. In: Proc. of the 1st Int. Conf. on COTS-Based Software Systems (ICCBSS), Orlando, Florida, USA, pp. 176–187 (2002)
21. Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Transactions On Software Engineering 21(6), 528–562 (1995)
22. Príeto-Diaz, R.: Implementing faceted classification for software reuse. Communications of the ACM 34(5), 88–97 (1991)
23. Cardelli, L.: A semantics of multiple inheritance. Information and Computation 76(2), 138–164 (1988)
24. Frolund, S., Koistinen, J.: QML: A language for quality of service specification. Technical report, Hewlett-Packard Laboratories, Palo Alto, California, USA (1998)
25. Beugnard, A., Sadou, S., Jul, E., Fiege, L., Filman, R.: Concrete communication abstractions for distributed systems. In: Object-Oriented Technology, ECOOP 2003 Workshop Reader, Darmstadt, Germany, November 2003, pp. 17–29 (2003)

26. Defour, O., Jézéquel, J.M., Plouzeau, N.: Extra-functional contract support in components. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 217–232. Springer, Heidelberg (2004)
27. Alvaro, A., de Almeida, E.S., Meira, S.: A software component quality model: A preliminary evaluation. In: Proc. of the 32nd EUROMICRO Conf. on Soft. Eng. and Advanced Applications (SEAA) (August 2006)
28. OMG: UML 2.0 superstructure final adopted specification, document ptc/03-08-02 (August 2003), http://www.omg.org/docs/ptc/03-08-02.pdf
29. Bertoa, M., Vallecillo, A.: Quality attributes for COTS components. I+D Computación 1(2), 128–144 (2002)
30. IEEE: IEEE Std. 1061-1998: IEEE Standard for a Software Quality Metrics Methodology. IEEE computer society press edn (1998)
31. Boegh, J.: Certifying software component attributes. IEEE Software 40(5), 74–81 (2006)
32. Zaremski, A., Wing, J.: Signature matching: a tool for using software libraries. ACM Trans. On Soft. Eng. and Methodology (TOSEM) 4(2), 146–170 (1995)
33. Sadou, S., Mili, H.: Unanticipated evolution for distributed applications. In: 1st Int. Workshop on Unanticipated Software Evolution (USE) (June 2002)
34. Sadou, S., Koscielny, G., Mili, H.: Abstracting services in a heterogeneous environment. In: IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001, Heidelberg, Allemagne (November 2001)
35. George, B., Fleurquin, R., Sadou, S.: A component-oriented substitution model. In: Proceedings of 9th Int. Conf. on Software Reuse (ICSR 9) (June 2006)
36. George, B., Fleurquin, R., Sadou, S.: A methodological approach for selecting components in development and evolution process. Electronic Notes on Theoretical Computer Science (ENTCS) 6(2), 111–140 (2007)

Opportunistic Reuse: Lessons from Scrapheap Software Development

Gerald Kotonya, Simon Lock, and John Mariani

Computing Dept., InfoLab 21, South Drive
Lancaster University, Lancaster LA1 4WA,
United Kingdom
{gerald,lock,jam}@comp.lancs.ac.uk

Abstract. Many organizations use opportunistic reuse as a low-cost mechanism to improve the efficiency of development. Scrapheap reuse is a particular form of opportunistic development that we explore in this paper with the aid of an experimental study.

1 Introduction

Opportunistic reuse is the most common of all software reuse strategies. It does not rely on specific technology; moreover it can be done without a formal content management system. A study conducted by Sen in 1997 showed that software developers seldom choose the predetermined reuse path; instead, they select reuse tasks opportunistically [1]. Like systematic reuse, successful opportunistic reuse relies on the availability of reusable software artefacts (requirements, designs, code, test cases etc.) [2]. However, unlike systematic reuse, for which reusable artefacts are well-defined and clearly located, opportunistic reuse requires that the developer not only be aware of the opportunities for reuse, but also search for and retrieve reusable artefacts.

Even if developers are able to identify opportunities for reusing assets, they may still reuse the artefacts inappropriately as there are often no guidelines or safeguards. To explore these issues further, we devised a study called "Scrapheap Software Challenge" to investigate how different teams of developers apply opportunistic reuse in different application contexts. In particular, our aim is to try to reveal the factors that influence the quality, speed and productivity of this form of development.

In our study, teams of experts drawn from different application domains were asked to build a number of pre-specified software systems by scavenging and "borrowing" functionality from discarded and functioning software systems. The systems produced were composed mostly of reused scrap software, with limited new code being crafted to plug significant gaps in the architecture or to simply act as glue between other components. The study aimed to explore the following three key questions: When should we use a scrapheap development approach? What factors determine the success of scrapheap development? What kinds of systems are produced by scrapheap development?

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 302–309, 2008.

2 Background

Studies have shown that reuse has improved software quality and productivity over the past 40 years. However, the widespread uptake of reuse in software development has not materialised [4, 5]. Several reasons have been advanced to explain the slow uptake, including difficulty in estimating the impact of reuse, inadequate tool support to facilitate reuse, limited understanding of how developers reuse software artefacts, and how the nature of application and support for reuse influences the outcome [5]. However, although various strategies have been proposed to improve software reuse within organisations [6, 7], many of the strategies proposed focus on long-term rather than short-term benefits of software reuse, which would preclude opportunistic reuse.

We believe that a pragmatic approach to reuse has a beneficial role in software development, and that short-term successes derived from opportunistic reuse can be used to seed systematic reuse programmes in organizations. An industry study by Henry and Faller [8] showed how pragmatic opportunistic reuse can result in far-reaching success. They report the results of two large industry projects, in which reuse across projects and the organization improved time-to-market, productivity and software quality.

Scrapheap reuse is a unique form of system development in which whole applications or large parts thereof are composed from "scraps" of software functionality retrieved from discarded systems or cancelled projects. These scrap components are often unwanted and have been discarded – thrown onto the scrapheap; they are free (or at least cheap) to those who wish to make use of them; they still retain some implicit value, due to the original investment in their development and potential residual utility; they may be broken and no longer function as originally intended in their current operational context; they may be incomplete, with key parts missing; they may be outdated and unable to fulfil the non-functional requirements of their current operational context (e.g. compatibility, performance, usability, aesthetics etc.); they may have been created with no intention that they be reused.

A key question is why we would wish to make use of such components in the development of new systems. The main reason for this is that scrap components, although discarded, often represent a considerable investment in terms of time and development effort. If we can reuse these components, then we have the potential to unlock the original investment and, in doing so, provide a rapid, low-cost means to develop new products. It is no coincidence that development projects that make use of scrap are typically those with tight deadlines and limited resources available to them.

2.1 Sources of Scrap – "The Scrapheap"

A distinguishing characteristic of scrap components is that they are not actively maintained or part of currently viable systems. Such components may either be still in the development phase (partially complete, possibly untested, potentially from cancelled projects) or in the retirement phase (from legacy or redundant systems, outdated and superseded components, potentially with broken or "worn" parts). Scrapheap reuse does not however encompass currently working and potentially evolving components from operational systems – this is the realm of traditional reuse which is not the focus of this paper.

There are a wide variety of reasons why a software project can be cancelled, or individual components discarded before deployment. These can include the shifting of requirements that invalidate developed system components; budget and/or time overrun leading to cancellation of project before completion; unforeseen technical problems which could not be solved with the available resources. Similarly, there are a wide variety of reasons why a system or components of a system will be retired from use. These can include the upgrade of a system or parts of the system with newer versions; the rejection of the entire system by the end users or organisational policy makers; incompatibility with new platforms, applications and processes.

3 The Study: The "Scrapheap Software Challenge"

To make this study interesting and compelling for the participants, we designed it to take the form of a competition. The *Scrapheap Software Challenge* tasked competitors with the objective of building a system from scrap software components within a constrained timeframe, to achieve a particular high-level functional goal. The development teams were drawn from academic staff and students in different research groups in our computing department. It was essential that team members knew each other well before the competition began due to the very short deadline for the challenges.

In our study, there were three teams of four developers who competed in three separate challenges held on 3 different days (with a week in between each challenge to allow teams to recover). On each day, the objective of the challenge was revealed at 8 am and the teams had until 5:30 pm to build a system from scrap components that would achieve that objective. At 5:30 pm, all teams were brought back together and a demonstration of the systems and judging took place.

Scores were awarded by a panel of judges based on criteria that included: functional and non-functional properties, usability, scalability, novelty, creativity and aesthetics of their products. The team with the highest overall score won the challenge, and the team with the most wins at the end of the challenge were declared champions. At the end of each challenge the judges individually awarded the teams points for a range of criteria that were laid out in the challenge descriptions. The team with the most points overall won that particular challenge.

Each of the three teams represented one of the major research areas of our department, and each of the three challenges was also drawn from those areas. The members of the teams were all experienced researchers and developers, ranging from senior lecturers through research assistants to post-graduate students.

This *Scrapheap Software Challenge* case study was selected for a number of good reasons. The limited duration of the study made it practical to stage as well as cost effective in terms of time and resources. The small and fixed location made it easy to document and observe the activities of the teams and the small scale of the development teams and final products made them very convenient for analysis.

What is of importance is that, in spite of the very limited timeframe, the teams involved were still able to produce relatively complete final systems. As a direct consequence of the time limitation, teams were forced to scavenge and reuse as much as possible, with no possibility for large scale redevelopment. This all resulted in what

we have termed a "pressure cooker" development environment - the competitive nature of the challenges and the very tight deadlines fuelled a rich and intense development microcosm which could be easily observed and studied in detail.

3.1 The Challenges

The three challenges that were set for the teams to complete were as follows:

Challenge 1 (Mobile Computing): Audio Graffiti - The teams had to construct a mobile system to facilitate 'audio graffiti'. The system would provide the user with the ability to associate audio (speech or music) with particular locations or regions in geographical space. Users were expected to be able to 'browse' the community airwaves by wandering through physical space. Each team also had to solve the problem of location tracking in both indoor and outdoor locations. The solutions produced were as follows:

(1) An electronic 'spray can' built from a Programmable Intelligent Computer (PIC) for recording audio, combined with a set of audio graffiti tags placed in the environment. Tags sense the proximity of a spray can via infrared and then notify it of the current location. The user can listen to any audio graffiti associated with the tag, and/or add their own audio for others to listen to at a later date.
(2) A vision based, scene recognition system that used a portable webcam to determine the user's current location. This was achieved using a colour spectrum profiling tool in order to distinguish between different scenes. Users could listen to or record audio graffiti associated with a particular location, with a central server being used to store the location and audio data.
(3) A mobile computing solution which made use of distance from WI-FI hotspots (determined by signal strength patterns) for location sensing. Audio data was stored on a number of different media servers, the server used being dependent on its proximity to the graffitied location.

Challenge 2 (Ubiquitous computing): Absent Presence - In this challenge the teams had to create a system that could sense a visitor's presence and take this forward in some way for future presentation. The aim was not to establish contact with past people, but just to give visitors to the space (e.g. museum, web page, monument) a sense that others have been there. The system had to in some way capture an aspect of the previous visitors' behaviour (sound, mouse movement, link history, physical movement). The solutions produced were as follows:

(1) An augmented coffee table which graphically 'remembered' the objects placed on top of it or moved across it. This made use of a camera to record the objects currently on the table, a layered history of previous images of the table, an image addition tool to produce a composite image and a top-down projector to overlay the historical image onto the table.
(2) A weight sensor augmented area of flooring which recorded and preserved people's footprints as they walked across it. The presentation took the form of a grid with the colour of each cell representing the number of footfalls within that area. Presentation was again achieved through the use of a projector.

(3) An augmented sofa that was able to detect the presence of users using a cushion sensor. The system maintained a record of the pattern of people sitting on the sofa over a period of time and represented them on a nearby screen using 'fairy' sprites.

Challenge 3 (Human Computer Interaction): Informated Artefact - The teams had to create a piece of dynamic corporate art for the entrance foyer of the computing department. The artefact should change in some way in reaction to activity in the building. It needed to be both interesting to look at as a work of art, but also embody something of the work done in the building. At least some of the information presented should be obscure, so that it needs someone to explain it, or one can only figure it out by watching carefully for a while. The solutions produced were as follows:

(1) A robotic wizard's hat that physically moved and illuminated to represent activity in the building. Sources of data that fed the actuated hat were noise and motion sensors that could be distributed around key spaces in the building.
(2) A life-sized mannequin that displayed the collective emotional state of all residents in a building. Data on the emotional state of the residents was collected by aggregating the status indicators of users' instant messenger applications. The presentation of emotions was achieved by projecting expressions onto the blank white face of the mannequin.
(3) A 'Jacob's ladder' style sculpture that made use of sparks projected onto a conical structure to represent the activity in the building. The colour of the sparks represented the types of activity that was going on and the number of sparks indicated the volume of activity. Sources of data included the number of documenting being printed, the load on the departmental web proxy and the keystroke rates of users' keyboards.

It was important for all teams to have an equal opportunity to win and the challenges were written in such a way that teams would not be disadvantaged by their particular discipline, experience or background. The varied solutions produced by the teams reflected not only their different backgrounds, but also the differing mix of software and hardware components used. These components included: Programmable Intelligent Computers, webcams, generic image processing tools (e.g. Imagemagick), computing vision systems, cannibalised hardware, image capture software, LCD projectors, everyday household artefacts (e.g. tables), actuators and sensor boards from previous projects, image profiling tools, instant messenger prototypes, key logging software. The teams used a range of different programming languages for gluing the components together, including C, Flash, Java, PHP, shell scripts and DOS batch commands.

