Proceedings of the Fifteenth International Workshop on Component-Oriented Programming (WCOP) 2010

22nd June 2010, Praha, Czech Republic

Barbora Bühnová, Ralf Reussner,
Clemens Szyperski, Wolfgang Weck (Eds.)

2010

Fakultät für **Informatik**

Workshop on Component-Oriented Programming (WCOP) 2010

Preface

The Workshop on Component-Oriented Programming (WCOP) was one of the driving forces in the nineties that brought component orientation into broader consciousness of the software development community. Definitions of basic terminology were discussed and concepts were clarified. In the 2000 decade, the component idea got established in dedicated conferences (foremost CBSE, itself a former ICSE workshop) and also influenced the software architecture community strongly. During these times, WCOP evolved to a workshop for young researchers to present new ideas and to collect feedback from established members of the community. Right from the beginning, WCOP papers received high attention and often formed the premier way for researchers informing themselves about new developments in our community. This is, for example, well demonstrated by the high citation rate of WCOP papers.

Now, in 2010, at the beginning of a new decade, WCOP institutionalized its role as a forum for young researchers in our community. The success of WCOP as the doctoral symposium of the CompArch federated conference shows not only that this role of WCOP is appropriate, but much more it also shows that the component idea is still vibrant and attracts young researchers internationally. We welcome Assist.-Prof. Dr. Barbora Bühnová from Masaryk University in Brno as an additional organizer, next to Clemens Szyperski, Wolfgang Weck, and Ralf Reussner. As a new incentive, WCOP 2010 awards the CompArch Young Investigator Award, which is given to the work of a young researcher in our community to award specifically promising work of expected high importance. The award is sponsored by the steering commitee of CompArch which also forms the programme commitee of WCOP. This year, the first CompArch young investigator award was given to Zoya Durdik from FZI, Germany, for her paper on the integration of architectural modeling into agile development methods. This year, after a rigorous review process, we accepted ten papers. The reader will notice, that all of these papers show original ideas with well-started research, highlighting topics showing the role that components can play for the future of software development.

We thank the programme committee for their valuable help in the paper selection and Erik Burger for preparing the proceedings of this year's WCOP. We are grateful to the CompArch organizers, especially to Frantisek Plasil and Petr Hnětynka, for taking care of all the local organization and for accommodating our special requests.

We wish the participants and presenters of WCOP 2010 many insights, useful connections, and further successes in our community.

Brno, Karlsruhe, Redmond and Zurich

Barbora Bühnová, Ralf Reussner, Clemens Szyperski, Wolfgang Weck

Workshop Co-organizers

Barbora Bühnová
Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno, Czech Republic
E-mail: buhnova@fi.muni.cz
Web: http://www.fi.muni.cz/~buhnova/

Ralf Reussner
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
E-mail: reussner@kit.edu
Web: http://sdq.ipd.kit.edu/

Clemens Szyperski
Microsoft
One Microsoft Way
Redmond, WA 98053, USA
E-mail: clemens.szyperski@microsoft.com
Web: http://research.microsoft.com/~cszypers/

Wolfgang Weck
Independent Software Architect
Böszelgstrasse 13
8600 Dübendorf, Switzerland
E-mail: mail@wolfgang-weck.ch
Web: http://www.wolfgang-weck.ch

WCOP Website

http://wcop.ipd.kit.edu/

Contents

Formal Model Assisted Code Generation for Critical Embedded Systems

Etienne Borde

Mälardalen Real-Time and Research Center

Mälardalen University

P.O. Box 883, SE-72 123 Västerås, Sweden

Email: etienne.borde@mdh.se

Abstract—In order to cope with the growing complexity of embedded software while shortening time-to-market, component-based software engineering offers the possibility to reuse existing functionalities while automating (i) the analysis of the system under design and (ii) the production of the final software. However, it is difficult to ensure that the produced software actually conforms to the hypothesis used for analysis purpose. Indeed, model based verification relies on a very different semantics from the one used in the software implementation.

In this paper, we propose a design approach that consists of automating the production of a detailed description of the software application, as an intermediate step towards its concrete production. As a result, the analysis of the system can be made at both levels, considering its abstract modelling and the description of its concrete realisation.

I. INTRODUCTION

More and more products in our everyday life take advantage of the miniaturization of electronics to provide functionalities that are controlled by a software embedded system. In order to cope with an increasingly competitive market, these functionalities are more and more sophisticated, and thus increasingly complex. For the same reason, the design of such embedded systems must cope with even shorter time-to-market. Computer systems that are embedded in cars, air planes, military systems etc., are called **critical** embedded systems because a failure of such a system may have catastrophic consequences. In this paper, we propose a new design approach that we intend to implement in order to cope with the complexity of developing critical embedded systems while increasing the confidence one can have in the final result. To achieve these goals, we propose a solution that:

1) eases the analysis of a software system at different stages of its design process;
2) enables reusability of already existing design artefacts;
3) automates the different steps of the design process, until completion.

Analysis of a system under design increases confidence one can have in the final product. The reusability of existing design artefacts accelerates the time-to-market of a product, even more in product-lines as it is often the case in industry. Finally, automation of the design process is very important since the increasing complexity of embedded software induces integration of a huge set of information that cannot be handled manually.

If each of these points provides solutions to design phases of embedded systems, their integration in a consistent design process still raises an important problem: when it comes to analysis of critical embedded systems, formal methods enable insurance that an abstract model of the system under design respects some safety properties. These formal techniques rely on mathematical constructions that are used to abstract the behaviour of a system. On the other hand, the final system is implemented with a programming language whose semantic (*i.e.* meaning when interpreted by a machine or a software tool) is very different from the one used for analysis purpose. This poses the following problem: how to ensure that the implementation of the system actually respects the hypotheses that were used for verification purposes?

In this paper, we tackle this problem by proposing a new approach to automate the implementation of the formal semantic: instead of directly producing this implementation, we propose to use an intermediate level of representation of the software (*i.e.* between the formal level and the implementation level), whose semantic is close to the implementation semantics but enables to automate its analysis.

The remainder of this paper is organised as follows: section II presents the motivations of this work describing the two main challenges it aims at tackling. Section III constitutes a brief state of the art of research works that tackled those challenges. In section IV, we describe in details the approach for which we intend to implement a dedicated component framework. Finally, section V concludes this paper and presents the perspectives of this work.

II. MOTIVATIONS

The main goal of the approach presented in this paper is to bridge the gap between the formal semantics required by the analysis techniques of embedded systems and the implementation of this type of systems. Bridging this gap enables insurance that the verified model correctly represents the implemented system, and vice versa. Let us try to present hereafter the different challenges raised by this objective.

A. Formal semantics versus implementation

Different techniques exist to ensure that a model meets a given set of safety properties (properties ensuring that the system always provides the functionalities it was designed for).

Among those techniques, model-checking is the easiest to use in an automated design process. There exists two different approaches in order to perform model-checking.

One consists of modelling a system that has been implemented without any simplifying hypothesis on its execution semantics. In this case, the formal model must represent the implemented system as closely as possible. *The implementation actually leads the design process.*

The other approach consists of restricting the execution semantics of the software that constitutes the system thanks to hypothesis that will simplify the verification process. One can for instance suppose that all the actions of a system are atomic so that timing properties is no longer an issue in the verification process. In this case, *verification leads the design process.* If using the former technique, the system is much simpler to implement (the implementation does not have to ensure a formal semantic) but more difficult to verify: model checking techniques suffer from the well known scalability problem, where this problem increases correspondingly to the complexity of the model. On the other hand, the simplifying hypotheses of the later approach are more difficult to implement, but ease the analysis [1].

In this paper, we choose the later approach, since it is more suitable for safety-critical systems.

Challenge 1. *Automate the implementation of a formally defined execution semantic.*

B. Reusability versus automatic production

Reusability of design artefacts is an important industrial requirement when it comes to manage the evolution of an existing product. On the other hand, embedded systems also have to cope with very heterogeneous requirements, like for instance: memory limitations, temporal determinism, power consumption and properties specific to a given system. These requirements may also lead to the selection of a particular execution platform (hardware, operating system and communication media), which can decrease the reusability of a piece of software. Indeed, embedded software most of the time include some platform dependent code: when changing the platform, it is often necessary to use a different compiler. To deal with this heterogeneity of requirements and execution platforms, it is possible to generate the software application, but reducing the reusability of the corresponding code.

Concluding the above, heterogeneity of embedded systems' requirements makes the reusability of existing software difficult [2].

Challenge 2. *Enable reusability of design artefacts while answering to the heterogeneity of requirements of embedded systems.*

III. SURVEY OF THE FIELD

a) Existing solutions to challenge 1: The problem of implementing formally defined semantics has been intensively studied over the last 20 years, and has been marked by a series of success stories (*e.g.* ESTEREL [3], LUSTRE [4], BIP [5],

CSP [6]). These languages rely on simplifying hypothesis that ease their mathematical analysis: for instance, synchronous languages (ESTEREL and LUSTRE) assume that every action of the software application is made instantaneously when receiving a triggering event. Of course, the implementation cannot strictly respect this hypothesis. Still, the compilers of these languages ensure that for any input pattern given to the system, its outputs will be the same as the one that would have been produced according to the formal semantics [7]. In the scope of our proposal, the main limitation of those solutions is that they do not address the issue of the reusability of existing design artefacts (see challenge 2).

b) Existing solutions to challenge 2: When it comes to reusability of design artefacts, component based software engineering is a well established solution [8], which is already used in the design of industrial non-critical software products (Koala [9], THINK [10], MyCCM [11]). As far as we know, only a few projects have investigated the usage of component-based software engineering in the domain of critical embedded systems (MyCCM-HI [12], ProCom [13]). ProCom defines a formal semantic that eases the formal analysis of a system's behaviour. The definition of this semantic consists of assuming that *the communications (control and data transfer) between collocated software components are atomic.* This actually relaxes the synchronous semantics: the execution time of the components is not null. On the other hand, MyCCM-HI does not define any formal semantics, but provides transformation rules to formal methods in order to ease the analysis. ProCom is a component based model lead by verification, while MyCCM-HI is lead by implementation. This is the reason why we propose, in the approach presented here, to use the ProCom component model.

Among the research works that aim at coping with the heterogeneity of requirements existing in the domain of embedded systems, architecture description languages and modelling languages constitute a promising solution. In the domain of real-time and embedded systems, UML and its profile MARTE [14], as well as AADL [15] offer (among others) the possibility to model precisely the software architecture of an embedded system. The architecture description language AADL has particularly retained our attention. The reasons of this interest are multiple:

1) AADL is a **standard** dedicated to the design and analysis of real-time and embedded systems;
2) several research programs lead to the creation of **analysis and code generation tools** using this language (Cheddar [16], Ocarina [17], ADAPT [18]);
3) its usage is complementary with the usage of ProCom: they both address different type of design and analysis requirements of a development process: ProCom targets the encapsulation of software functionalities in order to ease their reuse while AADL enables to describe the allocation of those functionalities over the execution resources.

As to summarize, AADL meets most of the qualities of an ideal ADL [19].

In next section, we present in more details the new approach we propose to answer both challenges 1 and 2.

IV. Technical Approach

The approach we propose in this paper consists of automating the production of platform dependent code while reusing the basic functionalities of a system. Platform dependent code is thus generated while the functional code is reused (regardless it has been handily written or generated by another tool). These basic functionalities will be encapsulated as software components in order to ease their reuse in different contexts. The corresponding component model will provide a formal semantics in order to ease system level verifications (*i.e.* the respect of the system level requirements). As a first step towards ensuring this formal semantic is actually respected in the implementation, we propose to model this implementation in an intermediate representation that must be close to the implementation semantics, but must also be analysable. The corresponding level of abstraction consists of representing the execution and communication semantics of the software: we describe how functionalities are triggered and how they communicate in the final implementation. Shared variables, periodic and sporadic tasks, as well as communication buffers have to be precisely represented in this model.

To begin with the presentation of this approach, we first describe ProCom, the component model we have selected for this approach. We then illustrate the gap existing between the semantic used for the analysis and for the implementation. To illustrate this difference, we rely on the example of ProCom components interaction. This will finally lead us to the presentation of our integrated approach.