4 Observations

Based on our observation made during this study, we are able derive a number of findings and conclusions which can be generalised to a wide range of software development situations.

Component selection was heavily dependent upon developers' knowledge and past experience. The teams tended to have knowledge of components from their own

particular fields (areas of research and past development projects). They rarely ventured outside these fields and instead, as we would expect, made use of familiar components. This resulted in a unique bottom-up "technology driven" development style being used, in addition to the more traditional top-down goal or requirements led approaches.

The limited time available to complete the challenges resulted in the development of very few new components. The teams did however spend a significant amount of time developing the glue to bind components together and persuade them to interoperate. This is because, unlike traditional component-based systems, the components involved in scrapheap development were often never intended to be reused. Due to the time constraints and difficulties in achieving interoperability between components, practicality took priority over good design and resulted in functional, but not particularly well-designed systems. Software components were also converted into rudimentary web services and hardware components wrappered by software and made similarly available. Multiple machines were then used to host these different components, resulting in large and coarse-gained distributed systems.

Early on in the development process, there was a phase of rapid evolution to the chosen design. This was a direct consequence of an initial influx of knowledge about the selected components. During this stage there were many revisions made in order to bring the conceptual solution into line with the reality of available components. The most successful teams in the challenges were those who committed to a particular solution early on - sometimes after only a brief initial discussion. The designs that these groups came up with evolved, sometimes radically, over the duration of the challenge. In the high pressure conditions of the challenge, a pragmatic approach had to be taken.

The systems produced through the scrapheap development process were often unstable and unreliable. Although this may be partly attributed to the very short development times permitted, it may also be a consequence of the fact that they are composed of components which were never intended to work together in the configurations developed. The environment and use of many of the components were not what they were initially created for. As a result of this, they would behave unpredictably and erroneously.

Despite this, the systems produced achieved the high-level objectives that we set for them. In addition, due to the fact that the components which made up a system were developed independently of each other (often for completely different systems) they tended to be very loosely coupled and very highly cohesive. These desirable properties of system components, although produced unintentionally, nevertheless resulted in systems that were very amenable to change and evolution. Components could be unplugged and replaced without affecting other components. The commonly observed "ripple effect" of change was thus minimal as change rarely propagated between components.

5 Conclusions

Scrapheap reuse is not suited to all forms of development. In order to successful employ the approaches discussed in this paper, it is essential to be able to determine

when it is applicable and when it is not. Scrapheap development is most effective when used to develop systems with loose and flexible specifications. Inflexible low-level requirements will significantly limit implementation options. The challenges that worked best had abstract high-level goals that allowed for negotiation and adaptation of the lower-level requirements.

Scrapheap reuse is ideal for the development of prototypes and proof of concept systems. These are applications intended for demonstration purposes or for personal or use internally by an organisation. In all of these aforementioned situations, less than the highest level of quality is acceptable. Such development efforts often have very short timescales with very limited budget available and the final system is likely to have a very short life expectancy. The user base of such systems tends to be very small, with the audience being the members of a design team, a single client, or even just oneself [9].

The level of experience of the development team is also of key importance when considering scrapheap development. The development team must have previous experience of similar projects and preferably first-hand knowledge of relevant development. It is useful to reflect however that our study indicates developers need not have experience in the application domain itself, so long as the experience that they do have is transferable.

During the study it was found that teams who committed to a solution early on were particularly successful. That is not to say they did not change their designs over the course of development. By adopting a design early on, it is possible to rapidly assess its feasibility.

The combined knowledge of the development team is a crucial element in the success of scrapheap development. It is essential that they have extensive knowledge of what scrap is available, where it can be obtained and how it can be rendered of use to the current project. Of equal importance to knowledge relating to the functionality offered by a component, it is essential to be aware of what features and properties are broken or inappropriate.

It is worth noting that due to the nature of the development process and intended purpose of the final products, it is often unrealistic to strive for perfection. Teams who took a pragmatic approach and aimed to produce products that were ‘good enough’ were found to be successful in the study. This may require a shift in attitude of developers who may aim to produce the best achievable solution.

The quality of the final system is not always particularly high. In many ways this is only to be expected - if you build a system from scrap, you can’t expect the end product to be a thing of beauty. The components found on the scrapheap are often not easily integrated with out components, resulting in the need for a significant amount of glue to get them to interoperate. For these reasons, it is no surprise that the systems produced by scrapheap development are often inelegant and inefficient.

One positive benefit of scrapheap systems is that the nature of the development process can help to prevent unrealistic solutions from being attempted. This is due to the fact that we already know what components are possible and practical to implement (as they are already in existence). Surveying the components that are in the scrapheap can thus offer an invaluable reality check for the designers of a system.

There are various features of industrial software development that differ markedly from the situation depicted in our *Scrapheap Software Challenge* study. For example,

project duration, size of development teams, the need to fulfil more specific requirements, a high level of reliability of the final system, skills and ability of the developers and so on. However, there are also many of similarities between our case study and real-world development that make our findings a useful contribution to the research and practice of opportunistic software reuse and scrapheap development. Factors such as severe time pressures, limited available components, incomplete knowledge of component availability and features, uncertainty as to the suitability of initial designs, potential mismatch between system goals and available components are common to both our study and industrial development.

As we have tried to make clear in this paper, scrapheap development is not appropriate for all types of system. However, if the guidance offered in this paper is followed, this form of reuse can be gainfully and directly employed in the production of rapid prototypes, personal applications and proof of concept systems. This makes great things possible for even small groups of developers with very limited resources.

References

1. Sen: The Role of Opportunism in the Software Design Reuse Process. IEEE Transactions on Software Engineering 23(7), 418–436 (1997)
2. Sommerville: Software Engineering. Addison-Wesley, Reading (2006)
3. Rockley, A.: Managing Enterprise Content: A Unified Content Strategy, New Riders (2002)
4. Frakes, W.B., Kang, K.: Software Reuse Research: Status and Future. IEEE Transactions on Software Engineering 31(7), 529–536 (2005)
5. Morisio, M., Ezran, M., Tully, C.: Success and failure factors in software reuse. IEEE Transactions on Software Engineering 28(4), 340–357 (2002)
6. Rothenberger, M.A., Dooley, K.J., Kulkarni, U.R., Nada, N.: Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. IEEE Transactions on Software Engineering 29(9), 825–837 (2003)
7. Ezran, M., Morisio, M., Tully, C.: Practical Software Reuse (Practitioner Series). Springer, Heidelberg (2002)
8. Henry, E., Faller, B.: Large-scale industrial reuse to reduce cost and cycle time. IEEE Software 12(5), 47–53 (1995)
9. Hartmann, B., Doorley, S., Klemmer, S.R.: Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development. Technical Report, Stanford University Computer Science Department (October 2006)

A Component Model for Control-Intensive Distributed Embedded Systems*

Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš,
Jan Carlson, and Ivica Crnković

Mälardalen University, Västerås, Sweden
{severine.sentilles,aneta.vulgarakis,tomas.bures,
jan.carlson,ivica.crnkovic}@mdh.se

Abstract. In this paper we focus on design of a class of distributed embedded systems that primarily perform real-time controlling tasks. We propose a two-layer component model for design and development of such embedded systems with the aim of using component-based development for decreasing the complexity in design and providing a ground for analyzing them and predict their properties, such as resource consumption and timing behavior. The two-layer model is used to efficiently cope with different design paradigms on different abstraction levels. The model is illustrated by an example from the vehicular domain.

1 Introduction

A special class of embedded systems are control-intensive distributed systems which can be found in many products, such as vehicles, automation systems, or distributed wireless networks. In this category of systems as in most embedded systems, resources limitations in terms of memory, bandwidth and energy combined with the existence of dependability and real-time concerns are obviously issues to take into consideration.

Another problem when developing such systems is to deal with the rapidly increasing complexity. For example in the automotive industry, the complexity of the electronic architecture is growing exponentially, directed by the demands on the driver's safety, assistance and comfort [3]. In this class of systems, distribution is also an important aspect. The architecture of the electronic systems is distributed all over the corresponding product (car, production cell, etc.), following its physical architecture, to bring the embedded system closer to the sensed or controlled elements.

In this paper, we propose a new component model called ProCom with the following main objectives: (i) to have an ability of handling the different needs which exist at different granularity levels (provide suitable semantics at different levels of the system design); (ii) to provide coverage of the whole development process; (iii) to provide support to facilitate analysis, verification, validation and

* This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 310–317, 2008.

testing; and (iv) to support the deployment of components and the generation of an optimized and schedulable image of the systems. The focus of this paper is on the component model itself, described as means for designing and modelling system functionality and as a framework that enables integration of different types of models for resource and timing analysis.

The component model is a part of the PROGRESS approach [7] that distinguishes three key activities in the development: design, analysis and deployment. The *design* activity provides the architectural description of the system compliant with the semantic rules of the component model presented in this paper and enables the integration analysis and deployment capabilities. *Analysis* is carried out to ensure that the developed embedded system meets its dependability requirements and constraints in terms of resource limitations. The proposed component model provides means to handle and reuse the different information generated during the analysis activity. The *deployment* activity is specific for control-intensive embedded systems; due to timing requirements and resource constraints, the execution models can be very different from the design models. Typically, execution units are processes and threads of tasks.

The main focus of this paper is oriented towards system design. The two supplementary activities (analysis and deployment) are outside the scope of the paper. A component model that enables a reusable design, takes into consideration the requirements' characteristics for control-intensive embedded systems, and is used as an integration frame for analysis and deployment, is elaborated in the subsequent sections.

The ideas underlying ProCom emanate partly from the previous work on the SaveComp Component Model (SaveCCM) [1] within the SAVE project, such as the emphasis on reusability, a possibility to analyse components for timing behavior and safety properties. Several other concepts and component models have inspired the ProCom Design. Some of them are the Rubus component model [2], Prediction-Enabled Component Technology (PECT) [10], AUTOSAR [3], Koala [9], the Robocop project [8], and BIP [4].

2 The ProCom Two Layer Component Model

In designing our component model, we have aimed at addressing the key concerns which exist in the development of control-intensive distributed embedded systems. We have analyzed these concerns in our previous work [6], with the conclusion that in order to cover the whole development process of the systems, i.e. both the design of a complete system and of the low-level control-based functionalities, two distinct levels of granularity are necessary.

Taking into consideration the difference between those levels, we propose a two-layer component model, called *ProCom*. It distinguishes a component model used for modelling independent distributed components with complex functionality (called *ProSys*) and a component model used for modelling small parts of control functionality (called *ProSave*). ProCom further establishes how a ProSys component may be modelled out of ProSave components. The following subsections

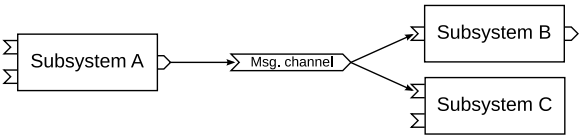


Fig. 1. Three subsystems communicating via a message channel

describe both of the layers and their relation. The complete specification of ProCom is available in [5].

2.1 ProSys — The Upper Layer

In ProSys, a system is modeled as a collection of concurrent, communicating *subsystems*, possibly developed independently. Some of those subsystems, called *composite subsystems*, can in turn be built out of other subsystems, thus making ProSys a hierarchical component model. This hierarchy ends with the so-called *primitive subsystems*, which are either subsystems coming from the ProSave layer or non-decomposable units of implementation (such as COTS or legacy subsystems) with wrappers to enable compositions with other subsystems. From a CBSE perspective, subsystems are the "components" of the ProSys layer, i.e., design or implementation units that can be developed independently, stored in a repository and reused in multiple applications.

The communication between subsystems is based on the asynchronous message passing paradigm which allows transparent communication (both locally or distributed over a bus). A subsystem is specified by typed input and output *message ports*, expressing what type of messages the subsystem receives and sends. The specification also includes attributes and models related to functionality, reliability, timing and resource usage, to be used in analysis and verification throughout the development process. The list of models and attributes used is not fixed and can be extended.

Message ports are connected via *message channels* — explicit design entities representing a piece of information that is of interest to several subsystems — as exemplified in Fig. 1. The message channels make it possible to express that a particular piece of shared data will be required in the system, before any producer or receiver of this data has been defined. Also, information about shared data such as precision, format, etc. can be associated with the message channel instead of with the message port where it is produced or consumed. That way, it can remain in the design even if, for example, the producer is replaced by another subsystem.

2.2 ProSave — The Lower Layer

The ProSave layer serves for the design of single subsystems typically interacting with the system environment by reading sensor data and controlling actuators accordingly. On this level, components provide an abstraction of tasks and control loops found in control systems.

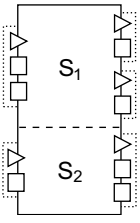


Fig. 2. A ProSave component with two services; S_1 has two output groups and S_2 has a single output group. Triangles and boxes denote trigger- and data ports, respectively.

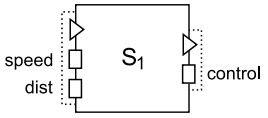
A subsystem is constructed by hierarchically structured and interconnected ProSave *components*. These components are encapsulated and reusable design-time units of functionality, with clearly defined interfaces to the environment. As they are designed mainly to model simple control loops and are usually not distributed, this component model is based on the pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components.

A ProSave component is of a collection of services, each providing a particular functionality. A service consists of an *input port group* containing the activation trigger and the data required to perform the service, and a set of *output port groups* where the data produced by the service will be available. Fig. 2 illustrates these concepts. The data of an output group are produced at the same time, at which the trigger port of that group is also activated. Having multiple output groups allows the service to produce time critical parts of the output early.

ProSave components are *passive*, i.e. they do not contain their own execution threads and cannot initiate activities on their own. So each service remains in a passive state until its input trigger port has been activated. Once activated, the data input ports are read in one atomic operation and the service switches into an active state where it performs internal computations and produces data on its output ports. Before the service returns to the inactive state again, each of its output groups should be written exactly once.

Input data ports can receive data while the service is active, but it would only be available the next time the service is activated. This simplifies analysis by ensuring that once a service has been activated it is functionally (although not temporally) independent from other components executing concurrently.

A component also includes a collection of structured *attributes* which define simple or complex types of component properties such as behavioural models, resource models, certain dependability measures, and documentation. These attributes can be explicitly associated with a specific port, group or service (e.g. the worst case execution time of a service, or the value range of a data port), or related to the component as a whole, for example a specification of the total memory footprint. New attribute types can also be added to the model.




```
typedef struct {
  int *speed;
  float *dist;
} in_S1;
typedef struct {
  int *control;
} out_S1;
void init();
void entry_S1(in_S1 *in, out_S1 *out);
```

Fig. 3. A primitive component and the corresponding header file

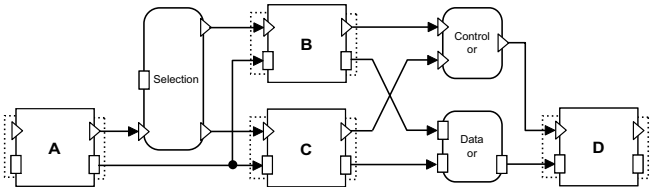


Fig. 4. A typical usage of *selection* and *or* connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

The functionality of a component can either be realized by code (*primitive component*), or by interconnected sub-components (*composite component*). For primitive components, in addition to a function called at system startup to initialise the internal state, each service is implemented as a single non-suspending C function. Fig. 3 shows an example of the header file of a primitive component.

Composite components internally consist of *sub-components*, *connections* and *connectors*. A *connection* is a directed edge which connects two ports (output data port to input data port of compatible types and output trigger port to input trigger port) whereas *connectors* are constructs that provide detailed control over the data- and control-flow. The existence of different types of connectors and the simple structure of components makes it possible to explicitly specify and then analyse the control flow, timing properties and system performance.

The set of connectors in ProSave, selected to support typical collaboration patterns, is extensible and will grow over time as additional data- and control-flow constructs prove to be needed. The initial set includes connectors for *forking* and *joining* data or trigger connections, or *selecting* dynamically a path of the control flow depending on a condition. Fig. 4 shows a typical usage of the selection connector together with *or* connectors.

ProSave follows the push-model for data transfers and the triggered service always uses the latest value written to each input data port. Since communication may eventually be realised over a physical connection, the transfer of data and triggering is not an atomic operation. For triggering and data appearing together at an output group, however, the semantics specify that all data should be delivered to their destinations before the triggering is transferred, to avoid components being triggered before the data arrives.

2.3 Integration of Layers — Combining ProSave and ProSys

ProCom provides a mechanism for integrating the low-level design of a subsystem described by ProSave into the high-level design described by ProSys. A ProSys primitive subsystem can be further specified using ProSave (as exemplified in Fig. 6). Concretely, in addition to ProSave components, connections and ProSave connectors, additional connector types are introduced to *(a)* map the architectural style (message passing used in ProSys to pipes-and-filters used in ProSave, and vice versa), and *(b)* specify periodic activation of ProSave components.

Periodic activation is provided by the clock connector, with a single output trigger port which is repeatedly activated at a given rate. To achieve the mapping from message passing to trigger and data, and vice versa, the message ports of the enclosing primitive subsystem are treated as connectors with one trigger port and one data port when appearing on the ProSave level. An input message port corresponds to a connector with output ports. Whenever a message is received by the message port, it writes the message data to the output data port and activates the output trigger. Oppositely, output message ports correspond to a connector with an input trigger and input data ports. When triggered, the current value of the data port is sent as a message.

These composition mechanisms do not only allow a consistent design of the entire system by integrated pre-existing subsystems but also provide mechanisms for analysis of particular attributes such as timing properties or performance of the entire system using specifications or analysis results of the subsystems.

3 Example

To illustrate the ProCom component model we use as an example an electronic stability control (ESC) system from the vehicular domain. In addition to anti-lock braking (ABS) and traction control (TCS), which aim at preventing the wheels from locking or spinning when braking or accelerating, respectively, the ESC also handles sliding caused by under- or oversteering.

The ESC can be modeled as a ProSys subsystem, as shown in Fig. 5. Inside, we find subsystems for the sensors and actuators that are local to the ESC. There are also subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). In the envisioned scenario, the TCS and ABS subsystems are reused from previous versions of the car, while SCS corresponds to the added functionality for handling under- and oversteering. Finally, the "Combiner" subsystem is responsible for combining the output of the three. The internal structure of a SCS primitive subsystem is modeled in ProSave (see Fig. 6). The SCS contains a single periodic activity performed at a frequency of 50 Hz, expressed by a clock connector. The clock first activates the two components responsible for computing the actual and desired direction, respectively. When both components have finished their respective tasks, the "Slide detection" component compares the results (i.e., the actual and desired directions) and decides whether or not stability control is required. The fourth component computes the actual response, i.e., the adjustment of brakeage and acceleration.

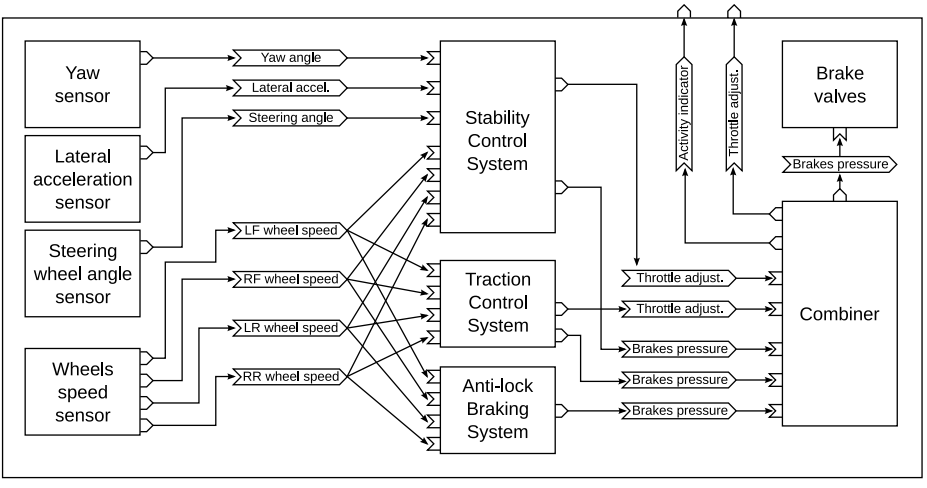


Fig. 5. The ESC is a composite subsystem, internally modelled in ProSys

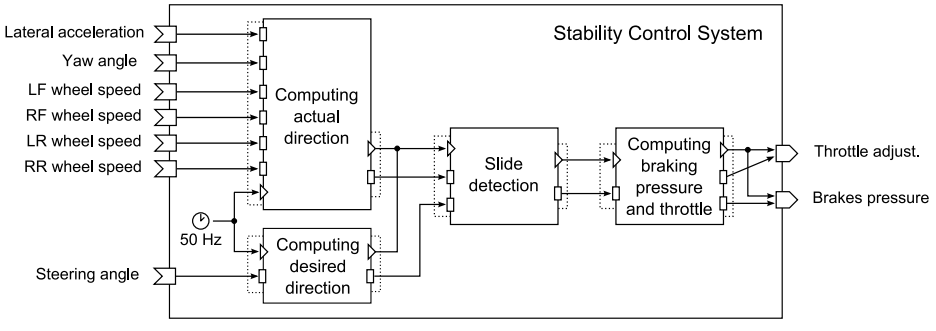


Fig. 6. The SCS subsystem, modelled in ProSave

4 Conclusions

We have presented ProCom, a component model for control-intensive distributed embedded systems. The model takes into account the most important characteristics of these systems and consistently uses the concept of reusable components throughout the development process, from early design to deployment. A characteristic feature of the domain we consider is that the model of a system must be able to provide both a high-level view of loosely coupled subsystems and a low-level view of control loops controlling a particular piece of hardware. To address this, ProCom is structured in two layers (ProSys and ProSave). At the upper layer, ProSys, components correspond to complex active subsystems communicating via asynchronous message passing. The lower layer, ProSave, serves

for modelling of primitive ProSys components. It is based on primitive components implemented by C functions, and explicitly captures the data transfer and control flow between components using a rich set of connectors.

The future work on ProCom includes elaborating on advanced features of the component model (e.g. static configuration, mode shifting, error-handling, etc.), building an integrated development environment and evaluating the proposed approach in real industrial case-studies.

References

1. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: SAVE approach to component-based development of vehicular systems. Journal of Systems and Software 80(5), 655–667 (2007)
2. Arcticus Systems. Rubus Software Components, http://www.arcticus-systems.com
3. AUTOSAR Development Partnership. Technical Overview V2.2.1 (February 2008), http://www.autosar.org
4. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods, pp. 3–12. IEEE, Los Alamitos (2006)
5. Bureš, T., Carlson, J., Crnković, I., Sentilles, S., Vulgarakis, A.: ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University (June 2008)
6. Bureš, T., Carlson, J., Sentilles, S., Vulgarakis, A.: A component model family for vehicular embedded systems. In: The Third International Conference on Software Engineering Advances. IEEE, Los Alamitos (2008)
7. Hansson, H., Nolin, M., Nolte, T.: Beating the automotive code complexity challenge. In: National Workshop on High-Confidence Automotive Cyber-Physical Systems, Troy, Michigan, USA (April 2008)
8. Robocop project page, http://www.extra.research.philips.com/euprojects/robocop
9. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer 33(3), 78–85 (2000)
10. Wallnau, K.C.: Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon (2003)

The CoSi Component Model: Reviving the Black-Box Nature of Components*

Přemek Brada

Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic
brada@kiv.zcu.cz

Abstract. Many component models and frameworks have been created since the advent of component-based software engineering. While they all claim to follow fundamental component principles, the black-box nature in particular, a deeper look reveals surprising problems mainly in the component models built into the mainstream frameworks. In this paper we elaborate on these fundamental principles, analyse a selection of industrial and research component models in light of them, and propose a component model named CoSi. Its aim is to address the problems uncovered by the analysis while keeping the good aspects of current state state of the art models. It supports OSGi-style lightweight components with a rich set of features, and puts a strong emphasis on facilitating component comprehension and application consistency.

1 Introduction

Since the advent of component-based software engineering (CBSE), many frameworks have followed the component paradigm to support application development. Enterprise JavaBeans [1,2], CORBA Component Model [3], Spring [4], and OSGi [5,6] have been successful in practical applications and serve as good examples of the industrial applicability of component principles.

The universally accepted foundational works [7,8] list several constituting characteristics of components, of which the key one is the need to treat components as opaque black boxes with explicit interface declaration. These characteristics are enforced and the structure of the component's surface[1] is defined by a *component model* (more precisely, its meta-model part), which defines the (functional) features it provides for client components or declares as dependencies, as well as its behavioural specifications and extra-functional properties. The other

* This research was supported by the grant "Methods and models for consistency verification of advanced component-based applications" number 201/08/0266 from the Grant Agency of the Czech Republic.

[1] We use this term instead of the commonly used "interface" to avoid mistaking it for an interface type as in Java or IDL; a SOFA [9] synonym is *frame*.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 318–333, 2008.

key roles [10] of component models are to describe the allowed ways of inter-component bindings, i.e. architectural constraints, and aspects like component lifecycle management or capabilities of an underlying runtime framework.

The success of CBSE lies in pursuing the black-box idea thoroughly, much further than the preceding software engineering concepts (modules, objects). For example, the design of component-based applications involves activities like high-level architectural modeling with reused components, integration of (sometimes ill-documented) 3rd-party components, analysis of effects caused by component dependencies or updates, etc. In all these we need to reason about the complete component in the black-box fashion, without being distracted by implementation details, to overcome the conceptual complexity of the application.

1.1 The Goal of the Paper

In this paper we want to discuss in more detail the fundamental problems found in current component models and frameworks, and propose a component model called CoSi. Its aim is to address the problems while keeping the good aspects of current state state of the art models.

The discussion of these models is covered by section 2 which first elaborates on the foundational characteristics ("what constitutes a good component model") and then analyses several industrial and research component frameworks from these perspectives. It is motivated by the finding that adherence to component principles are generally acknowledged, at least superficially, by the component framework designers. Nevertheless, deeper look reveals surprising problems in the component models built into the mainstream frameworks. The approach taken in the study is a purist's one, showing what problems are caused by the component model deficiencies.

The CoSi component model is described in sections 3 and 4. Building on the results of the analysis and on an architectural rationale, the section describes the meta-model used, the capabilities of the underlying runtime framework, and several technical details. The strong and weak aspects of the proposed model are discussed in section 5, by means of comparing them with related work and component frameworks.

2 Current Component Models: Strengths and Weaknesses

There exist dozens of component models with different purposes, features and levels of popularity. In this section we first discuss the fundamental and practical properties a component model should posess, and present an digest of a broader analysis of several ones with respect to these properties; for a more detailed evaluation see [11].

The models we look at here are a sampling of the more popular ones, each representing a distinct approach – OSGi Release 4 [12], CORBA Component Model version 4 [3], and the iPOJO [13] research model.