A. ProCom, the component model

1) General presentation: The ProCom component model has been specifically developed to address the specificities of designing distributed real-time and embedded systems. To address the different concerns that exist on different levels of the design of such systems, ProCom consists of two distinct, but related, layers. At the upper layer, called ProSys, the system is modelled as a number of active and concurrent subsystems, communicating by message passing. The lower layer, ProSave, addresses the internal design of a subsystem down to primitive functional components implemented by code.

In ProSys, a system is modelled as a collection of concurrent subsystems that communicate by asynchronous message passing.

Contrasting this, the lower lever, ProSave, consists of passive units, and is based on a 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. Data ports always appear in a group together with a single trigger port, and the data ports in the same group are read or written together in a single atomic

action. This is the main hypothesis of the ProCom component model in order to ease the formal analysis of the system under design.

Both layers are hierarchical, meaning that subsystems as well as components can be nested. The way in which the two layers are linked together is that a primitive ProSys subsystem (i.e., one that is not composed of other subsystems) can be further decomposed into ProSave components. At the bottom of the hierarchy, the behavior of a primitive ProSave component is implemented as a C function.

2) Specificities of ProCom: The main characteristic of the ProCom component model is that it exhibits, through the structure of the components, the execution model of the software architecture. To achieve this objective, the ProCom component model imposes restrictions on the behavior of its constructs that we explain hereafter:

The functionality of a ProSave component is captured by a set of services. The services of a component are triggered individually and can execute concurrently, while sharing only data. A service consists of one input port group and zero or more output port groups, and each port group consists of one trigger port and a number of data ports. An input port group may only be accessed at the very start of each invocation, and the service may produce parts of the output at different points in time. The input ports are read in one atomic step, and then the service performs internal computations and writes at its output port groups. The data and triggering of an output group of a service are always produced at the same time. Before the service returns to idle, each of the associated output port groups must have been activated exactly once. This restriction serves for tight read-execute-write behavior of a service.

In order to implement complex functionalities, ProCom components must be connected. This can be done by simple *connections* that transfer data or control, and *connectors* providing more elaborate manipulation of the data and control flow.

Finally, ProCom gives the possibility to model the internal structure of ProSys components thanks to connected ProSave components, connectors, and *clocks*. Ports primitive ProSys subsystems, dedicated to message passing, are then connected to data and trigger ports of ProSave components. Besides, clocks serve for generating periodic trigger and activate component assemblies periodically.

Figure 1 shows the model of a primitive ProSys subsystem composed of ProSave components, connectors and clocks. As one can see on the top right part of the figure, message ports of ProSys components can be connected to trigger and data ports of ProSave components. This actually means that data contained in messages received on the system ports are transmitted to the connected subcomponents. Connectors are represented in this figure as n-to-m connections. For instance, a *control fork* connector is positioned between the clock activated at 50 Hertz and the components triggered by this clock (components computing the actual direction and the desired direction). This connector states that both computations have to be executed in parallel.
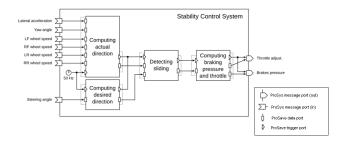
Fig. 1. Stability Control System

B. Deployment of ProCom components

Modelling the deployment of ProCom components just consists of representing the allocation of ProSys components onto *virtual nodes* that are latter on mapped onto the concrete hardware platform. This information is then used to realize the deployment of components. Since ProCom is dedicated to the design of critical distributed real-time and embedded systems, components are deployed statically: the definition and initialization of data structures corresponding to components, tasks and interactions, is made at compile time (or during the very beginning of the system initialization if necessary). As a consequence of this choice, the definition of these data structures, as well as their initialization, is synthesised into the code of the system implementation.

This synthesis process mainly consists of the components-to-resource allocation. In the scope of distributed and real-time embedded systems, this allocation consists of mapping:

- interactions of ProSave components to shared variables and call sequences;
- ProSave components activation (clocks and communication channels) to real-time tasks;
- interactions of ProSys components to the physical communication media.

As presented in the introduction of this paper, synthesising the system code while ensuring its conformance with the model used for its analysis is a difficult problem. In the reminder of this section, we present our current works that aims at answering this issue in the scope of ProSave components.

The main hypothesis of data transfer between ProSave components is atomicity. Of course, this hypothesis cannot be ensured in a multi-threaded implementation since it requires data copying and locking. However, the implementation must ensure that the data transfer pattern (emission and reception) is the same as the one that have been considered for analysis (*i.e.* with the atomicity hypothesis). To ensure this, we propose to rely on a three steps data transfer with a double buffer implementation: the output port of a component is deployed as a set of two buffers, one that can be updated during the execution of the component, and one that contains the last up-to-date value.

1) The first step of a data transfer involves the writer component: during its execution, it can write on the accessible buffer.

2) The second step begins when the output port group (containing the considered data port) is triggered. At this moment, the buffer roles are switched: the accessible buffer becomes the last up-to-date buffer and vice versa.

3) The last step is executed when the port group containing the reader data port is triggered. Then, the last up-to-date value is copied in the internal structure of the reader service.

In order to ensure atomicity and data consistency, the second and third steps must never be concurrently and simultaneously accessed. This double buffer solution has been preferred to a single buffer implementation since it reduces time spent in the critical section, thus getting closer to the atomic hypothesis. Besides, it is similar to the solution used in the scope of synchronous programming, for which a conformance proof has been provided [1]. However, we did not yet prove that this implementation conforms to the hypothesis of the ProCom model analysis [13]. As one can easily understand, the conformance between this implementation and the analysis semantic is very difficult to ensure.

In the remainder of this paper, we propose a solution to improve the confidence one can have in the conformance between the analysed model and its implementation.

C. Describing the execution model with AADL

In order to bridge the gap between the analysis and the implementation semantics, we propose to produce an AADL description representing the allocation of ProCom functionalities onto the execution resources (shared variables, call sequences, real-time tasks, and communication media).

AADL [15] is an architecture description language (standardized by the SAE [1]) dedicated to the design and analysis of distributed real-time and embedded systems. AADL relies on a notion of component which falls in four different domains: System, Abstract, Software and Hardware. The definition of the system and abstract domains are very generic, so that the associated semantic is imprecise (which is not really surprising considering the purpose of using such components). On the other hand, software and hardware components are **"concrete components"** that come with a precise semantic. Besides, AADL aims at representing precisely the allocation of software components onto the execution platform. In the remainder of this section, we present the different component categories belonging to AADL software and hardware components; we describe some of the possible SW/HW mappings provided by the language; finally we illustrate the granularity level AADL enables to reach.

a) Software components: AADL defines six component categories among software components:

- **Data** components can represent data types, classes, objects and shared variables.
- **Subprogram** components represent sequentially executed source text (with its parameters).
- **Subprogram groups** represent subprogram libraries.

- **Thread** components represent schedulable unit that can execute concurrently on the execution platform.
- **Thread group** components represent logically grouped threads.
- **Process** components represent a virtual address spaces, available to store the state of other software components.

b) Hardware components: AADL defines mainly six component categories among hardware components:

- **Processors** components represent the set of hardware and software platform elements that is responsible for executing and scheduling threads and virtual processors.
- **Virtual processors** represent logical resources capable of scheduling and executing threads or other virtual processors.
- **Memory** components represent the hardware entities that enable to store code and data binaries.
- **Buses** represent communication media dedicated to the transfer of control and data between memories, processors, and devices.
- **Devices** represent hardware entities that interface with the execution environment.

c) Software to hardware allocation: The allocation of software components onto hardware components is mainly modelled into system components, by the means of "bindings": depending on its type, a software subcomponent contained in a system component can be bound to one of the hardware components of this system. For instance, a thread component can be bound to a process components, a data to a memory, a connection between software components to a bus, etc...

d) AADL, precision of the semantic: The above presentation shows how concrete is the AADL semantic. The AADL semantic is also very precise: for instance, when the communication between two software components requires the usage of a buffer, AADL enables to specify the size of the buffer, as well as the expected behaviour of a full buffer: when the buffer is full and a data has to be written, this data can overwrite the last or the first data contained in the buffer.

The above presentation of ProCom and AADL shows that these two modelling techniques are complementary. ProCom defines software components focusing on their reuse and the formalisation of their semantic. AADL focuses on the concrete software and hardware description, as well as the mapping between them. As a consequence, mapping ProCom components onto AADL components gives the opportunity to model precisely the allocation of ProCom components onto the execution platform. Thus, the design and implementation of a system, based on ProCom components, can be assisted by formal verification at both level: considering the formal semantic of ProCom components, and analysing the AADL description of their implementation. The remainder of this paper presents the corresponding approach in more details.

D. Overview of the approach

Figure 2 summarizes this approach. As one can see on this figure, initial modelling will be performed with the ProCom component model, and transformed into an AADL model that describes the implementation of the ProCom semantics. Our approach enables to ensure the correctness of the model transformation process thanks to two strong principles:

1) the implementation of the ProCom semantics is correct-by-construction (the implementation enforces the respect of the formal semantics hypothesis, the same way as in [1]);
2) the verification realized at the formal semantic level can also be performed using the description of the implementation with AADL (which ensures that the implementation enforces the respect of those properties).

The non-functional properties of the ProCom components are then integrated into the generated AADL model (see top left part of the figure) in order to extend analysis capabilities. The model transformation from the formal semantics level to the description of the execution semantics consists of representing in AADL (target model) how the ProCom (source model) semantics is implemented. The source model consists of a set of clocks and events that trigger components, which in turn trigger the components they are connected to. On the other hand, the description of the execution semantics consists of a set of tasks that execute operations, require or provide access to data, use communication buffer, etc...

The formal semantic and the execution semantic are two complementary abstraction levels: by representing functionalities encapsulated by components with a formal semantics, ProCom enables the reuse of existing design artefacts and eases the analysis of the system requirements (see top right part of the figure, the analysis can be performed on components of components assemblies); by representing the corresponding implementation, AADL eases the generation of platform dependent code and enables the real-time properties analysis (for instance that all the tasks meet their deadlines, see middle right part of the figure).

To conclude this subsection, let us explain how far this approach answer the two challenges described in section II.

1) Our approach answers challenge 1 firstly by using implementation patterns that preserve the formal semantic hypotheses [1], and secondly by enabling to check safety properties twice: once first thanks to the formal specification and then using the description of its implementation (described in AADL). Besides, using the intermediate level of representation (in AADL here) enables to extend the scope of analysis of safety properties.
2) Our approach answers to challenge 2 firstly by offering the possibility to reuse existing design artefacts, and secondly by automating the assembly process of these different entities.

Of course, our approach still misses the proof that the different steps of production are correct, which is actually part of the perspectives of this work. We illustrate in next section our preliminary work on mapping ProCom components onto AADL.
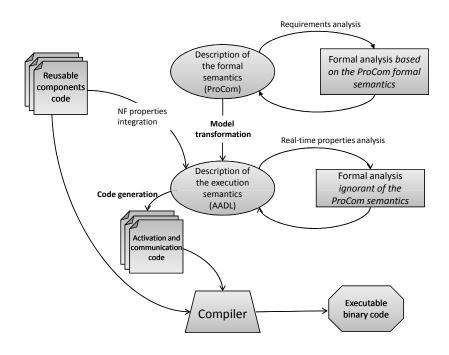
Fig. 2.   Overview of the technical approach

E. Mapping ProCom onto AADL

Mapping ProCom components to AADL components mainly consists of mapping the implementation of ProCom components, their activation, and their interactions, onto AADL software components. Indeed, the virtual nodes modelling in ProCom can easily be transformed into a set of AADL virtual processors and processes. Following the same idea, the modelling of hardware in ProCom is easy to transform into AADL: each modelling entity in ProCom dedicated to the execution platform description has an equivalent entity in AADL.