2.1 Motivation and Evaluated Properties

The properties we were interested in are based on the characteristics generally perceived as constituting the "what and why" of software components. They have been treated in detail many times, e.g. in [7,10,8]. Though full agreement does not exist in the research community, the general understanding [14] is that a component is:

- a black-box functional unit with explicit (contractually specified) provided features and dependencies;
- a unit of independent deployment and substitution (ideally, these tasks require no human intervention); and
- subject of 3rd party composition, in contexts unforeseen at design time.

The first characteristics is possibly the most important one from both theoretical and practical standpoints. It takes the notions of modularity and information hiding [15] a step further conceptually as well as on the granularity scale. It also serves as a prerequisite for the latter two characteristics, and facilitates analysis and modeling of existing components (since surface representation can be reconstructed from the specification) needed during component-based development.

Two more characteristics can be added that we believe are important for CBSE success. A reasonably rich set of feature types is necessary, mainly to achieve sufficient expressive power relevant to the target domain of the component model – interfaces are the accepted base but more should be available, e.g. attributes, events. This enables clean component-based design and facilitates reuse. Recent experiences of the industrial CBSE have additionally shown the importance of the ease of development with components.[2] Practical aspects therefore need to be considered alongside the fundamental ones when evaluating a component model.

We therefore evaluate the component models from the following perspectives:

1. Are the components *truly black-box*? Does the component model prevent situations where features can be used by the clients (cross the component surface boundary) yet not described in component specification?
2. Can we easily obtain *complete representation* of component's surface, e.g. for modeling purposes? Is appropriate information accessible in machine readable form (declarative meta-data, introspection), without instantiating the component?
3. Does the model help *development efficiency*? Is the feature set of the component model sufficiently rich and at a suitable level of abstraction, does programming a component require the creation of only a few artifacts backed by good tool support?

In the following subsections we briefly describe each given model (plus associated runtime framework) and provide substantiation for its evaluation.

[2] Discontent with the overly complex development using EJB components was behind the success of lightweight solutions in POJO style, like the Spring framework [4] or OSGi.

2.2 Open Services Gateway Initiative (OSGi)

The Open Services Gateway Initiative (OSGi) platform [5,12] is an open Java-based framework for efficient component-based service deployment and management. The base component model of OSGi is relatively simple in terms of feature types and component implementation, but some of the standardized services in effect enhance it with additional features.

Black box In theory, bundles have a good chance to be considered clean black boxes – all features can be specified in manifest headers, and system services can provide additional meta-data. In practice, several OSGi core aspects violate the black box principle. In particular, provided services can be registered with the framework and required services looked up and bound, without being declared in the manifest.

Representation Again, the bundle manifest theoretically provides a good starting point for discovering features, and introspection together with possible additional meta-data from system services could be used to recover the details. However, all feature-related manifest headers are defined as optional and since OSGi Revision 4 the key headers `Export-Service` and `Import-Service` have been deprecated[3]. This can in the extreme case turn discovering component features into complete guesswork.

Development The component model provides only two kinds of application-relevant abstractions (packages and services), which is not very helpful for modeling. However, their granularity fits well with current application architectures and the conceptual simplicity is appealing also from the practical standpoint. OSGi is a code-based model with low structural overhead and good tool support.

2.3 CORBA Component Model

The CORBA Component Model [3] is the core of an industrial-strength component framework. It defines one feature-rich type of components and uses an Interface Definition Language (CIDL) plus a type repository for component specification.

Black box CORBA components are clean black boxes – only the information contained in CIDL specification is available to potential clients. Since implementation skeleton is generated from the specification, mutual correspondence is ensured.

Representation The declarations contained in the CIDL are stored in a type repository and therefore the representation of a given type can be obtained easily. The only disadvantage is that stand-alone component analysis is difficult if repository with its declaration is not accessible.

Development CORBA component model provides one of the richest sets of features, and has the advantage of multiple implementation language support. On the other hand, developing CORBA components is rather tedious by today's standards due to its IDL-first approach.

[3] In fact, they are not much used in practice – a sample of 112 bundles we analyzed contained only 5 bundles with service specification headers.

2.4 iPOJO

The iPOJO framework [13,16] aims at building an extensible service-oriented model on top of the OSGi infrastructure. Component features are defined by so called handlers which "wrap" the implemention POJO component class.

Black box Since handlers are the only legal way to interact with iPOJO component, the component model follows the black-box principle. However, we find the component meta-model conceptually unclean – the single concept of handlers is used to implement several rather different aspects, from functional features to component management.

Representation The component declaration (usually in XML format) contains all features it supports, and is clearly related to the respective Java implementation types. Component surface type reconstruction is thus straightforward. The possibility to implement new feature handlers however means there is no common meta-model which can make it extremely hard to study and understand an iPOJO component in isolation.

Development The core component model is rather poor on component features, only service interfaces (provided and required) can be specified. The framework has however been designed with extensibility in mind. New handlers can be easily implemented and linked to the framework, enabling components to declare and support new types of features.

2.5 Summary

To summarize the analysis from [11] and the above text, we can say that the component models differ (sometimes significantly) in the level to which they achieve the fundamental properties of CBSE. Table 1 presents the results in a concise form including some component models not discussed above; simplified classification values were used to express the level for individual perspectives.

In one sentence and exaggerating slightly, we could say that any given component model is either not too black-box, or not too practical; sometimes both.

Table 1. Component model comparison overview

Characteristic	**OSGi**	**EJB**	**CCM**	**iPOJO**	**SOFA**
Black-box	poor	poor	good	good	good
Representation	moderate	moderate	moderate	good	moderate
Development	good	good	moderate	moderate	moderate

3 The CoSi Component Model and Framework

The findings about the fundamental properties of component models are not too encouraging, especially for the industrial ones. Since the development simplicity is something we consider a value for real-world success of components, we have designed an experimental component model that aims to take the best of both worlds – strictly adhering to the fundamental concepts yet providing sufficient practicality. The model is called *CoSi*, an acronym from *Components Simplified*.

3.1 Rationale

The ideas that were driving the design of the CoSi component model are as follows, roughly in decreasing order of importance:

- Strong pure black-box. That is, nothing is acccessible from outside a component except what is explicitly declared as such.
- Complete yet minimal feature specification. The component specification has to contain all basic information about features (existence, name, type), introspection can be used to augment the necessary details.
- Maximum simplicity in the underlying infrastructure[4]. The emphasis is on the component model properties, not on the framework capabilities. In practice this means no distribution, remoting, security, or dynamic updates, simple runtime framework, preference of text over XML formats. OSGi was a strong inspiration in this respect.
- Support weakly typed languages. The component model is designed so that it enables research in suitability of scripting languages for component implementation especially in the context of component substitutability. The Groovy scripting language was chosen for component implementation, for its close ties to Java.
- Reasonable feature set. We want to include practically useful features like events and streams, and use named features; CORBA Component Model was inspirative in this respect.

In some architectural and practical design decisions the model follows the ideas of the OSGi core, which we think in principle strikes a good balance between (potential) rigour and simplicity, despite the shortcomings described above. In fact, CoSi could be seen as an attempt to build an OSGi-like component model which is formally strong from the black-box and surface representation perspectives.

This strength is achieved in particular by the emphasis on complete specification and a rich feature set. The CoSi component model design aims to make it easier and practical to use high-level abstractions for inter-component communication (thanks to the rich feature set) and to model/visualize existing components (being able to reconstruct details of all surface elements starting from component specification).

3.2 The Component Model of CoSi

The CoSi meta-model allows flat (not hierarchical) components with four feature types, a lifecycle management interface and component surface specification in the form of meta-data contained in a descriptor file; see Figure 1.

The *component* has a provider, name and version which together provide its unique identification. Four types of *features* can be provided and/or required by a component. Each feature has a name (where applicable), type and optional attributes like version identification. Several features of the same type can be specified, distinguished by their names.

[4] That's where the "simplified" part of the model's name comes from.

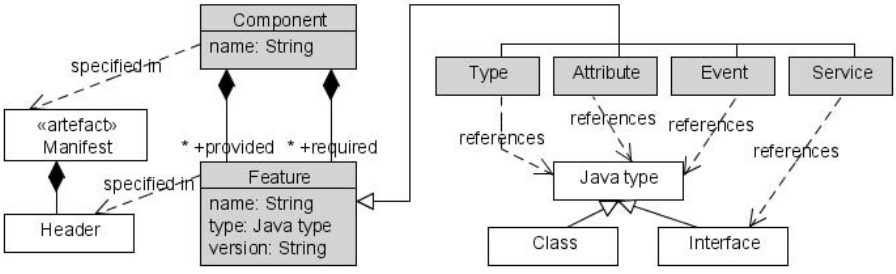


Fig. 1. The CoSi component meta-model

- *Service* is an implementation of functionality, specified by a Java interface. Provided services are registered with the runtime container which then mediates the bindings to the requiring components. The binding is realized by *service reference* objects.
- *Type* referes to a language class or interface. Provided types are exported by the component's packages, and are accessible by the requiring components via the exporter's classloader.
- *Events* enable messaging among components, mediated by a system message service. Events are named and typed, which enables event consumers to set filters on the kinds of events they subscribe to. Both asynchronous and synchronous event delivery is supported.
- *Attributes* define typed values which can be set or read by the component. Attributes can be read-only and read-write, and are implemented as *(key:String, value:Object)* pairs accessible via a system-wide attribute registry.

The component model has no abstraction of the *application architecture* (like OSGi, unlike e.g. SOFA [17]), due to its flat nature. Bindings between the components, that is between pairs or tuples of provided and required features, are therefore expressed intrinsically by the matching provided-required feature pairs and managed by the CoSi container (see next).

3.3 Component Runtime – The CoSi Container

The deployment, component lifecycle management, static and dynamic feature binding (type resolution, service management), and user interactions are the task of the *container*. The core of the container is quite small. It keeps the various lists, most importantly component and service registries, and exports several standard packages and interfaces for use by the installed components.

An integral part of the container is however a set of standard system services, which realize core framework functionality. In the current CoSi version there are two such services: the *system service* which implements container startup and shutdown sequence, component lifecycle management and meta-data querying, and *message service* which implements the message queue and the delivery

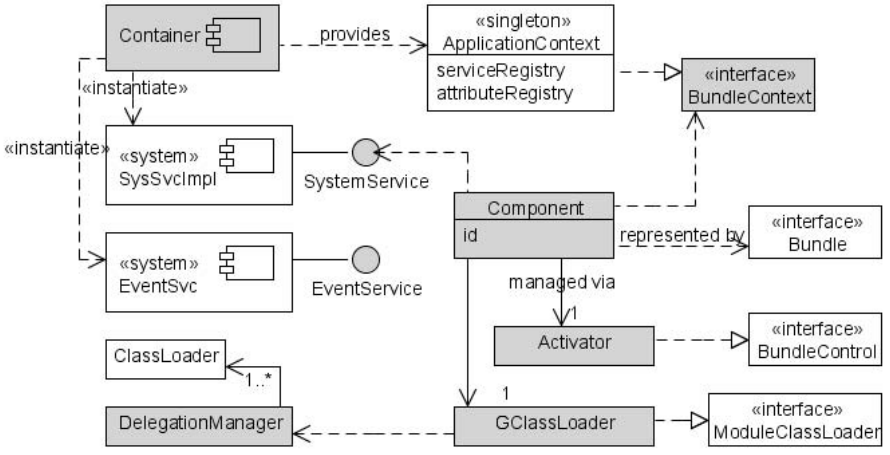


Fig. 2. The CoSi container and runtime interfaces

mechanisms for *Event* features. These services are implemented by system components, with internal structure and run-time representation of standard CoSi components. However, they are compiled into the framework implementation library and access some of its core functionalities directly.

The CoSi distribution provides also some additional *standard services*: simple input/output, logging and shell. All of these are implemented as standard components (using Groovy and not bundled in the framework library) which get loaded at framework startup according to its configuration.

The framework run-time uses a *classloader architecture* to isolate the individual component type spaces (similarly to OSGi, cf. [12, Section 3.4]) and allow the interaction between Java and Groovy implementation classes. There is one instance of so called *module classloader* per component, which loads resources located on the bundle's own classpath. It is capable to load both Java classes (from `.class` files) and Groovy scripts which it translates on-the-fly to class code through the Groovy parser.

In addition, this classloader loads classes of required features, by contacting the classloader of the exporting component through a delegation manager. Finally, it references a system classloader which loads the types that must be shared – system classes (`java.*`, `com.sun.*`, `groovy.*` etc. packages) and CoSi framework classes.

The component can access key container services and registries via an object typed to the `BundleContext` interface. This is injected to the component upon its activation. The context object provides functionalities that allow the component to register and obtain services, read and write attributes, obtain its own metadata, and access standard input and output streams provided by the container.

3.4 Component Lifecycle and Management

Each installed component (the package) has a unique identifier assigned by the container and is represented by a `Bundle` interface object. This has methods to query component state and meta-data and manage state transitions. Further, a compulsory *control class* or activator of the component provides operations for its initialization and lifecycle management.

The *component lifecycle* is essentially the same as in OSGi, cf. [12, Section 4.3]. It includes the following key states: INSTALLED (component package has been successfully read, verified for formal correctness and completeness, and assigned an identifier), RESOLVED (static feature type dependencies have been resolved, component has not yet been started or has been stopped), STARTED (a resolved component is running, after calling the activator object's `start()` method).

Components can be installed, updated, and uninstalled. The update mechanism is simpler than in e.g. SOFA or OSGi: the component is stopped if necessary, uninstalled, the new package installed and resolved. The component's identifier is preserved during the process.

3.5 Implementation Classes and Distribution Format

The author of a CoSi component needs to create at least the *control class* of the component (known as Activator in OSGi world) which implements the lifecycle callback methods. Then, an interface plus an implementation class are usually written for each provided service or event sink, and for all provided types. Last but not least, the manifest file containing specification of the component's surface features and other meta-data has to be created.

The compulsory `MANIFEST.MF` file, written in the standard manifest text format, contains *manifest headers* which declare component's properties. The `Cosi-Version` meta-header enables versioning of the whole component meta-model. The OSGi rules [12, Section 3.2] for header format and parameters are used.

The distribution format of the CoSi component is a JAR file with structure resembling OSGi bundle archive. The `bin/`, `lib/` and `imports/` directories are required, holding respectively the component implementation classes (in Groovy or `.class` format), bundled libraries (if any), and class types of component's required features as a "snapshot" of those types used when the component was developed. Additional directories and files can be included as needed, e.g. for documentation and component's resources like icon or localization bundles.

At present, the CoSi framework does not use any kind of elaborated component repository. The container uses a filesystem directory at a configurable location from which components are automatically installed upon container startup, and it imports the distribution packages of installed components into its internal cache. Beyond that there are no structures or services through which the component distribution packages could be obtained; that is left to future extensions.

3.6 Application Consistency

The CoSi framework enforces the consistency of component application through several integrity measures. First, the container raises an exception if an attempt is made to register a provided or bind a required feature that is not declared in the component's meta-data. This enforces the components' black-box nature by preventing bindings to undeclared component internals.

Next, a component cannot be installed, or updated, if its dependencies cannot be satisfied by the components currently installed. For example, if the component to be installed declares an attribute dependency, the container checks whether an exporter of the attribute with the same type already exists in the framework. While these controls cannot guarantee that the depended-on feature *will* be available at the time the component needs it, they protect the application against run-time errors caused by clearly unsatisfiable dependencies.

Quite importantly, the container ensures consistency of component bindings during the lifecycle state transitions: a STARTED (resp. INSTALLED) component cannot be stopped (resp. uninstalled) if it provides a feature bound to (resp. required by) an existing client component(s). Again, this ensures application consistency is maintained throughout architectural changes.

4 Example

To illustrate the concepts described above, let us now show an example of a simple CoSi application. It consists of a weather station component connected to sensors (e.g. thermometers); the connection is obtained via a sensor registry service, which the individual sensors need to contact in order to become available. Figure 3 provides a model of the application.

The measuring station consists in this simple implementation of only two files – the control class and the manifest. The manifest file looks as follows; notice how it reflects the component's connections, corresponding to the architecture shown in Figure 3.

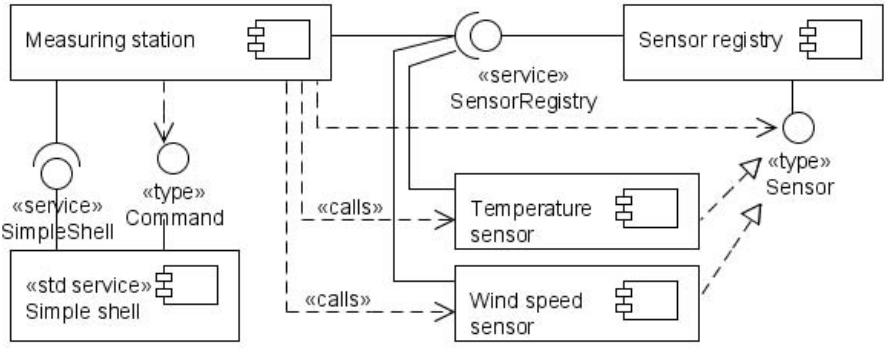


Fig. 3. Example application architecture

```
Bundle-Name: MeasuringStation
Bundle-Version: 1.0.0
Control-Class: cz.zcu.kiv.measuringstation.impl.Activator
Require-Interfaces: cz.zcu.kiv.simpleshell.SimpleShell,
  cz.zcu.kiv.sensorregistry.SensorRegistry
Require-Types: cz.zcu.kiv.simpleshell.Command,
  cz.zcu.kiv.sensorregistry.Sensor
Provide-Attributes: sensor.numValues; type=java.lang.Integer
```

The following fragment shows parts of the control class. It shows how a service reference is obtained from application context and then used, as well as the export of a provided attribute. Its value is later obtained by the `Sensor` objects using the context's `getAttributeValue()` method.

```
import cz.zcu.kiv.sensorregistry.SensorRegistry;
// ... further imports omitted for brevity

public class Activator implements BundleControl, Command {

  private SensorRegistry sRegistryService;
  private List<Sensor> sensors;

  /* Component initialization code */
  public void start(BundleContext context) throws Exception {
    sRegistryService = (SensorRegistry) context.getService(
        "cz.zcu.kiv.sensorregistry.SensorRegistry");
    // service call to obtain sensor references
    sensors = sRegistryService.getAllAvalilableSensors();
    // export the provided attribute
    Integer numValues = ... ; // obtain from a config source
    context.setAttributeValue("sensor.numValues", numValues);
  }

  // ... further methods omitted for brevity
}
```

The service registry creates and exports the corresponding service reference through the `BundleContext.registerService()` method. The component implementation consists of the manifest (which declares this provided service) and four classes: `Sensor` and `SensorRegistry` in a `cz.zcu.kiv.sensorregistry` package are the public types used in communication with other components (cf. the measuring station required interfaces), the `Activator` and `SensorRegistryImpl` in package `cz.zcu.kiv.sensorregistry.impl` are "hidden" internals not accessible by other components.

As a last example, we show a part of a user session with the CoSi container and this application. We have augmented the main component's implementation with an attempt to set another attribute, to present the black-box checks enforced by the container (compare with the output of `attributes 6` command). After stopping the wind speed sensor component, the measures which the station outputs contain only the values from the temperature sensor.

```
D:\work\research\CoSi\>.\start
Starting CoSi framework...

Bundle measuringstation.jar cannot set attribute
sensor.windDirection because this attribute isn't
in Provide-Attribute header of manifest.mf.

>attributes 6
Provided attributes of bundle measuringstation.jar (id 6)
-----------------------------------------------------
sensor.numValues (java.lang.Integer)

>stop 4                              <-- id of wind sensor
>measure
Sensor type: Temperature
19,9 degree Celsius
13,1 degree Celsius
>
```

5 Discussion and Related Work

Among the foundational publications on fundamental component properties, which the CoSi model strives to achieve, are the works by Szyperski [18] and by the SEI CMU team [8]. In these schools of thought on fundamental component concepts, there is a clear statement of the need for opaqueness and consequent explicit description of component interface.

The analysis presented in section 2 attempts to capture component properties not dealt with in similar surveys. Bachmann [8] and Lau [14] mention the fundamental constituent characteristics of components, but the consequences of black-box nature are not pursued far enough. Works dealing with individual technologies are quite frequent. For instance, our previous work [19] discusses the EJB component model, Hall et al [20] list several issues with the OSGi framework (including the problems resulting from undeclared services).

5.1 Strengths and Unique Features

The proposed CoSi model has several unique characteristics which we believe form a novel contribution to the field of component models. The key advantage of CoSi over its industrial counterparts is the emphasis on enforcing fundamental component characteristics, the black-box nature in particular. First, feature declaration is a necessary condition for its use by implementation code – no provided feature can be registered or exported unless it is specified in manifest, similarly with required features. The container enforces this by runtime controls.

Secondly, the container keeps track of component bindings similarly as OSGi does but prevents lifecycle state transitions of providing components which would break these bindings. This conservative approach is in some respect close to the architectural consistency constraints enforced by e.g. SOFA [17]. CoSi however

retains the flexibility of creating the architecture ad-hoc using late binding [12,4] rather than fixing it a-priori in component architectural specification.

Our model emphasizes the use of bundling complete meta-data with the component. Because all features of a CoSi component must be declared in the manifest, including their class type names, there exists a well-defined single starting point for obtaining a complete representation of component's surface. The complete type representation of features is obtained by introspecting the relevant component's classes – this can be done on a standalone component package, interaction with CoSi container is not necessary. In the case of required features (their types are not available until the component is deployed and resolved), introspection is in CoSi uniquely facilitated by bundling the relevant types with the component in the `imports/` directory.

In comparison, OSGi tends to treat information in manifest headers as optional which makes it hard to discover component features. This weakness has been addressed by the bundle repository [21] and declarative services (originating from [22]) specifications. The problem of this "aggregate" platform is the resulting incoherent set of abstractions and formats scattered through several physical locations. Neither of these also helps in reconstructing type information of the required features. The CORBA and SOFA approach [17] solves both problems by using meta-data repository. This however need not be accessible at the time of analysis, forming an obstacle to stand-alone analysis of the components.

Among the minor points is the use of named features which allows several features with the same type to be provided by a component. The EJB model [1] is a standard example of an opposite approach, which has the unpleasant consequence that a bean cannot distinguish through which role a client accesses its functionality.

Last but not least, an aspect important mainly with respect to future experimentation is the choice of a weakly typed scripting language for component implementation. This is a unique feature of CoSi not found in any of the current component models. It enables the developers of CoSi applications to balance run-time efficiency and development ease – it is very simple to create and modify Groovy-based components (no need to recompile), and the ones that require optimizations can be relatively simply transformed into compiled Java bytecode which then loads and runs quickly at run-time.

5.2 Restrictions and Shortcomings

The goal of simplicity in CoSi architectural design limits its abilities, and therefore usage contexts, in several dimensions. From theoretical standpoint, the component has a quite simple lifecycle and control interface which cannot be extended or modified in any way. Also, no extra-functional and semantic properties are used at the component and/or feature level (unlike [17] we believe the control interface does not belong to the extra-functional properties). Other research component models like SOFA or Palladio [23] provide much richer capabilities in these areas, plus hierarchical component models which CoSi avoids (at least in the present version).

From practical standpoint, CoSi lacks support for aspects not directly related to the component model itself – distribution, security, component repository access, and so on. Without these engineering aspects its applicability in real world contexts is limited; however, the framework was not designed with industrial application in mind. Concerning development ease, in the current version of the CoSi platform the component implementation (typically the control class) must manually handle registration of provided features and binding of required ones. It would be much easier for practical development to employ dependency injection mechanism for this purpose.

Another aspect which can be seen as cumbersome for daily development is the choice of individual types, not packages, as one of the surface features in the current platform version. While we acknowledge that it requires the developer to exercise some effort in creating component specification, we note that it is very easy to generate the relevant manifest headers automatically. A corresponding enhancement is planned for the next version of the CoSi component model.

6 Conclusion and Future Work

In this paper we have presented a component model called CoSi, which blends rigorous adherence to fundamental component concepts with practical usefulness and simple design inspired by the core OSGi framework. The text of the paper has covered the principles of our approach, results of an initial analysis of other component models, and a comprehensive description of the CoSi component model and runtime platform.

The presented component model aims at improving the possibilities to reliably reason about component surface structures, especially during design and deployment activities, as well as enhancing the overall safety of the applications. The first goal is attained through explicit declaration of all features in manifest as the first well-known point, plus by ensuring no other features can actually be exported/required (via container checks). To achieve the second goal, the CoSi container prevents component state changes which would adversely affect its clients or its internal functionality.

A working implementation of the CoSi framework has been built [24] and successfully used for experiments. The container and system services are written in Java, the standard components like shell use Groovy implementation. Preliminary experiences with the implementation have shown no major problems in creating, deploying, resolving and running components in a component model which stresses explicit declaration of all surface features.

We believe CoSi contributes to the current state of the art in component based software engineering, by demonstrating that it is possible to create a component model that is at the same time lightweight, feature rich, formally well founded, and facilitating component comprehension.

The component model itself is open for future changes, and work is currently on the way to enhance it with extra-functional properties. Experiences from our research make us consider finer decomposition of the component lifecycle,

notably the update process, to allow intercepting state transitions. The platform will also be used in the research of component substitutability and advanced component meta-model concepts.

Acknowledgments. The author would like to thank Bretislav Wajtr and Vojtech Liska for fruitful discussions and the work on CoSi implementation.