When it comes to software components, the main transformation step consists in deciding of the components to task allocation. After having automatically decided of this allocation [20], ProCom clocks, components, connectors and connections are mapped into a set of AADL tasks, subprograms, data, ports, and connections. Figure 3 illustrates the resulting mapping of the system described in figure 1. On this figure, one can see that the enclosing ProCom system is mapped into an equivalent AADL system. Message ports of the ProCom subsystem are mapped onto AADL data ports. In case the control flow was transferred with data, the mapping would have produced a system with AADL event data ports (as it is the case for the output ports of the system). The ProCom clock (50 Hz) has been transformed into an AADL periodic task (20 Ms), and the components activated by this clock are described thanks to AADL subprogram calls. The data ports of the components are, in this case, represented thanks to connected subprograms parameters. The reason of this mapping is that, in this example, the data transfer is realised between

Fig. 3.   Stability Control System Mapped in AADL

components executed in the same task. When the data transfer involves different tasks, then data ports have to be mapped into AADL data components, and the accessors implementation have to be mapped into AADL subprograms with a shared access to the data. Such a description corresponds to the double buffer implementation described in section IV-B.

F. Ongoing research work

The results of the approach presented in this paper will be twofold (theoretical and experimental):

1) the theoretical part will tackle scientific issues related to our objectives. On the one hand, accelerating the time-to-market requires automating the design process, and/or reusing existing design artefacts. On the other hand, it is necessary to ensure that the automation of the software

production, or the reuse of existing functionalities, does not introduce undesired characteristics in the final product. We thus need to provide answers to this question: how to make automation and reuse reliable?

2) As one can see in the Figure 2, the experimental part will provide a concrete solution to this problem: a component framework that automates the production of the software application, enables the reuse of existing entities, and improves the analysis of the system under design.

V. CONCLUSION AND PERSPECTIVES

The expected result of this approach is a component-based framework led by a formal semantics, a requirement of determinism, and the integration of an architecture description language dedicated to the design and analysis of real-time software. As far as we know, this will be the first model driven approach that merges those three key aspects:

- component-based software engineering for reusability;
- formal model driven engineering;
- modelling languages dedicated to the design and analysis of distributed real-time systems.

This project opens several perspectives that will have to be addressed in future works. Out of these perspectives, here are the two ones we think should be addressed first:

1) in order to reinforce our solution, it is important to deal with the demonstration that the implementation patterns we use for compiling the ProCom semantics, as well as the model transformation algorithms we propose, produce an execution semantic equivalent to the ProCom semantic;

2) to go further in the answer to real-time and embedded systems design, we should consider the dynamic reconfiguration and fault-tolerance issues in the scope of ProCom.

REFERENCES

[1] Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis, "Semantics-preserving multitask implementation of synchronous programs", *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1–40, 2008.

[2] Michael Mrva, "Reuse factors in embedded systems design", *Computer*, vol. 30, pp. 93–95, 1997.

[3] Gérard Berry, "The foundations of esterel", pp. 425–454, 2000.

[4] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language lustre", *IEEE Trans. Softw. Eng.*, vol. 18, no. 9, pp. 785–793, 1992.

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis, "Modeling heterogeneous real-time components in bip", in *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, Washington, DC, USA, 2006, pp. 3–12, IEEE Computer Society.

[6] C. A. R. Hoare, "Communicating sequential processes", *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[7] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Venier, and Jacques Pulou, "Efficient compilation of esterel for real-time embedded systems", in *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, New York, NY, USA, 2000, pp. 2–8, ACM.

[8] Ivica Crnkovic, *Building Reliable Component-Based Software Systems*, Artech House, Inc., Norwood, MA, USA, 2002.

[9] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee, "The koala component model for consumer electronics software", *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[10] Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polakovic, Marc Poulhiès, Jacques Pulou, Stéphane Seyvoz, Julien Tous, and Thomas Watteyne, "Think: View-based support of non-functional properties in embedded systems", in *ICESS '09: Proceedings of the 2009 International Conference on Embedded Software and Systems*, Washington, DC, USA, 2009, pp. 147–156, IEEE Computer Society.

[11] Etienne Borde, Grégory Haïk, , Virginie Watine, and Laurent Pautet, "Really hard time developing hard real-time", in *2nd National workshop on Control Architecture of Robots*, june 2007.

[12] Etienne Borde, Grégory Haïk, and Laurent Pautet, "Mode-based reconfiguration of critical software component architectures", in *DATE*, 2009, pp. 1160–1165.

[13] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson, "Formal semantics of the procom real-time component model", in *SEAA '09: Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, Washington, DC, USA, 2009, pp. 478–485, IEEE Computer Society.

[14] Object Management Group (OMG), "The promarte consortium. uml profile for marte, beta 2, omg document number: ptc/08-06-08", Tech. Rep., OMG (Object Management Group), 2008.

[15] SAE, "Architecture analysis and design language (aadl)", Tech. Rep., The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, November 2004.

[16] Frank Singhoff et al., "Aadl resource requirements analysis with cheddar", in *SAE AADL Working Group meeting, Paris, October 18-21, 2005*, 2005.

[17] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications", in *Reliable Software Technologies'09 - Ada Europe*, Brest, France, jun 2009.

[18] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaniche, "The adapt tool: From aadl architectural models to stochastic petri nets through model transformation", *Seventh European Dependable Computing Conference*, vol. 0, pp. 85–90, 2008.

[19] Nenad Medvidovic and Richard N. Taylor, "A classification and comparison framework for software architecture description languages", *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[20] Johan Fredriksson, Kristian Sandstrm, Mikael kerholm, and Kristian S, "Optimizing resource usage in component-based real-time systems", in *In: Proceeding of CBSE8 International Symposium on Component-based Software Engineering*, 2005, p. 0836.

Towards Formal Certification
of Software Components

Erik Burger

Institute for Program Structures and Data Organization, Faculty of Informatics

Karlsruhe Institute of Technology, Germany

E-mail: burger@kit.edu

Abstract—**Software certification as it is practised today guarantees that certain standards are met in the process of software development. However, this does not make any statements about the actual quality of implemented code. We propose an approach to certify the non-functional properties of component-based software which is based on a formal refinement calculus, using the performance abstractions of the Palladio Component Model. The certification process guarantees the conformance of a component implementation to its specification regarding performance properties, without having to expose the source code of the product to a certification authority. Instead, the provable refinement of an abstract performance specification to the performance description of the implementation, together with evidence that the performance description reflects the properties of the component implementation, yields the certification seal.**

The refinement steps are described as Prolog rules so that the validity of refinement between two performance descriptions can be checked automatically.

Index Terms—**Components, Certification, Performance Engineering.**

I. INTRODUCTION

The term *software certification* usually denotes a process during which an external authority attests certain properties of the software development process in a company. These properties concern the development process or the qualification of the people envolved with it. Despite the fact that a neutral third party is involved, this kind of certification does not guarantee the quality of an actual software product, but, as Voas [1] puts it, rather "make[s] software publishers take oaths concerning which development standards and processes they will use." However, this type of certification is the most common, in contrast to product certification [2].

The process of *software verification*, on the other hand, proves the functional correctness of a piece of software, based on formal verification methods. However, software verification does not cover non-functional requirements such as performance or safety. The advantage of formal verification over a testing and simulation based approach is the exhaustive coverage of all system states, which cannot be reached with classical testing [3].

In our proposed approach, software certification is a formal process of product certification, concerning the non-functional properties of software. This process includes the definition of

Service Effect Specifications (SEFF) [4] for the specification documents as well as for the implementation. The software issuer can then certify the conformance of the piece of software to the specification document. In combination with techniques that certify the implementation against its SEFF description, the quality of a software product can be certified through the complete chain of software development, from the specification document to the actual application. This is especially important for distributed software development processes, where the components of a product are developed by external software suppliers in a globalised market. In order to guarantee the non-functional properties of a piece of software, the quality of the externally developed parts has to be determined. Since software suppliers are often not willing to have their source code inspected by a certification authority or the customer, the definition of non-functional properties of the software provides a level of abstraction that hides implementation details of the product, but can still be used to gain a certificate about the quality of the software.

By using formal methods for the certification of software, the actual process may also be performed by the issuer of the piece of software himself with the support of semi-automatic certification tools, which are provided by a certification authority. The result of this kind of self-certification is reproducable and can be comprehended by the customer of the software, or an external authority. However, the "classical" certification scenario, with the authority performing the certifcation, is also supported. In both situations, human interaction is inevitable, but can be reduced through the introduction of certification tools. In one possible scenario, the compliance of product and specification is specified by the user, and the validity of this compliance is checked by the tool, yielding the certificate for the product.

Our approach focuses on non-functional properties such as performance and resource usage. The envisioned method shall be fully parametric regarding usage profile, deployment and assembly. For the initial version, we will assume that some of these properties are fixed in order to reduce the complexity of the problem.

This paper is structured as follows: In Section II, an overall view of the performance modeling constructs in the Palladio Component Model and the reverse engineering methods is given. Section III surveys related work, while the certification scenario is outlined in Section IV. The definition of refinement

rules and a formal representation is presented in Section V together with an example in Section VI. The limitations of the presented approach are discussed in Section VII. After an outlook on future work in Section VIII, the paper concludes with Section IX.

II. FOUNDATIONS

In this section, we will describe the parts of the Palladio Component Model that are relevant for our approach, and outline the reverse engineering approach for the extraction of performance properties from existing component implementations.

A. Palladio Component Model

The *Palladio Component Model (PCM)* [5] is a meta-model for the description of component-based software architectures. The model is designed with a special focus on the prediction of Quality-of-Service attributes, especially performance. Furthermore, a component-based development process is described that contains four types of developer roles: *component developer*, *software archtitect*, *system deployer* and *domain experts*. The part which is of interest for this paper concerns the *component developer*, who specifies the components and their behaviour.

Service Effect Specifications (SEFF) describe the relationship between provided and required services of a component. In the PCM metamodel, they are defined in the form of *Resource Demanding Service Effect Specifications (RDSEFF)*, which are used for performance prediction contain a probabilistic abstraction of the control flow. RDSEFFs use a notation stemming from UML activity diagrams, i.e. activities are denoted by nodes. For each RDSEFF, a *resource demand* can be specified as well as dependencies of transition probabilities and resource demands on the formal parameters of the service. RDSEFFs can be annotated to each provided service of a component. They describe

- how the service uses hardware/software resources;
- how the service calls the component's required services.

Resource demands in RDSEFFs abstractly specify the consumption of resources by the service's algorithms, e.g., in terms of CPU units needed or bytes read or written to a hard disk. Resource demands as well as calls to required services are included in an abstract control flow specification, which captures call probabilities, sequences, branches, loops and forks.

RDSEFFs abstractly model the externally visible behaviour of a service with resource demands and calls to required services. They present a grey box view of the component, which is necessary for performance predictions, because black box specifications (e.g., interfaces with signatures) do not contain sufficient information. RDSEFFs are not white box component specifications, because they abstract from the service's concrete algorithms and do not expose the component developer's intellectual property. Component developers specify RDSEFFs during or after component development and thus enable performance predictions by third parties.

```
void execute(int number, List array) {

    requiredService1();

    // internal computation
    innerMethod();

    if (number>=0)
        for (item in array)
        requiredService2();
    else
        requiredService3();
}
```


Figure 1.   SEFF Example (from [5])

To get an initial idea of RDSEFFs, consider the example in Figure 1. The top part depicts the simplified code of the service execute. The bottom part shows the corresponding RDSEFF. It includes calls to required services as ExternalCallActions, and abstracts computations within the component's inner method into an InternalAction. Control flow constructs are modelled only between calls to required services, while control within the internal computations is abstracted. The example includes parametric dependencies on the branch transitions and the number of loop iterations.

A single InternalAction can potentially subsume thousands of instructions into a single modelling entity as long as these instructions do not interact with other components and perform only component-internal computations. In many cases, an RDSEFF consists only of a few InternalActions and ExternalActions while at the same time modelling large amounts of code.

In addition to internal and external actions, RDSEFFs model control flow (Figure 2): StartAction and StopAction are the beginning and end points of the action chain of a ResourceDemandingBehaviour. Additionally, RDSEFFs may contain branches, loops, and forks.

Branches are expressed by BranchActions, which model XOR control flow alternatives. The contain a number of AbstractBranchTransitions, which can be either Guarded-
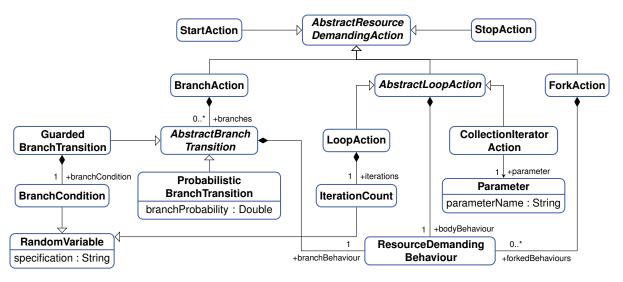
Figure 2.   RDSEFF control flow primitives (from [5])

BranchTransitions, i.e. depending on a parameter, or ProbabilisticBranchTransitions. Since the value of an input parameter is not known at design time, guarded branches can also have transition probabilities if the input parameter is specified as a stochastical function in the usage model. The branch probabilities of guarded branches are determined once the usage model is defined. Probabilistic branches can be used to model probabilistic behaviour without dependencies to the input values. Both kinds of branches contain a ResourceDemandingBehaviour, which includes actions executed in the body of the branch.

Loops can be modeled in two ways: LoopAction and CollectionIteratorAction. Both contain inner actions to model the behaviour of the loop body. A LoopAction includes the number of loop iterations as a RandomVariable, which consists of constants or abitrary distribution functions. The RDSEFF metamodel only allows to model loops explicitly, without backward references in the chain of actions within a ResourceDemandingBehaviour. For this reason, the graph structure of an RDSEFF is always a tree, since cycles or loops are modeled explicitly. CollectionIteratorActions are a special construct for loops iterating over a collection, so that the number of repetitions depends on the size of the collection.

ForkActions model AND control flow alternatives. The contained ResourceDemandingBehaviours are executed in parallel, so the succeeding action of a ForkAction is not executed until all forked behaviours have terminated.

B. Reverse Engineering

The performance predictions that can be performed with the Palladio Component Model are applicable in early stages of development as well as in the case of existing software. In order to obtain performance models from black-box components, Krogmann et al. [6] have developed a reverse engineering approach that uses genetic algorithms, static and dynamic analysis, and benchmarking. The approach has been validated

for Java-based code.

III. RELATED WORK

A. Formal Description of SEFFs

1) Finite Automata: In [7], Firus and Koziolek describe a formal model for *Service Effect Specifications* that is based on annotated finite automata. The automata are enriched by annotations that contain stochastical expressions for transition probabilities, depending on the usage profile. The basic elements are serialization, loops, and branches.

However, the actual SEFF metamodel specified as part of the PCM posesses a higher expressive power than finite automata. For example, the number of iterations in a loop may depend on an input variable, which is not expressable by a regular language.

2) Queueing Petri Networks: The semantics of PCM models have been described in [8] in terms of *Hierarchical Queueing Petri Nets (HQPN)* as defined in [9]. The performance-relevant parts of the PCM such as the usage model and RDSEFFs are modeled with Petri nets, but not the component-related concepts like interfaces and roles. The transformation presented by Koziolek also contains limitations regarding the stochastical expressions such that it is assumed that all distribution functions can be expressed by phase-type distributions and that all random variables are stochastically independent. Furthermore, composite and incomplete component types are not supported yet.

B. Refinement Calculi and Petri nets

In [10], rule-based modifications are applied to high-level Petri nets in order to reach a refinement that preserves safety properties, which are expressed in temporal logic. In contrast to other refinement calculi on Petri nets, the approach does not operate on low-level Petri nets, but on algebraic high-level nets. However, it is concerned with safety rather than performance properties of systems, like most refinement calculi for petri nets [11].

17

C. Certification of software implementation against SEFF

In [12], Groenda suggests a test-based validation of performance requirements, i.e. resource demands, against a deployed component implementation. With this approach, the validity of SEFF descriptions for component-based systems can be tested for certain parameter ranges that can be specified for the testing scenario. Based on this, automatic testcases are generated, which check the performance results against the specification and vice versa.

In Groenda's approach, the certification step is performed by an external certification authority. It is not intended for self-certification.
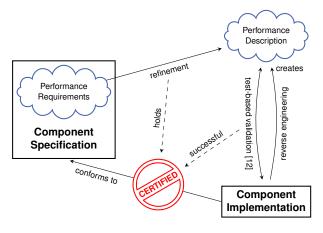

Figure 3. Certification Idea

IV. SCENARIO

The scenario of certification can be seen in Figure 3. In the proposed component-based software development process, a specifications document for components is created and enriched by non-functional requirements concerning the performance of a component (depicted as a cloud symbol on the left hand side). These requirements have to be laid down formally in the specifications document, using the formalism of *Service Effect Specifications (SEFF)* (see Subsection II-A).

The performance requirements serve as a contract which has to be fulfilled by the implementing party. However, Service Effect Specifications can not only be used in the specification of a software system, but also as a means of description for an actual implementation of that system.

Based on the specifications document, the implementation of the component is created, usually by a third party supplier. The resulting component is shipped with a description of its performance properties (depicted as a cloud symbol on the right hand side). This description can be determined by the implementing party in two ways: In the first case, the developer of the component creates the performance description manually. The conformance of these descriptions to the actual implementation has to be validated by the approach described in Subsection III-C. In the second case, the reverse engineering techniques discussed in Subsection II-B are used to create the performance descriptions a posteriori from the implemented

component. Assuming the correctness of these reverse engineering techniques, the resulting performance description can be used for a comparison with the requirements.

The availability of both the performance requirements and the performance description in the form of Service Effect Specification is a necessary precondition for the approach proposed in this work. If both artefacts are present, it is to be determined if the implementation SEFF (which serves as the performance description of Figure 3) is a *refinement* of the requirements SEFF (which serves as the performance requirements). For this purpose, a formal refinement definition is specified that allows both parties to check the conformance of implementation to specification regarding the performance properties, on the level of the abstract descriptions in the form of SEFF annotations. With the help of a checking tool, which could be provided by a certification authority, it is then checked if a refinement relation between the two SEFF artefacts holds, and if positive, the certificate can be issued. In case this performance description has been created manually, a validation has to be performed, which is indicated by "test-based validation" in Figure 3. If the refinement relation holds and the test-based validation is successful, this means that the implementation complies with the performance requirements.

The separation of performance requirements and performance description on the implementation side is necessary for several reasons:

- If one were to attempt a validation of performance requirements against the implementation directly, the approach of Groenda (see Subsection III-C) would not be applicable. The implementation of a component can have performance properties that are very different from the requirements in the sense of providing a much better performance, but still fulfilling the requirements. Thus, the test-based validation would return a negative result which does not reflect the intended purpose of the certification.
- If the performance descriptions are determined by reverse engineering techniques, the resulting description will not be identical with the performance requirements, so that the refinement is necessary for a conformance check.

In addition to these necessities, the separation of performance requirements and performance description has the following advantages:

- In a scenario where the component is developed by a third party, it may not be desireable that all implementation details of the component, i.e. its sources, are published to the certifying party or the customer. In order to protect intellectual property, the performance description can be used as an abstraction of the software.
- The refinement relation can also be used to find an existing component implementation from a repository that meets the performance requirements of the component specification.

V. RDSEFF REFINEMENT

A. Idea

For the definition of refinement between RDSEFFs, two things are required. Firstly, the notion of refinement itself has to be described, and secondly, a formal representation of RDSEFFs and the rules has to be found in order to check if the refinement relation holds for actual RDSEFF instances.[1]

In the following subsection, we will provide a set of refinement rules that defines the refinement relation constructively: If there is an application of rules for two given SEFFs, then the refinement relation holds. The proof consists of a set of valid rule applications.

For the top refinement relation, we propose a definition of refinement that is dependent from the resource demand of the SEFF:

Definition 1: A Service Effect Specification $SEFF_1$ is refined by $SEFF_2$ if, for each resource, the resource demands of $SEFF_1$ are greater or equal than those of $SEFF_2$:

$$SEFF_1 \geq_{RD} SEFF_2 \qquad (1)$$

Whether the relation $\geq_{RD}$ is given depends on the elements contained in the SEFFs, since the total resource demand of a SEFF is calculated by the actions contained in it. In the following subsections, the single refinement steps for the different types of actions will be described. In order to express the resource demand of an action a, we write RD_a. A comparison on resource demands means that for every type of resource, the resource demands are compared.

This definition is quite coarse at the moment. The resource demand of a SEFF or of the actions contained in it can be parametric regarding the input parameters. It can be a complex stochastical expression for which the "greater than or equal" comparison of Definition 1 may be difficult to define. At the moment, we assume that the comparison of two stochastical expressions is possible and focus on the structural possibilities of refinement.

B. Refinement steps

An internal action is the simplest building block of a SEFF, since it represents an action that has no externally visible behaviour. It can contain a resource demand. For an internal action, we define the following refinement steps:

Definition 2: An internal action a is refined by

- an internal action b if $a \geq_{RD} b$
- a branch action c containing[2] inner actions d_i, $i \in \mathbb{N}$ if $\forall d_i \in c : a \geq_{RD} d_i$
- a loop action l with a maximum number of iterations $p \in \mathbb{N}$ containing an action e if $RD_a \geq \sum_p RD_e$

Note that the refinement for the branch action does not take into account the branch condition. For this estimation, it is only important that the inner action fulfills the comparison of resource demands, so that $\geq_{RD}$ holds in any case.

The following two definitions are derived from Definition 2, but consider the opposite direction of refinement: It is also possible to refine an action with a less complex internal action. As a consequence of this, the refined SEFF is not necessarily smaller in its structure, which has serious influences on the complexity of checking for a valid refinement relation (see Section VIII).

Definition 3: A branch action b containing inner actions c_i is refined by an internal action a if $\forall c_i \subset b : c_i \geq_{RD} a$.

Definition 4: A loop action l with maximum number of executions $p \in \mathbb{N}$ containing an action c is refined by an internal action a if $\sum_p RD_c \geq RD_a$.

As we can see, all these refinement rules are "worst case" rules: it is assumed that the resource demands are always less or equal, independent of the circumstances, i.e. the usage profile. This is why we do not take the branch condition into account and only regard the maximum number of loop iterations. It is possible to define different notions of refinement, so that for example the resource demands are lower or equal on average, and not in all cases. For the certification scenario outlined in Section IV, this weaker refinement would however not be useful, so we stick to the stronger definition for the moment.

C. Formal definition of refinement

After having defined the refinement steps in Subsection V-B, a formal representation of these rules has to be determined. In the following subsections, we will present several possibilities for the representation of SEFFs and the refinement relation.

1) SEFF metamodel: The definition of Service Effect Specifications is laid down in the Palladio Component Model [5]. There, an EMF metamodel for *Resource Demanding Service Effect Specifications (RDSEFF)* is described. This is a formal definition which serves as a basis for the modeling tools of Palladio and for various model transformations that can be performed on PCM instances.

However, for our purposes, this definition is not sufficient. In order to check whether a refinement between two instances of a SEFF exists, a series of applications of the refinement rules described in Subsection V-B has to be provided. If such a series is found, it is possible to define the concrete refinement relation as a model transformation in terms of the model-driven development. However, the general definition of valid refinement would require a type of higher-order transformation, of which the actual refinement relation would be an instance. But even then, the determination of such an instance from two existing SEFFs would still be a problem which cannot be solved with the mechanism of model transformations. For this reason, we are looking into different formalisms to express the refinement relation.

2) Petri Nets: Higher forms of Petri nets such as Stochastical Petri nets are a possible formalism, since a transformation of SEFFs in to Hierarchichal Queueing Petri nets (HQPN)

---

[1] In the following, we write SEFF instead of RDSEFF for the sake of brevity.
[2] The containment relation is expressed with set operators here. By containment, we mean the relation depicted as ◆— in Figure 2.

has been performed with limitations in [8]. The process of refinement can be described in terms of Petri net refinement as mentioned in Subsection III-B, but the problem here is also that the source and target net already exist. Refinement usually describes a transformation that is to be applied on a source net, which creates the target. Here, the refinement relation has to be created for existing nets. Furthermore, a class of refinement steps has to be defined which represents our desired refinement relation, and a proof method has to be established that checks if a given refinement complies to this definition.