References

1. Sun Microsystems: Enterprise JavaBeans Specification, Version 2.1. (November 2003)
2. Sun Microsystems: Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements. JSR220 Final Release (May 2006)
3. Object Management Group: CORBA Component Model Specification, Version 4.0 OMG Specification formal/06-04-01 (April 2006)
4. Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., Sampaleanu, C.: Professional Java Development with the Spring Framework. Wiley, Chichester (2005)
5. The OSGi Alliance: OSGi Service Platform, Release 3 (March 2003), http://www.osgi.org/
6. The OSGi Alliance: OSGi Service Platform, Release 4 (August 2005), http://www.osgi.org/
7. Szyperski, C.: Component Software. ACM Press, Addison-Wesley (1998)
8. Bachmann, F., et al.: Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute. Carnegie Mellon University (2000)
9. Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: architecture for component trading and dynamic updating. In: Proceedings of ICCDS 1998, Annapolis, Maryland, USA. IEEE CS Press, Los Alamitos (1998)
10. Heineman, G.T., Councill, W.T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, Reading (2001)
11. Brada, P.: The strengths and weaknesses of current component models from black-box perspective. Technical Report DCSE/TR-2008-08, Department of Computer Science and Engineering, University of West Bohemia (July 2008)
12. The OSGi Alliance: OSGi Service Platform Core Specification, Release 4.1 (April 2007), http://www.osgi.org/
13. Escoffier, Hall, Lalanda: iPOJO: An extensible service-oriented component framework. In: Proceedings of IEEE International Conference on Services Computing (SCC 2007), pp. 474–481. IEEE Computer Society, Los Alamitos (2007)
14. Lau, K.K., Wang, Z.: A taxonomy of software component models. In: EUROMICRO 2005: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, Washington, DC, USA, pp. 88–95. IEEE Computer Society Press, Los Alamitos (2005)
15. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM (December 1972)
16. Escoffier, C., Hall, R.S.: Dynamically adaptable applications with iPOJO service components. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829. Springer, Heidelberg (2007)
17. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: Proceedings of SERA 2006, Seattle, USA, August 2006, pp. 40–48. IEEE CS, Los Alamitos (2006)

18. Szyperski, C.: Component technology - what, where, and how? In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, May 2003, pp. 684–693 (2003)
19. Brada, P.: The ENT meta-model of component interface, version 2. Technical Report DCSE/TR-2004-14, Department of Computer Science and Engineering, University of West Bohemia (September 2004), http://www.kiv.zcu.cz/publications/
20. Hall, R.S., Cervantes, H.: An OSGi implementation and experience report. In: Proceedings of the Consumer Communications and Networking Conference (CCNC 2004), pp. 394–399 (January 2004)
21. OSGi Alliance, Hall, R.: Bundle repository. Technical Report RFC 112, OSGi Alliance (2005), http://www.osgi.org/download/rfc-0112_BundleRepository.pdf
22. Cervantes, H., Hall, R.S.: Automating service dependency management in a service-oriented component model. In: Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (2003)
23. Reussner, R., et al.: The Palladio component model. Technical report, Universitaet Karlsruhe (May 2007)
24. Brada, P.: The CoSi component model. Technical Report DCSE/TR-2008-07, Department of Computer Science and Engineering, University of West Bohemia (July 2008)

Ada-CCM: Component-Based Technology for Distributed Real-Time Systems*

Patricia López Martínez, José M. Drake, Pablo Pacheco, and Julio L. Medina

Departamento de Electrónica y Computadores, Universidad de Cantabria,
39005-Santander, Spain
{lopezpa,drakej,pachecop,medinajl}@unican.es

Abstract. This paper proposes a technology for the development of distributed real-time component-based applications, which takes advantage of the features that Ada offers for the development of applications with predictable temporal behaviour, and which can be executed in embedded platforms with limited resources. The technology uses the Deployment and Configuration of Component-based Distributed Applications Specification of the OMG for describing the components, the execution platforms and the applications. The framework defined in the Lightweight CCM standard of the OMG is taken as the basis of the internal architecture of the components and the applications. It has been extended with a number of features to make the temporal behaviour of the applications predictable. Among these extensions, the usage of CORBA has been replaced by special distributed components, called connectors, which implement the interaction between components by means of predictable and customizable communication services. Besides, special mechanisms have been introduced in the environment to make the threading characteristics of the components configurable. The technology fixes the responsibilities and the knowledge required by each actor involved in the component-based development process, and for each of them it defines the input and output artifacts that they have to manage.

Keywords: Ada 2005, Component-based, embedded systems, real-time, OMG.

1 Introduction

The design of real-time software for embedded systems has a strategic interest in the industry nowadays. In many application areas, like robotics, industrial control, automotive, etc., systems are built by assembling subsystems (controllers, vision systems, carburation systems, etc.). A subsystem may be equipped with its own embedded processor, or deployed in a number of them, each of which is in charge of controlling its own hardware, and the communication among them is achieved by means of a

* This work has been funded by the European Union's FP6 under contracts FP6/2005/IST/5-034026 (FRESCOR), FP7/224330 (ADAMS) and ArtistDesign, EU FP7 NoE 214373 and by the Spanish Government under grant TIC2005-08665-C03 (THREAD) and EVOLVE. This work reflects only the author's views; the EU is not liable for any use that may be made of the information contained herein.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 334–350, 2008.

dedicated network (Ethernet, CAN bus, firewire, etc.). This architecture provides considerable modularity and reconfigurability, and it minimizes and standardizes the wiring.

The increasing capacity and memory provided by the processors, and the subsequent rise in the amount of functionality that they must support, together with the distributed nature of the execution platforms, and the real-time requirements of the final applications involved, make the software for this kind of systems very complex. Applying component-based design strategies to this domain offers several advantages:

- It provides a simple architecture based on interfaces instead of protocols between subsystems.
- The reconfiguration of a system can be achieved by modifying the deployment plan, without requiring any hand-made code modification. It also simplifies the evolution and versioning of systems since it is only required to replace or add new components with well-defined functionalities.

Conventional component technologies are not easily adaptable to embedded systems, since they require a large amount of services from the operating systems, file systems, middleware or networks, which are not compatible with the limitation of resources suffered by them. Various proposals dealing with the adaptation of CBSE to real-time systems have appeared in the last years. Some companies have developed their own solutions, adapted to their corresponding domains. Examples of that kind of technologies are Koala [1], developed by Philips, or Rubus [2], used by Volvo. These technologies have been successfully applied in the companies that created them, though none of them have stimulated an inter-enterprise software components market. However, they have served as the basis of other academic approaches. The Robocop component model [3] is based on Koala and adds some features to support analysis of real-time properties. Similarly, Rubus has been used as the starting point of the SaveCCT technology [4], which is focused on control systems for the automotive domain; and under appropriate assumptions for concurrency, simple RMA analysis can be applied and the resulting timing properties introduced as quality attributes of the assemblies. From the Ada language perspective, even though it is significantly used in the design and implementation of embedded real-time systems, we have not found references of its usage in support of component-based environments.

This paper proposes a component-based technology, denominated Ada-CCM, which is specifically conceived for embedded, distributed and real-time systems. The key aspects of the technology are:

- It follows the components specification style and the programming model proposed in the Lightweight CCM (LwCCM) [5] specification of the OMG. Therefore, a container/component pattern is used, though with an essential difference: CORBA is not the communication mechanism used. The connection between a facet and a receptacle is always local, and the communication between remote components is achieved by means of special distributed components called connectors. Besides, special mechanisms have been included in the containers to make the temporal behaviour of the application execution predictable.

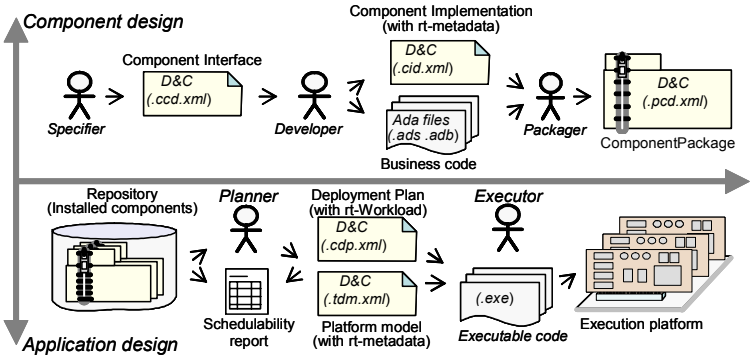


Fig. 1. Actors and main artifacts in components and applications design

- Both the business component implementations and the containers code are written in Ada 2005 [6]. Making use of the Ada's native support for concurrency, scheduling policies and synchronization mechanisms, it is possible to generate code with predictable temporal behaviour. Ada is intended for embedded systems, and there are small foot-print run-time libraries for Ada that can be executed on bare embedded computers. The new version of the language is essential to this work, since it introduces support for multiple inheritance based on interfaces, which are key aspects in the development of component-based technologies.
- The external interface and the internal implementations of components, execution platforms and deployment plans are described following the Deployment and Configuration of Component-based Distributed Applications Specification of the OMG (D&C) [7]. It has been extended to include metadata about the temporal behaviour of components, platforms and applications. This information is used to analyse the schedulability of the application as part of the development process.

The responsibilities and the artifacts that serve as inputs and outputs for the different actors that take part in the development process of an application are precisely defined in the proposed technology; they are briefly sketched in Figure 1. A detailed explanation of the development process and the involved actors and artifacts is given along the paper. Due to the real-time nature of the developed applications, this process adds a number of aspects to the standard one. The developer must formulate, together with the business code, the description of the temporal behaviour of the component. Real-time models describing the capacity provided by the elements of the execution platforms are also required. After defining the structure of an application by means of the deployment plan, the planner can build its temporal behaviour model. Based on the real-time requirements established in the specification of the application, he defines the workloads, which are the basis for the schedulability analysis. The results of the analysis are the set of scheduling parameters with which the component instances are configured, as well as the specification of the platform resources that shall be reserved in order to schedule their execution timely.

The paper is organized as follows, Section 2 describes the process of generation of a deliverable component in the technology. In Section 3, the reference model of the technology is explained, together with the structure of Ada packages to which a component is mapped. Section 4 details the process of development of an application built as an assembly of components. Section 5 describes the development suite. Section 6 details the features of the execution platform and an application example. Section 7 presents the way in which real-time models are added to the description of components and platforms, and introduces the modelling and analysis suite used in the technology. Finally, Section 8 summarizes our conclusions and future work.

2 Component Development Process

Figure 2 shows the process that is followed to generate a deliverable component, which will be able to be automatically assembled and executed in future applications.

When the *specifier*, who is an expert in the application domain, finds out that a certain functionality is demanded, he creates the specification of a new component that satisfies it. The component specification is formulated according to the D&C specification, by means of a Component Interface Description (.ccd file). As it is explained in Section 7, the D&C specification has been extended to incorporate special real-time composability requirements, and the metadata related to the temporal behaviour of the component. Besides, with the purpose of controlling the number of threads managed by a component, and making their scheduling parameters configurable (i.e. the priority), an important aspect has been introduced in the technology. Each thread needed by the business implementation of a component to implement its functionality, is required by means of an special activation port declared on its specification (the way in which the container manages these activation ports is explained in Section 3).

As an example, using the LwCCM graphical notation, Figure 3 shows the specification of the *SoundGenerator* component. This component is part of an application we have developed to test the technology. It offers a facet, *playerPort*, which implements the *I_Player* interface and is used by the client components to generate different sounds. It is an active component, since it requires a thread to play the sound without forcing the client to be blocked until the sound is completed. The thread is demanded by means of the declaration of the *soundThread* activation port. The component declares a configurable property, *soundThreadPeriod*, which represents the period with which the thread provided by the container will invoke the update() procedure corresponding to the soundThread port (see Section 3).

The *developer* writes the business code of the component as a set of Ada packages (.ads and .adb files). The code has to implement the *Component Business Interface*. This interface is generated using the *ComponentTemplateGenerator* tool, which takes the specification of the component, and the IDL descriptions of the related functional interfaces as inputs. It defines the set of methods that the business code must implement in order to be managed by the container in an automatic way. It has no dependencies with the technology, so the component developer is free to design the business code without having to be aware of any internal detail of the technology. The developer has complete knowledge about the internal behaviour of the component, so he

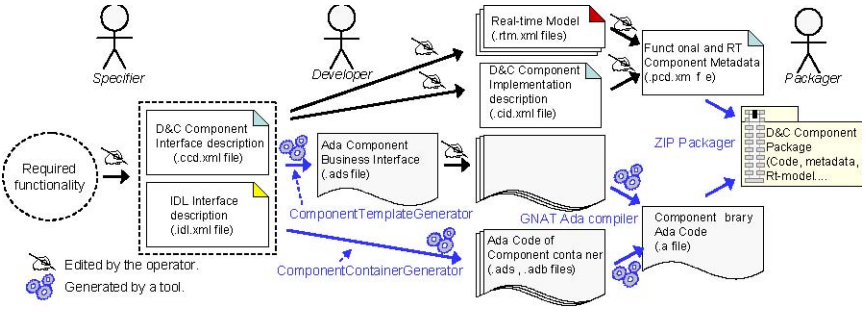


Fig. 2. Actors and artifacts involved in component development with AdaCCM

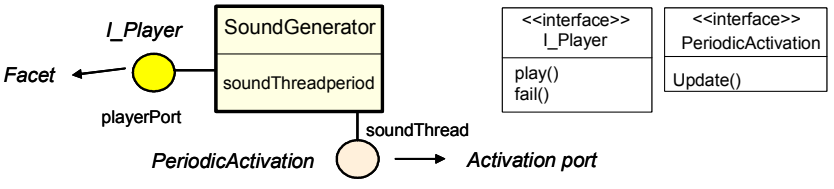


Fig. 3. SoundGenerator component declaration

has to create, together with the code, the real-time model that describes the temporal behaviour of the component. Besides, the developer has to specify the requirements that the component imposes on the platform to be able to execute. All this information is described by means of a D&C's Component Implementation Description (.cid file).

The *packager* carries out the last phase of the process, which consists in building the package that constitutes the deliverable component. Taking the specification of the component as input, a new code generation tool, called *Component Container-Generator*, generates the set of Ada source files (.ads and .adb files) that implement the container of the component. The container groups all the resources that are used to adapt the business code implementation to the execution environment (its structure is explained in the next section). These files are compiled together with the business code using the standard GNAT Ada compiler and a library is generated (.a file). This library is the only artifact that is required to execute the component on the target platform. Finally, the packager gathers all the information available about the component, and creates and publishes the package that describe the component. This package includes both the binary code of the component and the metadata (both functional and non-functional) that allow a future user to decide about the suitability of the component in an application, and also describe the way in which the component can be instantiated and executed. This package constitutes the deliverable component and the corresponding metadata is defined according to the Package Configuration Description element of the D&C (.pcd file).