3) Prolog encoding: Since the refinement steps are defined as single refinement rules, a rule-based approach can be used to define the refinement relation between SEFF instances and also to find the sequence of rule applications that refines the source instance to the target instance. Rule-based languages like Prolog [13] can be used to formally describe the rules of Subsection V-B, provided that an encoding of Service Effect Specifications in Prolog is defined.

```
1   % checks if the demand of action A is greater
2   % or equal than those of the actions in a list
3   demand_geq(A,[Head|Tail]) :-
4         demands(A,D1),
5         demands(Head,D2),
6         D1 >= D2,
7         demand_geq(A,Tail).
8   demand_geq(A,[]).
9
10  % refinement of internalAction to branchAction
11  refinesIntBranch(I1,B1) :-
12        intaction(I1),
13        branchaction(B1),
14        demands(I1,D1),
15        branches(B1,L),
16        demand_geq(I1,L).
```

Listing 1.   Refinement of internal action to branch action in Prolog

In Listing 1, we can see the Prolog representation of the second part of Definition 2 (refinement of an internal action by a branch action). For the refinement, we need the recursive helper function `demand_geq()` that compares an action `A` with a list of actions. The clause `branches(A,B)` indicates that action `A` demands a number `B` of resources (for the sake of simplicity in this example, `B` is a number). Lines 12-15 describe the preconditions for refinement, such as the correct types of actions. If the clause in line 16 also evaluates to true, then the refinement relation holds.

VI. EXAMPLE

A. Description

In the example of Figure 4, two different SEFFs are shown. For the purposes of this example, it is assumed that these SEFFs describe the behaviour of the same service, for example in a scenario where $SEFF_1$ is part of the component specification, whereas $SEFF_2$ is adjacent to the implementation. In order to prove that $SEFF_2$ is a refinement $SEFF_1$, one has to provide a mapping of the single actions of $SEFF_1$ to $SEFF_2$.

In our case, determining a mapping is quite straightforward, since the SEFFs are not complex. In the example,
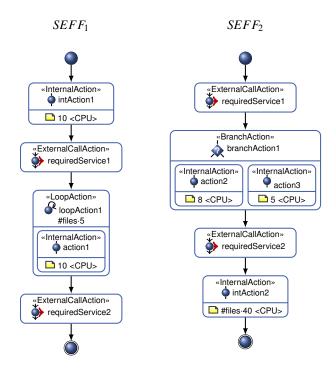

Figure 4.   Refinement Example

we have calls to two external services: requiredService1 and requiredService2. To both services, calls occur in $SEFF_1$ as well as in $SEFF_2$; the ordering is the same in the sense that requiredService1 is always called before requiredService2. Since the internal actions between the external calls do not affect the execution time of the external calls in this case, the refinement association is valid for these actions.

For the other actions, matching is more complex. Since we have different kind of actions on both sides, we must investigate all possible combinations. For example, if we compare intAction1 with branchAction1, we can see that branchAction1 contains two elements of the type InternalAction, which are easily comparable with intAction1. In this case, each of the internal actions contained in branchAction1 has a smaller resource demand than intAction1. This means that a refinement relation holds between intAction1 and branchAction1. This comparison is independent of the conditions under which the branch in branchAction1 occurs, since in every case the resource demands will be lower than the one in intAction1.

In comparison, no refinement relation exists between intAction1 and intAction2, since the resource demand of intAction2 grows linearly with an input parameter (the number of files) and is thus not necessarily lower than the fixed resource demand of intAction1. For the same reason, loopAction1 is not refined by branchAction1, since the internal action of loopAction1 has a resource demand that depends on an input parameter, but those of branchAction1 do not, and so it cannot be guaranteed that the resource demand of branchAction1 is always lower than that of loopAction1.

So the only two elements that remain are loopAction1 and

intAction2. The number of iterations of the loop is dependent from the number of files, and the resource demand of the internal action is fixed, so we can calculate that loopAction1 has an overall resource demand of #files·50. The resource demand of intAction2 is also linearly dependent from the number of files, but with a smaller factor, so the refinement relation holds between loopAction1 and intAction2.

B. Prolog

As an example of how such a SEFF can be coded in Prolog, we describe the actions intAction1 and branchAction1 from Section VI in Listing 2. The clauses should be quite self-explaining; as mentioned in Subsection V-C3, we are using natural numbers in the resource demands clause `demands()` for the sake of simplicity here.

```
1  intaction(intaction1).
2  demands(intaction1,10).
3  branchaction(branchaction1).
4  intaction(action2).
5  intaction(action3).
6  demands(action2,8).
7  demands(action3,5).
8  branches(branchaction1,[action2,action3]).
```
Listing 2. Example encoded in Prolog

Now that we have the SEFF instances, we can pose a query, which together with the refinement rule of Listing 1 gives us the following result:

```
1  ?- refinesIntBranch(X,Y).
2  X = intaction1,
3  Y = branchaction1 ?
4  yes
```
Listing 3. Query on refinement

With `refinesIntBranch(X,Y).`, we are querying if there are two entities that fulfill our refinement rule (which, in this example, only applies to the refinement of internal actions to branch actions). The interpreter delivers `intaction1` and `branchaction1` as the correct result.

VII. ASSUMPTIONS/LIMITATIONS

The approach presented in this paper can only be performed if the component specification as well as the component implementation contain performance descriptions in the form of Service Effect Specifications. The correctness of these descriptions is assumed. For the implementation side, this can be checked by the techniques mentioned in Subsection III-C or by reverse engineering techniques that can guarantee the conformance of performance description to implementation. Since the final certificate states that the implementation conforms to the performance requirement laid out in the specifications document, both the refinement check and the conformance check are necessary.

The formal refinement check is only correct under the assumptions that the refinement rules that are used are also correct. The preservation of resource demands or the fulfillment of performance requirements is not checked directly like in [10], but is encoded in the refinement rules: if there

is a valid application of rules, then the refinement relation holds. The rules themselves are not formally proved to be correct. The reason for this is that the performance semantics of PCM model instances are not specified formally either, so the correctness of refinement can not be checked formally at the moment.

VIII. FUTURE WORK

A. Comparability of Stochastical Expressions

In the presented approach, it is assumed that the resource demands contained in the SEFFs are comparable, so that a "greater than" relation can always be specified. However, resource demands are specified with the *Stochastical Expressions* language of the PCM, and these expressions can reach a complexity for which a comparison is not yet defined. For this reason, the comparability of Stochastical Expressions has to be researched further so that the approach presented in this paper can be applied to SEFFs containing any kind of resource demand definition, and also to take the usage profile into account.

B. Refinement rules

The definition of rules in Subsection V-B only covers some SEFF elements (branch, loop, internal action) and must be extended for all elements of the Palladio Component Model. Further aspects of refinement are not yet covered in the initial version presented in this paper. These aspects include

- rules for the matching of actions, so that all elements in the SEFF are part of a refinement rule
- ordering of actions (especially external actions)
- combining of actions so that multiple actions are refined by a single action
- preconditions for refinement, e.g. comparibility of certain types of stochastical functions

Furthermore, it may be possible that a refinement can be found between two SEFF instances that may require several applications of refinement rules. This would require the definition of intermediate states, i.e. SEFF instances that are neither part of the source nor of the target model. With the current approach, this is not possible yet, but can be included in a pre-processing step that would determine the intermediate states and add them to the set of SEFF instances for which a rule application path must be determined.

C. Formal representation

In order to check the refinement property for existing systems, the encoding of SEFFs in a formal language, e.g. Prolog, has to be specified. For this purpose, a model-to-text transformation can automatically generate the necessary code for an interpreter, which can than determine whether the refinement relation holds. For cases in which the refinement relation can not be found, a mechanism should be developed that points out the problems so that the parts which prohibit a valid refinement can be identified.

D. Validation

For the validation of the approach presented in this paper, a case study would have to cover the development process of a component from specification to implementation. From the implemented component, performance properties can be extracted through reverse engineering techniques. Then, the check for a refinement relation is performed for the SEFF contained in the specification and the generated SEFF, first manually following the refinement rules defined in natural speech, then automatically via the formal checks. This way, it can be determined in which cases a refinement relation can be found and in which not. From this experience, the set of refinement rules can be improved in order to meet the requirements.

IX. Conclusion

The proposed approach extends the notion of software certification into two directions. Firstly, the subject of certification is shifted from functional correctness to the fulfillment of non-functional properties such as performance. Secondly, the certification process is based on formal methods that correlate the specification documents to the actual implementation for component-based systems, so that an actual product certification can take place.

The process requires that software specification documents are enriched with descriptions of performance properties, which can later be checked formally and thus yield a certification that is concerned with the quality of software. The formal foundations of this process make it possible to use certified checking tools for the conformance checks and do not force the software developer to reveal intellectual property in the form of source code to the certification authority, but still gives the customer the possibility to comprehend the check of compliance between the requirements and the delivered piece of software.

Since the process is based on the Palladio Component Model, existing tools and mehods can be used to determine performance requirements and the performance properties of existing systems. For the latter case, reverse engineering methods make the approach also accessible if no performance descriptions in the form of design documents exist for the component implementation.

References

[1] J. Voas, "Developing a usage-based software certification process," *Computer*, vol. 33, no. 8, pp. 32–37, Aug 2000.

[2] K. C. Wallnau, "Software component certification: 10 useful distinctions," SEI, CMU, Tech. Rep. CMU/SEI-2004-TN-031, September 2004.

[3] C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer, "Integrating Verification and Testing of Object-Oriented Software," in *Tests and Proofs. Second International Conference, TAP 2008, Prato, Italy*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Berlin/Heidelberg: Springer, 2008, pp. 192–191.

[4] R. H. Reussner, S. Becker, H. Koziolek, J. Happe, M. Kuperberg, and K. Krogmann, "The Palladio Component Model," Universität Karlsruhe (TH), Interner Bericht 2007-21, October 2007. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/reussner2007a.pdf

[5] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[6] K. Krogmann, M. Kuperberg, and R. Reussner, "Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction," *IEEE Transactions on Software Engineering*, 2010, accepted for publication, to appear.

[7] H. Koziolek and V. Firus, "Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation," in *Proc. of the 5th Int. Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'06)*, ser. ENTCS, J. Kuester-Filipe, I. H. Poernomo, and R. H. Reussner, Eds., vol. 176, no. 2. Elsevier Science Inc., March 2006, pp. 69–87. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/koziolek2006e.pdf

[8] H. Koziolek, "Parameter dependencies for reusable performance specifications of software components," Ph.D. dissertation, Universität Oldenburg, Uhlhornsweg 49-55, 26129 Oldenburg, 2008.

[9] F. Bause, P. Buchholz, and P. Kemper, "Hierarchically combined queueing petri nets," in *11th International Conference on Analysis and Optimization of Systems Discrete Event Systems*, ser. Lecture Notes in Computer Science, G. Cohen and J.-P. Quadrat, Eds., vol. 199. Berlin/Heidelberg: Springer, 1994, pp. 176–182.

[10] J. Padberg, M. Gajewsky, and C. Ermel, "Rule-based refinement of high-level nets preserving safety properties," *Science of Computer Programming*, vol. 40, no. 1, pp. 97 – 118, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V17-42815TS-5/2/8c7f94a0a6e23e6eacebf11014b31312

[11] W. Brauer, R. Gold, and W. Vogler, *A survery of behaviour and equivalence preserving refinements of petri nets*, ser. Lecture Notes in Computer Science, Berlin/Heidelberg, 1991, vol. 483.

[12] H. Groenda, "Certification of software component performance specifications," in *Proceedings of the Fourteenth International Workshop on Component-Oriented Programming (WCOP) 2009*, ser. Interner Bericht. Fakultät für Informatik, Universität Karlsruhe, R. Reussner, C. Szyperski, and W. Weck, Eds., vol. 2009-11, 2009, pp. 13–21. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012168

[13] E. Y. Sterling, Leon ; Shapiro, *The art of prolog : advanced programming techniques*, 2nd ed., ser. (MIT Press series in) Logic programming. Cambridge, Mass: MIT Press, 1994.