3 Component Architecture

A full component implementation must address two complementary aspects:

- It has to implement the business functionality that it offers through its facets, making use of its own business logic and the services of other components accessed through its receptacles. This aspect concerns the application domain in which the component functionality is required.
- It must include the mechanisms that are required to instantiate, connect and execute the component in the corresponding platform and framework. This aspect is addressed by implementing the appropriate interfaces that allow managing the component in a standard way. This aspect is related to the component technology used, in our case LwCCM.

The architecture of a component proposed in this technology follows an structural pattern that achieves independency of the Ada packages that implement each aspect.

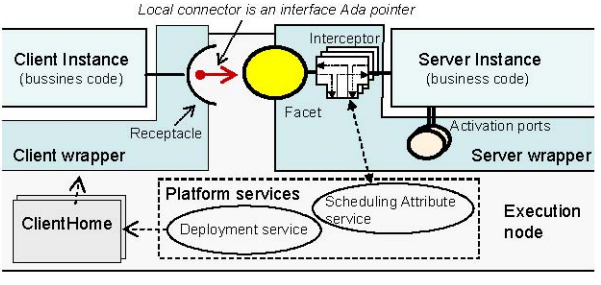


Fig. 4. Reference model of the technology

The packages that implement the technology related aspects are completely generated by automatic tools, taking the specification of the component as the only input. The component developer only has to design and implement the business code of the component, without having to have any knowledge about the underlying technology. The architecture of a component is generated according to the reference model of the technology, which is shown in Figure 4. It is based in the container-component framework proposed in LwCCM, but it has been extended with some new features required to make the behaviour of the application execution predictable:

- In order to make the threading and scheduling characteristics of an application configurable, and therefore, to control its schedulability, the business code of a component has not internal threads. The internal activity of a component is defined through the set of activation ports declared in its specification. These ports are recognized by the container, which creates and activates the corresponding controlled threads to execute the activity of the component once it is instantiated, connected and configured. These activation ports can implement one of the predefined interfaces: PeriodicActivation or OneShotActivation. The OneShotActivation interface declares a `run()` procedure, which will be executed once

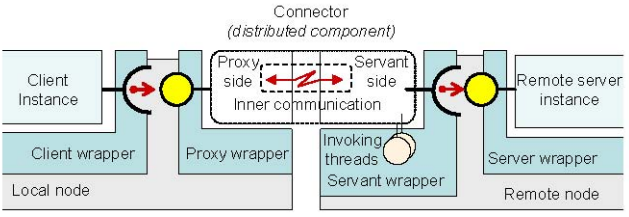


Fig. 5. Connector component

by the created thread, while the PeriodicActivation interface declares an `update()` procedure, which will be invoked periodically. A component can declare several activation ports, each of them representing an independent entity of concurrency. Activation ports are declared in the component specification, and all the elements required for their execution are created automatically by the container generation tool. Their configuration parameters, which include the thread priorities as well as the activation periods (in case of PeriodicActivation ports), are assigned to each component instance in the deployment plan.

- The connection established by each receptacle in a component is always local and it is implemented by an Ada pointer to the corresponding interface. If the connection between components is local, the pointer access directly to the facet of the server component. If the connection is remote, it is implemented by an specialized component called connector. As it is shown in Figure 5, a connector is a distributed component, composed by two parts: the proxy side which is instantiated in the client node, and the *servant* side, which is instantiated in the server node. The proxy offers a local facet to the client component and it includes the synchronization mechanisms for the invoking thread. The servant side has a receptacle which connects with the server facet and it includes and manages the threads that carry out the remote invocations. The communication mechanisms between the two parts (marshalling and unmarshalling of the invocation and return parameters, and transmission and dispatching of messages) are internal to the connector and depend on the communication service chosen for its implementation. The code of the connector is completely generated by automatic tools according to the interface of the connected ports, the location of the components, and the communication service used for the connection. We have developed connectors which use directly the RT-EP protocol [8], which is a real-time protocol implemented over Ethernet. This kind of connectors are suitable for connections with real-time requirements. An alternate implementation with no prioritized messages has been made using GLADE [9], an implementation of the Ada Distributed Systems Annex (DSA).
- Interception mechanisms [10] and a special internal service are introduced in the container to control non functional features of the component service executions. In our technology they are specifically used to control the scheduling parameters with which each invocation received in a component operation is executed. Based on the configuration parameters assigned to each instance in the deployment plan, each interceptor knows the scheduling parameter which

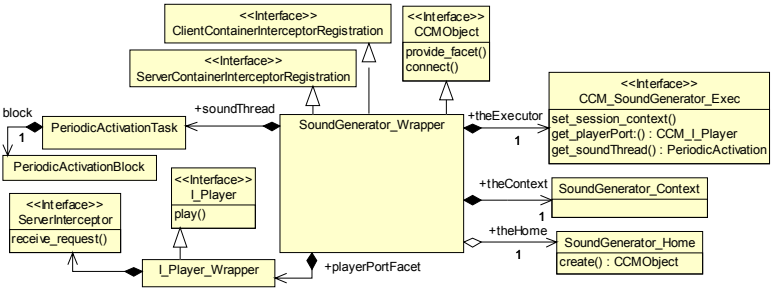


Fig. 6. Component wrapper structure

corresponds to the current invocation, and uses the *SchedulingAttributeService* to modify it in the invoking thread. With this strategy, different schemes for scheduling parameters assignment can be implemented. Besides common assignment policies, like Client Propagated or Server Declared [11], our technology allows to apply an assignment based on the transactional model of the application. With this policy, a service can be executed with different scheduling parameters inside the same end-to-end flow depending on the particular step inside the flow in which the invocation takes place. This scheme enables better schedulability results [12].

For each component specification, four Ada packages are generated. The first package represents the adapter of the component and includes all the resources to adapt the business code of the component to the platform, following the interaction rules imposed by the technology. The wrapper class of the component is defined in this package. This class implements the equivalent interface of the component, which LwCCM establishes as the only interface that can be used by clients or by the deployment tools to access to the component. With that purpose, the class implements the CCMObject interface, which, among others, offers operations to access to the component facets, or to connect the corresponding server components to the receptacles. Besides, the capacity to incorporate interceptors is achieved by implementing the Client/ServerContainerInterceptorRegistration interfaces, a modified version of the homonymous interfaces defined in QoSCCM [10]. As it is shown in Figure 6 for the SoundGenerator component, this class is a container which aggregates or references all the elements that form the component:

- The component context, which includes all the resources required by the component to access to the components that are connected to its receptacles.
- The home, which represents the factory used to create the component instances.
- The executor of the component, which represents the link to the real business code implementation and whose structure is explained below.
- An instance of a facet wrapper class is aggregated for each facet of the component. They capture the invocations received in the component and transfer them to the corresponding facet implementations, which are defined in the executor. The facet wrappers are the place in which the interceptors for managing non-functional features are included.

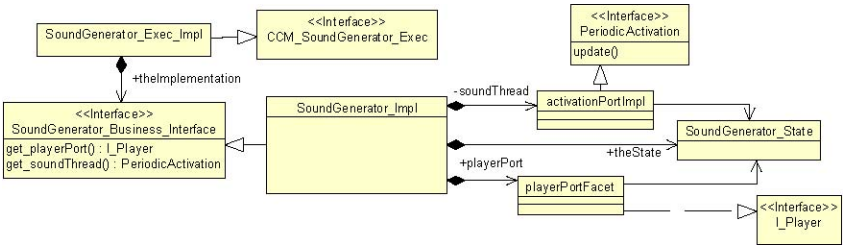


Fig. 7. Component executor structure

- Each activation port defined in the specification of the component represents a thread that is required by the component to implement its functionality. To implement those threads two kinds of Ada task types have been defined. The OneShotActivationTask executes the corresponding run() procedure of the port once, while the PeriodicActivationTask executes the update() procedure of the corresponding port periodically. Both types of task receive as a discriminant during its instantiation, a reference to the data structure that qualify their execution, which includes scheduling parameters, period, and the procedure to call. For each activation port defined in the component, a thread of the corresponding type is declared. They will be activated and terminated by the environment by means of the standard procedures that LwCCM specifies in the CCMObject interface to control the lifecycle of the component.

The rest of generated Ada packages represent the executor of the component. LwCCM defines a set of abstract classes and interfaces which have to be implemented, either automatically or by the user, to develop the executor of the component. This set of root classes and interfaces are grouped in the generated package *{ComponentName}_Exec*. The *{ComponentName}_Exec_Impl* package includes the concrete class for the component implementation which inherits directly from the interfaces defined in the previous package, and therefore, includes dependencies on the technology. As it is shown in Figure 7, this class contains as an aggregated object, the business code implementation of the component, which must implement the *{ComponentName}_Business_Interface* interface. As it has been said before, this interface includes all the methods that the business implementation must implement in order to be managed by the environment in an automatic way. By using this interface together with the aggregation pattern, the environment internals are hidden to the code developer, who is completely free to implement the business code of the component. The only requirement to meet is that the implementation must offer the {ComponentName}_Business_Interface interface, however, relevant aspects that should be included in a correct implementation are:

- For each facet offered by the component, a facet implementation object should be aggregated. In the case of simple components, the class itself can implement the interfaces supported by the facets.
- For each activation port defined in the component, the corresponding implementation object should be aggregated.

- All the implementation elements (facet implementations, activation ports, etc.) operate according to the state of the component, which is unique for each instance. Based on that, the state can be implemented as an independent aggregated class, which can be accessed by the rest of the elements, avoiding cyclic dependencies.

The current available Ada mapping for IDL [13] is based in Ada95, so for the development of the code generation tool, new mappings for some IDL types have been defined in order to get benefit of the new concepts introduced in Ada 2005. The main change concerns the usage of interfaces. The old mapping for the IDL "interface" type led to a complex structure, now, it can be directly mapped to an Ada interface.

4 Application Development Process

The development process of component-based applications, as it is shown in Figure 8, includes the design, configuration, deployment and launching of applications built as assemblies of components previously installed in the development environment.

The *assembler* describes the application as an assembly of component instances, selecting them among those stored in the repository of the design environment, and connecting them according to their requirements. The description is made by means of a Component Assembly Description (.cad file), as it is defined in the D&C specification. For real-time applications, this structural description must be complemented with the description of their workload. The workload of an application is defined as the set of real-time end-to-end flow transactions concurrently executed on it [14]. For each operational mode of the application with real-time requirements to meet (real-time situation), a workload model must be defined. Schedulability analysis tools can then be applied to each real-time situation. The real-time extension of the D&C includes also the definition of special metadata to describe the workload of an application.

In the next phase of the process, the planner takes the assembly description, and designs a deployment planning for the application. This process consists in assigning component instances to nodes, and deciding the mechanisms used for the communication between instances. The result of this stage is the deployment plan (.cdp file), which completely describes the application and the way in which it is planned to be executed.

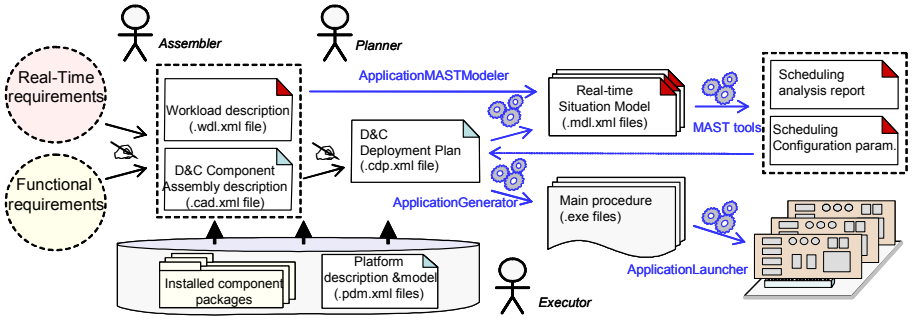


Fig. 8. Component-based application development process in AdaCCM

At this point, a real-time design specific task is included in the process. The deployment plan defines the nature of the communication between component instances, assigning to each connection between component ports, the communication service to use and its corresponding configuration parameters (D&C has been extended to include this kind of information). These data will be used by the deployment tool to generate the corresponding connectors between components, but at this moment it is used to generate the real-time models of those connectors, whose templates should be stored in the repository or must be developed together with the deployment plan. Obviously, the communication services used for the connections must hold predictable behaviour. So, the deployment plan includes all the information required to generate the real time model of the complete application by composition of the real-time models of the components that form it, the platform resources (which must be also stored in the repository) and the connectors used for the interaction between components. This final model is used to calculate the optimal values for the scheduling parameters and to analyse the schedulability of the application under each workload.

Finally, in the last stage of the process, the executor makes use of a launching tool, which performs the following sequence of tasks:

- Using the deployment plan as input, it generates the code of the connectors involved in the application and the code of the main Ada procedures that have to be executed on each node in order to launch the application. These procedures instantiate, connect and configure the components and the connectors according to the information defined in the deployment plan. They also include the configuration of the internal service of the execution environment (*SchedulingAttributeService*) which, together with the interceptors, manage in an automated way the scheduling parameters of the threads during the application execution. The configuration parameters of this service, whose values may be obtained by schedulability analysis or other verification techniques, are also specified in the deployment plan.
- The code of the generated main procedures is compiled and linked with the libraries corresponding to the components, and the code of the connectors involved in the application.
- The resulting executables are moved to and executed in the corresponding nodes.

5 Design Environment and Tools for Components Development

The design of a new component or the deployment of an application are processes which are performed in the development environment. They are to be assisted by tools to guarantee the correctness “by construction” of the generated artifacts. A design environment based in Eclipse has been defined for the Ada-CCM technology. It provides a set of frameworks and services which simplify resource management and tools development. The environment is composed of two key elements: the repository, in which the intermediate and final products are organized, and the tools which carry out the transformations between those products.

The information is organized in three Eclipse projects:

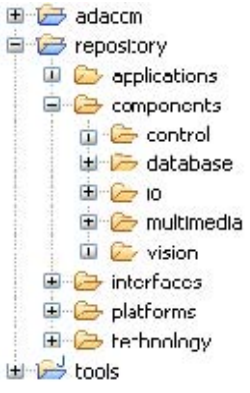


Fig. 9. Repository Structure

- The *repository* project is a general project. It stores and organizes the information relative to the registered elements. It is divided in five main sections: *applications, components, interfaces, platforms* and *technology*. Inside each section, the information is divided in domains which define different namespaces. Each element inside the repository is identified by the chain <*section*>/<*domain*>/<*name*>/<*extension*>.
- The *adaccm* project is an Ada project. It stores the source or compiled Ada code which is required to build a deliverable component or execute an application. Its internal structure corresponds to the structure of Ada packages suitable for compiling and linking with the tools provided by the Ada development plug-in for Eclipse.
- The *tools* project is a Java project. It includes the code of the tools developed for the transformation processes. The currently developed tools are:
  - Tools for importing and exporting deliverable elements: *ComponentImport, ComponentExport*, *InterfaceImport* e *InterfaceExport*.
  - Tools for components development: *ComponentTemplateGenerator* and *ComponentContainerGenerator*.
  - Tools for application management: *ConnectorGenerator*, *ApplicationGenerator* y *ApplicationLauncher*.
  - The tool which generates the final real-time model of an application: *ApplicationMastModeler*.

The Eclipse environment is provided with specialized editors, so it has not been necessary to develop specific tools for editing Ada source code, or the XML files corresponding to the D&C descriptors. For the latter, W3G-Schemas have been defined to facilitate the elaboration of this kind of files.

6 Execution Platform

Applications developed with Ada-CCM can be executed in distributed platforms which provide a run-time library with support for Ada applications. If the applications have hard real-time requirements, all the services of the run-time library and the communication mechanisms must have bounded response times. Likewise, for being able to port the applications to minimal embedded platforms, the run-time must be light, compatible with different targets (including microcontrollers), and it should not require a full file system or support for a hard disk. Figure 10a shows an example of the kind of applications that can be developed with this technology. It is an application whose purpose is to follow the trajectory of a moving object with a camera. The software architecture of the application is shown in Figure 10b. It is composed of six

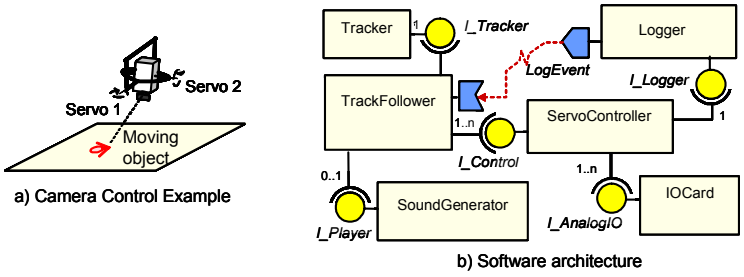


Fig. 10. Application example

components which come from different application domains. The TrackFollower component plays the role of client component, since it is source of business end-to-end flow transactions. It has been specifically designed for this concrete application. The rest of the components have a broad scope of applicability, so they can be reused in different systems. The ServoController component performs the control closed loop of the n servos controllers. In the example, it controls the two degree-of-freedom of the camera orientation. The rest of the components (Tracker, Logger, IOCard and Sound-Generator) are leaves components and their function is to control different resources of the system (vision system, I/O cards, sound generators).

An application like this has been used to probe and experiment with the technology. It has been run on a MaRTE OS (Minimal Real Time Operating System for Embedded Applications) [15] target. MaRTE OS is a real-time kernel which follows the Minimal Real-Time POSIX subset, defined in the IEEE 1003.13 standard. Besides, it offers support for hierarchical scheduling. The target hardware platform is any 386 PC or higher, with at least 512KByte of memory and with a device for booting the application (such as a floppy disk, flash memory, etc.), but not requiring a hard disk. Real-time communication mechanisms currently supported by MaRTE OS include CAN bus, and ethernet with the RT-EP protocol (Real-Time Ethernet Protocol) [8].

7 Real-Time Modelling and Analysis of Component-Based Systems

A real-time model is a timing abstraction that holds all the qualitative and quantitative information needed to predict/evaluate the timing behaviour of an application. It is used by designers to annotate timing requirements in the specification phase, to reason about the prospective architecture during design phases, and to guarantee its schedulability when the system is to be validated.

Software componentization is a structural pattern, which in principle is independent of the real-time design process, but, since it introduces deep changes in the development phases, it interferes with the traditional real-time design. An issue to consider in the component-based design strategies is the coordination between the structural (static) point of view, in which operations are identified as services of instances of

components, and the reactive (dynamic) one, in which the activities (invocation of operations) are serialized in threads.

A real-time component must include metadata that allow a designer to predict its timing behaviour and analyse the schedulability of the applications that make use of it. The modelling methodology must provide two elements:

- Composable and formalized entities to hold the information about the timing and synchronization characteristics of the internal code of the component in a self-contained way and independent of any external elements (other components or platform resources).
- A systematic composition process that allows building the complete real-time model of an application using the models of its constituent parts: business components and platform resources.

An extension to the D&C specification is proposed to add real-time metadata to the descriptions of components, platform resources and applications. These metadata have been distributed according to the phase of the process in which they are required.

The D&C component interface description (.ccd file) includes the information about the temporal behaviour of a component that an application designer needs to decide the utility and the compatibility of the component inside an application. It declares:

- The set of operations with real-time behaviour offered by each facet. Any implementation of the component will include models for these operations. Likewise, for each receptacle of the component, the interface description must declare the operations whose real-time model is required to develop the real-time model of the component itself. Two components will be composable when the server component provides the real-time models of the operations that the client component requires through its receptacles.
- The parameters of the real-time model of the component. The real-time model of a component is a parameterized template which can be configured to describe the component behaviour according to the specific way in which the component is planned to be used in an application. Concrete values must be assigned to each parameter of each component instance declared in an application.
- Components with client role, i.e., components which can trigger business end-toend flow transactions, must include the declaration of the kind of business transactions that they can initiate. The schedulability analysis of an application is performed regarding the workload of the application, and this workload is defined as the set of transactions concurrently executed in the application.

The D&C description of a component implementation (.cid file) must include the elements that describe the real-time behaviour of the internal code of the component:

- The execution time of the operations offered by the component. They are described through their worst, best, and average case values and are related to a reference processor, the one defined as having speed factor equals to 1.
- The synchronization resources that are used during the operations execution. They are necessary since they can cause blocking delays during execution.

- The scheduling entities in which the code execution is organized. These represent the execution capacity of the threads required to the environment.
- The description of the end-to-end flow transactions that are triggered in the component. Each transaction describes the set of activities that are executed in the system in response to external or timed events.

The D&C description of a platform (.tdm files) has been extended to include its real-time model. The platform model defines the models of the software resources (os, mutexes, drivers, etc.) and hardware resources (processors, networks, timers, etc.) that qualify and quantify the available processing capacity, the overheads associated to their management, the policies for the management of their access queues, etc.

The D&C description of the deployment plan that describes an application (.cdp files) incorporates two aspects regarding temporal behaviour:

- Each connection between component ports includes a reference to the corresponding real-time model of the connector used for the communication. The real-time model of a connector includes the information that describes the processes of marshalling and unmarshalling for the invocation parameters and return values, the activities involved in the transmission of messages and the processes of message dispatching.
- A deployment plan is associated with one or more declarations of the application workload. Each workload corresponds to a specific operation mode of the application that have timing requirements to meet, and for each of which schedulability analysis can be applied to verify that the requirements are met. Special metadata have been defined to declare the workload associated to an application.

Once an application is defined through a deployment plan, the planner can build its real-time model by means of the *ApplicationMastModeler* tool. As it is shown in Figure 8, this tool takes the information provided by the deployment plan, the metadata associated to the component descriptions and the metadata associated to the platform descriptions, and generates the final real-time model of the application. The real-time modelling and analysis methodology applied in Ada-CCM is MAST [14]. Specifically, an extension to MAST which incorporates the composability properties needed to generate the real-time model of a complex system by the composition of the individual real-time models of the software and hardware components that forms it [16].

MAST conceives the real-time model of an application as a description of its reactive behaviour. An application is modelled as a set of end-to-end flow transactions ("transactions" in MAST), which are sequences of activities that are triggered in response to external or timed events. A transaction is described by its set of activities, the generation pattern of the triggering events, and the timing requirements that must be met. The activities in different transactions only interact by sharing the processing-resources and the mutually exclusive passive resources.

The MAST environment includes several tools for real-time applications design:

- Schedulability analysis tools: They can be applied to both monoprocessor (RM Analysis y EDF Monoprocessor Analysis) and distributed systems (Holistic Analysis y Offset Based Analysis). They allow to certify that, in the worst case, the activities scheduled in the application meet their real-time requirements.

- Automatic priority assignment tools: Their usage is required, specially in distributed systems, when the amount of priorities or scheduling parameters to adjust makes the calculation process too complex to be developed without tool assistance. They can be applied to monoprocessor (Rate Monotonic and Deadline Monotonic) and distributed (Simulated Annealing and HOPA) platforms.
- Slack calculation tools: These tools calculate the percentage by which the execution time of the operations may be increased while keeping the system schedulable, or must be decreased to get the system schedulable.

8 Conclusions and Future Work

The proposed technology enables the development of hard real-time embedded component-based applications, whose temporal behaviour can be modelled and analysed by schedulability analysis tools. This is achieved by the combination and enhancement of well known technologies. (i) The usage of Ada makes the technology particularly suitable for applications that run in embedded nodes with limited resources, and interconnected with real-time communication networks. (ii) Some extensions introduced in the LwCCM container/component framework, together with the Ada's native support for concurrency and synchronization, provide the capacity of developing the code of the components with predictable temporal behaviour. (iii) The technology follows the D&C standard for the specification of components, platforms and applications. A real-time extension of D&C has been proposed to incorporate metadata describing the temporal behaviour of components and platforms. These metadata is used to analyse the schedulability of the application during the development process. The concepts and semantics added with the real-time extensions allow the assembler or the planner to design the real-time aspects of the application without knowing the modelling methodology used by the analysis tools. The developer formulates the real-time models of the components following a concrete modelling and analysis methodology, which in the case of Ada-CCM is MAST.

Relevant future work concerns reducing the cost of real-time components design. They have to be designed so that their execution holds bounded timing behaviour, this behaviour should be modelled in detail, and all the execution times used in the model must be consciously evaluated. This latter task is currently very costly, though recent advances in techniques and tools promise to help reducing this cost in the future.

References

[1] Ommering, R., Linden, F., Kramer, J.: The koala component model for con-sumer electronics software. In: IEEE Computer, pp. 78–85. IEEE, Los Alamitos (2000)

[2] Lundbäck, K.-L., Lundbäck, J., Lindberg, M.: Component based development of dependable real-time applications Arcticus Systems, `http://www.arcticus-systems.com`

[3] Bondarev, E., de With, P., Chaudron, M.: Predicting Real-Time Properties of Component-Based Applications. In: Proc. of 10th RTCSA Conference, Goteborg (August 2004)

[4] Åkerholm, M., et al.: The SAVE approach to component-based development of vehicular systems. Journal of Systems and Software 80(5) (May 2007)
[5] OMG: Lightweight Corba Component Model, ptc/03-11-03 (November 2003)
[6] Taft, T., et al. (eds.): Ada 2005 Reference Manual. Int. Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1. LNCS, pp. 43–48. Springer, Heidelberg (2006)
[7] OMG: Deployment and Configuration of Component-Based Distributed Applications Specification, version 4.0, Formal/06-04-02 (April 2006)
[8] Martínez, J.M., González, M.: RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. In: Vardanega, T., Wellings, A.J. (eds.) Ada-Europe 2005. LNCS, vol. 3555. Springer, Heidelberg (2005)
[9] Pautet, L., Tardieu, S.: GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In: Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, USA (March 2000)
[10] OMG: Quality of Service for CORBA Components, ptc/06-04-05 (April 2006)
[11] OMG: Real-Time CORBA Specification, v1.2 formal/05-01-04. Enero (2005)
[12] Gutiérrez García, J.J., González Harbour, M.: Prioritizing Remote Procedure Calls in Ada Distributed Systems. In: Proc. of the 9th Intl. Real-Time Ada Workshop, ACM Ada Letters, XIX, Junio 1999, vol. 2, pp. 67–72 (1999)
[13] OMG: Ada Language Mapping Specification - Version 1.2 (October 2001)
[14] González Harbour, M., Gutiérrez, J.J., Palencia, J.C., Drake, J.M.: MAST: Modeling and Analysis Suite for Real-Time Applications. In: Proc. of the Euromicro Conference on Real-Time Systems (June 2001), `http://mast.unican.es/`
[15] Aldea, M., González, M.: MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In: Strohmeier, A., Craeynest, D. (eds.) Ada-Europe 2001. LNCS, vol. 2043. Springer, Heidelberg (2001), `http://marte.unican.es/`
[16] López, P., Drake, J.M., Medina, J.L.: Real-Time Modelling of Distributed Component-Based Applications. In: Proc. of 32nd Euromicro Conference on Software Engineering and Advanced Applications, Croatia (August 2006)

Author Index