Architectural Modelling in Agile Methods

Zoya Durdik

FZI Research Centre for Information Technology
Software Engineering (SE)
76131 Karlsruhe, Germany
durdik@fzi.de

Abstract—**Agile methods and architectural modelling have been considered to be mutually exclusive. On the one hand, agile methods try to reduce overheads by avoiding activities that do not directly contribute to the immediate needs of the current project. This often leads to bad cross-project reuse. On the other hand, architectural modelling is considered a pre-requisite for the systematic cross-project reuse and for the resulting increase in software developer productivity. In this paper, I discuss the relationship between agile methods and architectural modelling and propose a novel process for agile architectural modelling, which drives requirements elicitation through the use of patterns and components. This process is in-line with agile principles and is illustrated on an example application.**

I. INTRODUCTION

Agile methods gain increasing attention in the broader software developer community [1]–[3]. The notion "agile methods" is used for software development processes that are incremental, have strong customer involvement and are sharply driven by the current needs of a project. They avoid overhead, i.e., activities and artefacts, which do not directly contribute to the goals of the current increment [4]. Agile methods are often adopted by smaller companies; however, an increasing number of larger companies with established heavyweight processes are also adapting agile development methods because of their lower overhead [5]–[8]. Success stories report on well-managed risks, high chances of customer satisfaction, and also a good climate in the development team [2], [9]–[13].

However, the fact that agile methods avoid overhead can be seen in a strong contrast to architectural modelling [4]. This is because the effort of explicitly modelling the architecture of a software system usually does not pay off during initial development. However, architecture modelling pays off rather in the evolution phase or if a cross-project reuse occurs because of the modelled architecture [14]. Exactly here lie the drawbacks of agile methods: cross-project reuse of software artefacts, such as patterns, reference architectures and components are not foreseen, and product-lines are not supported. However, software evolution is supported by agile methods in principle through code refactorings [15]. Refactorings are functionality-preserving changes of code, usually performed as a preparation for adding a new functionality [16]. Those refactorings are not concerned with the overall architecture of a software system. Furthermore, if the architecture is of any concern at all, it is considered to be fixed throughout the project [1].

The challenge addressed in this paper is how to reconcile agile methods with architectural modelling in the software development process. Its benefits would be the following: agile methods and their incremental approach lower risks, customer involvement lowers the danger of developing the wrong product. The explicit architectural modelling supports reuse of components and patterns and would also extend the degree of creating reusable artefacts for future projects. Furthermore, the presence of architectural models would support cost estimation [17] and architecture evaluations [18]–[20]. Information extracted from architectural decisions (reused design patterns and components) would support the requirements engineering process through a more goal-oriented requirements elicitation.

Given these benefits of architectural modelling, one can also envision how architectural modelling can support agile principles: architectural design supports lightweight development processes by systematically reusing existing artefacts, preventing wrong design-decisions being implemented through early architectural evaluations and by helping architects to make important design decisions [21].

Therefore, in this paper, I present an agile architecture modelling process which is driven by the reuse of patterns and components and itself drives the requirement elicitation by explicitly asking questions back to the customer.

The contribution of this position paper is the discussion of the antimony and symbiosis of agile methods and architectural modelling and the definition of an architecture-driven modelling and requirements engineering process. This process is a step towards architectural modelling in agile processes as well as for an agile architectural modelling process.

This paper is organized as follows: The following section discusses agile methods and architectural modelling. Section III presents a general model of an architecture-driven modelling and requirements engineering process. An example application is introduced in section IV. Section V presents process benefits and discussion.

II. FOUNDATIONS

Research in the software development processes often focuses on the heavyweight processes, which are complicated and highly time and resource demanding. The market pressure is increasing; projects often suffer from budget and time pressure in a race for a quick product deliverable. The amount and the speed of new technologies, requirements and standards

23

appearing nowadays produce additional pressure on the development process. Classical software engineering methods (like waterfall) require a big planning up-front and are not flexible enough to react on rapidly changing conditions. As a reaction to this, a method family "agile methods" was developed.

The most famous agile methods are: Extreme Programming (XP) [22], Scrum [23], Crystal Clear [24], Feature Driven Development [25] and Adaptive Software Development [26].

These methods have the following common characteristics:

- iterative and incremental process;
- lightweight process (as few forward planning as possible, code is the main artefact, many classical practices e.g. architecture modelling are considered to be redundant: YAGNI Principle "You Ain't Gonna need It");
- flexible (quick response to a changing environment, new requirements are welcomed);
- goal-oriented (project value oriented, every development increment shall add value to the product);
- customer oriented (strong customer involvement);
- team-oriented (team has the main role, self-organized).

The agile manifesto explicitly states the principles of agile development [27]: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan.

As already mentioned, agile methods are increasingly gaining popularity in the broader developer community. They promise a quick reaction to the changing environment and new requirements while reducing cost and overhead. There is a big number of success reports, especially for the methods XP and Scrum [2], [9]–[13].

However, empirical studies on the quality and efficiency of these methods are less clear [28], [29]. It seems that agile methods improve well over ad-hoc processes, but elements such as pair-programming and test-driven development (common XP practices) are not demonstrated yet to be superior to established classical software development quality assurance techniques, such as code and architectural reviews.

Software architecture is a high level abstraction of software [30]. Documenting architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view. A view is a representation of a set of system elements and the relations associated with them [30]. Architecture can be represented either graphically in a model form, for example with the help of the Unified Modelling Language (UML) [31], or with the help of Architectural Description Languages (ADL) [32].

For example, the famous "4 + 1 View Model" of P. Kruchten "describes software architecture using five concurrent views, each of which addresses a specific set of concerns" [33]. It consists of: the logical view (functionality of the system), the development view (static organization of software in the development environment), the process view (design's concurrency and synchronization aspects), the physical view (mapping of the software onto the hardware) and the scenarios (system use cases).

This includes views on its static structure (i.e. components and connectors), the inter-component control flow and the deployment of components and connectors on virtual or physical resources.

Software architecture typically plays a key role as a bridge between requirements and implementation [34]. Architecture and architecture modelling have the following benefits [30], [34]:

- communication: architecture may be used as a focus of discussion by system stakeholders;
- understanding: presenting a complex system at an easier abstraction level;
- analysis: consistency checking, conformance to constraints and quality attributes, dependence analysis;
- reuse: architectural descriptions support reuse at multiple levels and across a range of systems (styles, patterns, components, frameworks, code); existing components can be considered during design (COTS, in-house components, commissioned or off-shore);
- management: evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies and potential risks (cost-estimation, mile stone organization, dependency analysis, change analysis, staffing);
- implementation support: provides a partial blueprint for development by indicating the major components and dependencies between them;
- evolution: exposure of the dimensions along which a system is expected to evolve

There are two plausible scenarios where architectural modelling meets beneficially:

- *agile architectural modelling* is concerned with lowering the overhead of architectural modelling by using an incremental, customer-involved process. Challenges lie in the definition of architectural refactorings and a suitable redefinition of the term overhead.
- *agile methods with architectural modelling* are concerned with the introduction and utilization of architectural modelling in agile methods. In this scenario, one needs to define an agile process which works with several artefacts, i.e. with code and architecture. In addition, the benefits of architectures, as discussed need to be exploited in order to support an overall agility. This includes the use of architectures for cost estimation, early evaluations of architectural design decisions and the use of architectures to elicit the right requirements.

In the next section, I present a process where architectural modelling is done in an incremental process highly intertwined with requirements engineering. I claim that such a process is necessary for the above mentioned integration scenarios. Firstly, my process is an agile architectural modelling process, as it possesses central elements of an agile method. Secondly, my agile modelling process could play a central role in an agile method with architectural modelling.
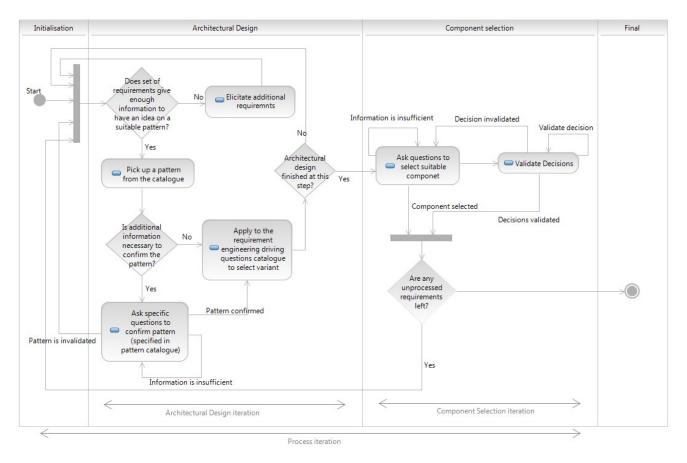
Fig. 1.   An agile architecture-driven modelling and requirements engineering process, Activity diagram

III. AN AGILE ARCHITECTURE-DRIVEN MODELLING AND REQUIREMENTS ENGINEERING PROCESS

This section presents the proposed process, describes separate steps and challenges connected to them.

The process is iterative and incremental. Each iteration consists of the set of phases (initialization, architectural design and components selection, accomplished with a goal-oriented requirements elicitation) which form a single process increment (see Scrum: Sprint). With each iteration, the person applying the process moves a step toward to the ready system design and adds product value as will be demonstrated below. Beside this, the artefacts are created according to the demand of the current process stage, which keeps the process lightweight. The innovation lies in the fact that architectural design and the potential reuse of component candidates are drivers of the requirements elicitation process.

The person using this process shall be involved in the system negotiation, design and development and shall have a technical understanding in the area of software intense system development (software architecture, in particular). This person may be either a software architect, or a person who has taken over her responsibilities in the development process. This can be technical project lead, technical team lead, team or software developer.

Therefore, the proposed process conforms to the characteristics of agile methods. It is presented as an activity diagram in Fig. 1 and consists of the following coarse-gained steps, which are repeated in iterations a) inside of each step and b) together:

Step 0. Initialisation.

a) Gather initial information about the system, i.e. its goals, required functionality and environment conditions.

b) Check if the following assumptions are fulfilled: initial requirements to the system are elicited, informally or formally recorded and are prioritized. The requirement prioritization is a challenging topic, however it lies beyond this paper's scope and will not be discussed. For the state of the art one may apply for example to [35]–[38].

Step 1. Architectural design.

a) Analyse if there is enough information provided by the gathered requirements for selection of a suitable pattern. I propose using a pattern catalogue for this purpose. Such a catalogue contains a detailed pattern description, like a pattern goal, benefits, drawbacks, variant, application cases and connected and similar patterns. If the initial information is not enough, additional requirements to be elicited first. This

25

can as well be supported through the pattern catalogue. The pattern selection may be tool supported.

b) Confirm the pattern. To finally decide whether a pattern is really suitable, additional information may be needed. In this case, one can use questions provided in the pattern catalogue to elicit additional requirements unless the pattern is confirmed (appropriate use) or invalidated (the idea of using the pattern is finally not appropriate). If a pattern was invalidated, I propose selecting the next pattern from the catalogue identified as a potential candidate. If the pattern was confirmed, proceed to the pattern variant selection (most of the patterns have several possible variants).

c) Select a pattern variant. I propose supporting the person using this process (further on called software architect) by a requirement engineering questions catalogue. It shall contain a description of the variant properties and a list of questions. These questions shall navigate the software architect through the decision-taking process. The design of such a kind of pattern catalogue would also have the benefit that the navigating questions also extend the pattern documentation in understanding when to use which pattern variant in general (also beyond the use of the process proposed here).

d) Check if there are enough architectural decisions taken to proceed to the next step: component selection. Only a part of the system is designed at each iteration, as the system architectural design is not intended to be completed (the process proposes an incremental development). If information is not yet enough to proceed to the next phase, the software architect shall return to point a).

New requirements and decisions taken at this step shall be documented; they can be used to complement the pattern and requirement engineering questions catalogues for future projects. Elicitation of new requirements may require customer (stakeholder) involvement into the process.

Step 2. Component selection.

This step is similar to the step 1, however it goes one level of abstraction down to the selection of re-usable components out of an repository.

a) Analyse the selected pattern variant structure. As the pattern consists of "building blocks", these blocks may be represented through components and thus can be reused.

b) Select a suitable component for each building block, if any component is available. The questions associated with this component can be used to either validate or invalidate the decision. These questions will help the architect to elicit additional requirements unless the component will be confirmed (can be reused) or invalidated (is found to be inappropriate).

For this step, I propose supporting the software architect with a tool for architectural evaluation. Certain decisions (especially regarding quality requirements) may not be obvious to him, or in some cases she may identify several components implementing similar functionality. Existing tools for architectural evaluation are for example the Palladio Component Model (PCM) [39] and AEvol (A tool for defining and planning architecture Evolution) [20].
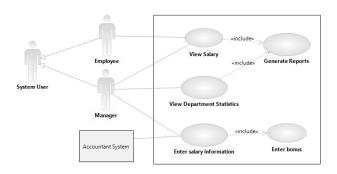

Fig. 2. The salary viewing system, Use Case diagram

If components for each building block of the selected pattern variant are selected, confirmed or invalidated, the software architect has finished the iteration and the increment.

c) Check if there any unprocessed requirements. At this point, the software architect can either proceed to the start to make more iterations and increments or quit the process.

Questions connected to the selected components help the architect to validate decisions and elicit new requirements. Therefore, this paper assumes that the components will be preferably selected and not created.

IV. AN EXAMPLE APPLICATION: A SYSTEM FOR VIEWING SALARIES

This section provides an illustrated example – a software system for viewing salaries, demonstrating the process and its advantages.

The salary viewing system (SVS) is a system for viewing salary information and generating reports. It is a simplified example which shall demonstrate the agile architecture-driven modelling and requirements engineering process.

The system provides a possibility for an employee to view the salary and to generate various reports, like annual salary or bonus overview. A manager can additionally view and edit salaries and bonuses of her employees. She has a possibility to view her department statistics and generate reports.

A Use Case diagram for the system is presented on the Fig. 2.

I proceed with the demonstration of the process on this test example; it is very simple lest we lose the focus from the process.

Step 0. Initialisation.

a) Initial information about the system is gathered. One has the knowledge that this is the system for viewing salary information and generating reports, which shall provide a possibility for an employee to view her salary and to generate various reports and a manager can additionally view and edit salaries and bonuses of her employees.

b) The initial requirements to the system are affiliated, informally recorded and prioritized. The software architect decides for a list form to record requirements, and the list is

TABLE I
A SAMPLE LIST OF REQUIREMENTS FOR THE SVS

Num.	Requirement	Priority	Risk
1	System must provide own salary and bonus information for any user	H	M
2	System must provide employee's salary information for her manager	H	L
3	System must provide department statistics for its manager	H	L
4	System must provide employee's salary and bonus editing functionality for her manager	H	L
5	System shall be able to support new report types	L	H
6	System must generate monthly / quarterly / annually reports	M	L

presented in the Table I. These requirements are: the system must provide separate salary and bonus information for any user, system must provide employee's salary information for her manager, system must provide department statistics for its manager, the system must provide employee's salary and bonus editing functionality for her manager, the system shall be able to support new report types and the system must generate monthly / quarterly / annually reports.

Since this process step is completed the software architect can proceed to the next step.

Step 1. Architectural design.

a) The requirements number 1-3 can be summarized as follows: several system user groups (manager, worker) require different data representations (view salary, view department statistics).

There is enough information to have the idea that the Model-View-Controller Pattern (MVC) could be appropriate.

MVC divides an interactive application into three components [40], [41]. The model contains the core functionality and data. Views display information of the user. Controllers handle user input. Views and controllers together comprise the user interface. A change propagation mechanism ensures consistency between the views and the model. The MVC pattern is often used with the Observer pattern.

The benefits of the MVC are [40], [41]: Multiple views of the same model, synchronised views, "pluggable views and controllers", exchangeability of "look and feel" and framework potential.

The drawbacks are [40], [41]: increased complexity, potential for excessive number of updates, intimate connection between view and controller, close coupling of views and controllers to a model, inefficiency of data access in view, inevitability of change to view and controller when porting and difficulty of using MVC with modern user interfaces tools.

b) To confirm the use of the MVC pattern, one has to check whether there is the need to keep views and models synchronised. Although this sounds trivial, it only becomes an issue if updates of the data can be done by one of several views. This is because the use of the MVC pattern would be a potential over-design, if the views provide no editing

possibility and all data updates come from the model. Here instead of the MVC pattern one could use classical data views on a data-base (as provided by database management systems or by mechanisms of ADO.NET for example).

Hence the questions are:

- Shall the user be able to manipulate data through views?
- Do views need to be consistent?

The requirement 4 "System must provide employee's salary and bonus editing functionality for her manager" states that the user shall be able to be manipulate data through views. For this example I assume that the software architect confirms the pattern. She also elicits a new requirement: the views have to be consistent.

c) The MVC Pattern has three possible forms: 1. The view is connected to the model through a controller, 2. The view is directly connected to the Model, 3. Mixed form of these two variants: view is connected to the model through a controller but in some case has a direct access to the model.

Sample questions which can be associated with the pattern:

- How many views will be needed?
- How many updates will the system have to cope with?
- What is the data volume of each update?
- Can controller become a bottleneck?
- What performance requirements are put on the system?

The software architect could derive the following the MVC specific questions:

- Shall user only be able to use predefined buttons to manipulate report queries? (pattern variant 1)
- Shall user be able to construct all queries together through GUI himself? (pattern variant 2)
- Shall this functionality depend on a user group? (pattern variant 3)

Based on these questions, the software architect chooses the variant 1. She has elicited a new requirement: Users must be able to (only) use predefined buttons to manipulate report queries and data.

d) The software architect can proceed to the component selection as she has gathered enough information to execute the next step.

Step 2. Component selection.

a) The selected pattern variant consists of three general "building blocks": model, view and controller. To keep the example simple, I will only explore the model "block" of the MVC (as it allows interesting technical speculations and thus questions to the system). In my application, a database (DB) is used as the model.

b) If a customer has no preferred technology or vendor, the software architect can be supported by a set of questions. For example the following questions can be helpful while selecting the suitable component (DB Type) for my SVS example:

- Is a database management system (DBMS) necessary or is simple data file based persistence sufficient?
- Are there concurrent accesses?

- Is the file-system access control sufficiently detailed (or is the more sophisticated access control of a DBMS required)?
- Who will be responsible for the system maintenance (does responsible side have enough expert knowledge)?

These questions will help the architect to select a suitable reusable component and elicit additional requirements. Choosing a component can also influence quality properties of the system. Here the software architect could also have used an architecture evaluation tool to evaluate possible decisions. This could be, for example, an average system response time with a defined amount of system users per period of time.

c) Unprocessed requirements to the SVS system are left, thus the software architect can make several more iterations.

I argue that already at this process point: a) new requirements were elicited more goal-oriented, b) a significant increment of the cross-cutting knowledge about the system was achieved and c) reusable artefacts (components) were identified.

V. PROCESS BENEFITS AND DISCUSSION

As demonstrated above, the benefit of this process lies in:

a) A more goal-oriented elicitation of requirements: the proposed process solves two problems, firstly, the danger that unsuitably cut requirement chunks can lead the architectural design process in a wrong direction and causes a lot of re-work, and secondly that it is unclear which requirements are truly necessary for making architectural decisions.

b) A systematic consideration of reusable artefacts: in the presented method the decision to reuse artefacts is done systematically based on the actual requirements.

c) The encouragement of reuse of architectural knowledge through patterns: the use of patterns itself presents the reuse of architectural knowledge.

I argue that the process conforms to the characteristics of agile methods:

- iterative and incremental: the process is iterative and each process iteration consists of the set of steps which form a single process increment;
- lightweight: a few forward-planning is required and only that artefacts are created, which directly contribute to the goals of the current increment;
- goal-oriented: supports the project value-oriented making of decisions;
- flexible: new requirements (and also their elicitation) are supported;
- strong customer involvement: the process supports customer involvement by pattern and requirement engineering questions catalogues;
- team-oriented: the process can be executed by e.g. a team, which took over the software architect's role (as proposed in Scrum).

The process presented in this paper should be seen as an example. It shows a back flow of information from architecture design into requirements elicitation through pattern and component selection. However, these steps may also appear in different order, i.e. components are selected before major architectural decisions are taken through the selection of architectural patterns.

VI. RELATED WORK

Questions concerning the role and potential insertion of architecture modelling in agile methods and their reconciliation are matters of high interest in research, judging by the amount of articles appearing on these topics. However, most of them tend to be rather philosophical than practical. Several relevant practical approaches are presented at the end of this section.

A good introduction into the problem is presented by Abrahamsson et al. in [4]. This article discusses the roots of misunderstanding architecture modelling and agile methods contradictions. It questions the general challenge of their reconciliation and gives several general advices, but provides no practical method.

[42] provides an introduction into agile methods, UML-modelling and describes general agile modelling methodologies. However all advices are kept very general, mostly concentration on the "best practices" of Extreme Programming (XP) and are more concerning a very general management level.

[43] presents a project success report where architecture and Scrum (one of the agile methods) were joined. This report however neither gives any details on architecture modelling (if it was used and how), nor proposes any method or process for their reconciliation.

An exploratory study conducted by IBM and University of Rome Tor Vergata [44] describes an agile developer perspective on the architecture. It shows that in 95% of cases, developers consider a focus on architecture as important, but it does not state what is understood by architecture in this case. It does not propose any method.

There is a set of approaches, dealing with the decomposition of the problem domain basing on the problem frames and extending them with software architecture concepts, like [45]–[47]. For more information on problem frames see [48]. Although these articles do not explicitly deal with the requirement elicitation or agile architecture modelling, they propose approaches for problem decomposition and architectural aspects. Thus they can be relevant for the proposed agile architecture-driven modelling and requirements engineering process, and, in particular, for the pattern and requirement engineering questions catalogues.

The approach presented in [49] proposes improving the software architecture evolution process by mining architecture and design patterns. Although the approach uses a similar idea of information extraction out of patters, its goal is to support the system's evolution and to extract general scenarios for it.

A practice-oriented approach is presented in [50]. It is based on Scrum, XP and sequential project management and requires an architect who plays a central role. Its main goal is architecture for communication, quality attributes and technology stack establishment purposes. It does not affect the

requirement engineering process. Design patterns are used to give a form to implementation, but not to support requirement elicitation and further architectural design. Beside that, the approach seems to be aimed for an organization having one software system type and thus similar project types (insurance software systems) as it is should use former architectural decisions.

A closely related article [51] states that a software architecture provides a basis for further requirement discovery and determination of the alternative design solutions. The proposed method is an adoption of the spiral life-cycle model and the author describes it as a complementary to XP. However, the requirements here emerge from: a) the view on models and prototypes b) from the commercially available software (COTS, commercial off-the-shelf software), which shall narrow the possible architectural decisions. The article mentions only on the reuse of COTS and requirement identification connected with them. Design patterns are mentioned for "fitting a component-based development approach into the development process". The article is on a very high level. It neither proposes an exact way on the analysis of the back flow information from patterns to requirements, nor gives exact details about the method itself.

VII. Conclusions and Future work

In this paper, I have discussed the role and potential benefits of architectural modelling in agile methods and presented a novel architecture-driven modelling and requirements engineering process. I argue, that such a process is an agile architectural modelling process, which constitutes a central element of an agile method which includes architectural modelling.

The future work will target the definition of a set of questions for driving requirements engineering, similar to the questions defined in my example application based on the MVC pattern, and the inclusion of architectural evaluation in order to deal with quality requirements in a better way. The validation of the overall process is still an open question. While its benefits are arguably clear (as discussed here), a rigorous empirical validation is still needed. Nevertheless, the process presented here is a promising starting point for a more comprehensive inclusion of architectural modelling in agile methods.

VIII. Acknowledgements

References

[1] M. A. Babar, "An exploratory study of architectural practices and challenges in using agile software development approaches," *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009.*, pp. 81 – 90, 2009.

[2] O. Salo and P. Abrahamsson, "Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum," *Software, IET*, vol. 2 , Issue:1, pp. 58 – 64, 2008.

[3] K. Silva and C. Doss, "The growth of an agile coach community at a fortune 200 company," *AGILE 2007*, pp. 225 – 228, 2007.

[4] P. Abrahamsson, M. Babar, and P. Kruchten, "Agility and architecture: Can they coexist?" *Software, IEEE*, vol. 27 , Issue:2, pp. 16–22, 2010.

[5] B. Drummond and M.-W. Chung, "Agile at yahoo! from the trenches," *Agile Conference, 2009. AGILE '09.*, pp. 113 – 118, 2009.

[6] S. Nair and P. Ramnath, "Teaching a goliath to fly," *In the proceedngs of Agile Conference, 2005.*, pp. 111 – 124, 2005.

[7] R. Rasmussen, T. Hughes, J. R. Jenks, and J. Skach, "Adopting agile in an fda regulated environment," *Agile Conference, 2009. AGILE '09.*, pp. 151 – 155, 2009.

[8] D. Tudor and G. A. Walter, "Using an agile approach in a large, traditional organization," *Agile Conference, 2006*, p. 373, 2006.

[9] B. Sheth, "Scrum 911! using scrum to overhaul a support organization," *Agile Conference, 2009. AGILE '09.*, pp. 74 – 78, 2009.

[10] C. Mann and F. Maurer, "A case study on the impact of scrum on overtime and customer satisfaction," *In the Proceedings of the Agile Conference, 2005.*, pp. 70 – 79, 2005.

[11] T. Dingsoyr, G. K. Hanssen, T. Dyba, G. Anker, and J. O. Nygaard, "Developing software with scrum in a small cross-organizational project," *Lecture Notes in Computer Science*, vol. Volume 4257/2006, pp. 5–15, 2006.

[12] K. Long and D. Starr, "Agile supports improved culture and quality for healthwise," *Conference Agile, 2008. AGILE '08.*, pp. 160 – 165, 2008.

[13] A. Cockburn and J. Highsmith, "Agile software development, the people factor," *Computer*, vol. Volume: 34 , Issue: 11, pp. 131 – 133, 2001.

[14] I. Gorton, *Essential Software Architecture*. Springer, 2006.

[15] R. C. Martin, *Agile Software Development. Principles, Patterns, and Practices*. Prentice Hall International, 2002.

[16] W. F. Opdyke, W. F. Opdyke, P. D, and R. E. Johnson, "Refactoring object-oriented frameworks," Tech. Rep., 1992.

[17] D. J. Paulish, P.Keil, and R. S. Sangwan, "Cost estimation for global software development," *International Conference on Software Engineering*, pp. 7–10, 2006.

[18] H. Koziolek, "Performance evaluation of component-based software systems: A survey." *Performance Evaluation, In Press,*, 2009.

[19] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, pp. 295–310, 2004.

[20] D. Garlan and B. Schmerl, "Aevol: A tool for defining and planning architecture evolution," *International Conference on Software Engineering*, pp. 591–594, 2009.

[21] S. Becker, M. Trifu, and R. Reussner, "Towards supporting evolution of service-oriented architectures through quality impact prediction," *In the Proceedings of the First International ARAMIS Workshop, L'Aquila, Italy*, 2008.

[22] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[23] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Pearson Studium, 2008.

[24] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Longman, Amsterdam, 2004.

[25] S. R. Palmer, M. Felsing, and S. Palmer, *A Practical Guide to Feature-Driven Development*. Prentice Hall International, 2002.

[26] J. A. Highsmith and K. Orr, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Co Inc.,U.S, 1999.

[27] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Agile manifesto," 2001, http://agilemanifesto.org/.

[28] H. Hulkko and P. Abrahamsson, "A multiple case study on the impact of pair programming on product quality," *In Proceedings of the Software Engineering, 2005. ICSE 2005. 27th International Conference*, pp. 495–504, 2005.

[29] S. Maranzano, J.F.and Rozsypal, G. Zimmerman, G. Warnken, P. Wirth, and D. Weiss, "Architecture reviews: practice and experience," *Software, IEEE*, vol. 22 , Issue: 2, pp. 34 – 43, 2005.

[30] P. Clements, F. Bachmann, L. Bass, and D. Garlan, *Documenting Software Architectures: Views and Beyond (SEI Series in Software Engineering)*. Addison-Wesley Longman, 2002.

[31] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 2005.

[32] P. Dissaux, M. F. Amine, and P. Michel, *Architecture Description Languages: IFIP TC-2 Workshop on Architecture Description Languages (WADL), World Computer Congress.* Springer Verlag, 2004.

[33] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. Volume 12 , Issue 6, pp. 42 – 50, 1995.

[34] D. Garlan, "Software architecture: a roadmap," *In Proceedings of the Conference on The Future of Software Engineering*, pp. 91 – 101, 2000.

[35] R. Beg, Q. Abbas, and R. P. Verma, "An approach for requirement prioritization using b-tree," *First International Conference on Emerging Trends in Engineering and Technology, 2008. ICETET '08.*, pp. 1216 – 1221, 2008.

[36] M. Ramzan, M. A.Jaffar, M. A. Iqbal, S. Anwar, and A. A. Shahid, "Value based fuzzy requirement prioritization and its evaluation framework," *Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009*, pp. 1464 – 1468, 2009.

[37] D. Port, A. Olkov, and T. Menzies, "Using simulation to investigate requirements prioritization strategies," *International Conference on Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM*, pp. 268 – 277, 2008.

[38] K. Logue and K. McDaid, "Handling uncertainty in agile requirement prioritization and scheduling using statistical simulation," *Conference Agile, 2008. AGILE '08.*, pp. 73 – 82, 2008.

[39] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009.

[40] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad, *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996, vol. 1.

[41] J. Gamma, Helm, *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1995, no. 1.

[42] S. Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* Wiley, 2002.

[43] M. Isham, "Agie architecture is possible - you first have to believe!" *Conference Agile, 2008. AGILE '08.*, pp. 484 – 489, 2008.

[44] D. Falessi, G. Cantone, S. A. Sarcia, G. Calvaro, P. Subiaco, and C. D'Amore, "Peaceful coexistence: Agile developer perspectives on software architecture," *Software, IEEE*, vol. 27 , Issue:2, pp. 23 – 25, 2010.

[45] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti, "Relating software requirements and architectures using problem frames," *In Proceedings of the IEEE Joint International Conference on Requirements Engineering, 2002.*, pp. 137 – 144, 2002.

[46] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh, "Architecture-driven problem decomposition," *In proceedings of the Requirements Engineering Conference, 2004. 12th IEEE International*, pp. 80 – 89, 2004.

[47] C. Choppy and M. Hatebur, D.and Heisel, "Architectural patterns for problem frames," *In the Proceedings of the Software, IEE*, vol. 152 , Issue:4, pp. 198 – 208, 2005.

[48] M. Jackson, *Problem Frames: Analysing & Structuring Software Development Problems.* Addison-Wesley Professional, 2000.

[49] L. Zhu, M. A. Babar, and R. Jeffery, "Mining patterns to support software architecture evaluation," *In proceedings of the Software Architecture, 2004. WICSA 2004. Fourth Working IEEE/IFIP Conference on*, pp. 25 – 34, 2004.

[50] J. Madison, "Agile-architecture interactions," *Software, IEEE*, vol. 27 , Issue:2, pp. 41–48, 2010.

[51] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34 , Issue:3, pp. 115 – 119, 2001.

A component concept for scientific experiments – focused on versatile visual component assembling

Kerstin Falkowski

Institute for Software Technology, University of Koblenz-Landau
Universitätsstrasse 1, 56070 Koblenz, Germany
falke@uni-koblenz.de

Abstract—**The intent of this PhD thesis is the specification, prototypical implementation and evaluation of a component concept for scientific experiments, focused on versatile visual component assembling by a human user, assisted by a comprehensive assembling environment. This is achieved by providing a visual assembling simultaneous from different viewpoints.**

I. Introduction

In the field of computer science most research areas comprise a *huge amount of data structures, algorithms and characteristic algorithm chains* providing a basis for the research area's tasks. These are normally used and combined consistently in different ways to solve new problems.

In established research areas there are standard libraries implementing those basic elements efficiently, best practices for their use possibly becoming manifest in patterns and/or even standard tools providing ready-made behaviour for all basic tasks of the research area. In relatively young research areas such standards do not exist, which unfortunately often leads to re-developments of basic data structures, algorithms and algorithm chains on the green field again and again.

The component concept targeted in this work provides a platform superseding this repeated re-development of algorithms and especially of algorithm chains. It provides rules for developing algorithms in a specific way, so that they constitute *components*. Built thereon it provides a comprehensive assembling environment, in which a human user can *assemble components visually to a more complex component* (corresponding to an algorithm chain). Of course, assembled components can in turn be part of even more complex components (hierarchic assembling).

An example for a relatively new research area is *image processing*. Here, two totally different kinds of tools are used for research. On the one hand there are open source C/C++ APIs, like the *Open Computer Vision Library (OpenCV)*[1], implementing basic data structures and algorithms in an efficient manner, but rarely more complex algorithm chains. On the other hand there is the commercial standard software *MATLAB Image Processing Toolbox (IPT)*[2], also providing basic data structures and algorithms, that can be combined to more complex algorithms. This is done textually in some editor in a closed system. A lot of image processing researchers test

their more complex algorithms in the MATLAB IPT before they implement them in C++ using some image processing API.

Using the planned component concept for scientific experiments, those researchers could implement basic image processing algorithms once as components and consistently use them for experiments via assembling them visually to arbitrarily complex components. In contrast to existing visual assembling environments, the assembling can be performed in a more versatile manner via a simultaneous assembling from different viewpoints.

The targeted component concept will be specified and implemented in a generic way, but evaluated using a large amount of well-defined image processing data structures and algorithms. Subsets of these data structures and algorithms are used as examples in the following.

This paper is structured as follows. Section II presents existing component concepts with interesting aspects for the planned component concept. Section III provides a detailed description of the targeted component concept and points out important research tasks. Section IV summarises the current state of the PhD thesis and describes future work including implementation and evaluation.

II. Component concepts

This section provides a short introduction to existing component concepts with interesting aspects for the targeted component concept. There are a lot of existing scientific and industrial component concepts and even more definitions of the term component. In line with the literature [1], [2] we define a *component* as a reusable software unit conforming to a *component concept* with a *well-defined interface*, encapsulating its realisation and possible internal states. The interface forms a *contract* between the component and its environment. It specifies which services a component is able to provide, if all demands of the component including required services are met by the environment (e.g. by other components).

Component assembling is the activity that distinguishes the component lifecycle from the common software lifecycle. It can be carried out from different behavioural *viewpoints*. A *component concept* has different purposes. At first it has to determine how components can be assembled and especially how they can be composed. This includes the component interface, the kind of assembling environment, the assembler

---

[1] http://opencv.willowgarage.com
[2] http://www.mathworks.com/products/image

and the viewpoints for assembling. Built thereon it has to specify the resultant component lifecycle in detail, including its activities, their possible order(s) and their corresponding actors. Based on this conceptual part, realisation techniques can be chosen and a specific environment for component development, assembling and/or execution can be developed. Finally, all this in conjunction determines the capabilities of the component concept's components.

There are two well known surveys of existing software component concepts. Lau and Wang [2] analyse 13 different scientific and industrial component concepts, compare them based on three different aspects (component semantics, component syntax and component composition) and classify them into four different categories according to component composition. *The Common Component Modeling Example (CoCoME)* [3] was a challenge, where 13 developers mapped a given component-based architecture example to their own scientific component concept. Goal of the challenge was making various component concepts comparable in a way. As a result the participating component concepts were classified into four categories, in this case according to their focus. In addition Selmat et al. [4] compare technical aspects of different scientific and industrial component concepts, namely their interaction mechanism and support for distributed applications.

A component concept for scientific experiments should provide the visual assembling of components that are already implemented and supplied (and potentially occur as binaries) from different viewpoints. For that purpose, there are two interesting groups of existing component concepts.

The first group contains component concepts, that allow the visual assembling of already implemented and supplied components, which Lau and Wang classify in the category 'Deployment with Repository'. Examples are *JavaBeans* [5] combined with a suitable assembling environment like *JBeanStudio*[3] and *ConQAT* [6]. In both component concepts functionality is implemented in atomic components in terms of Java classes with specific properties, that are specified, developed and supplied at development time. These can be visually assembled to a composite component and directly executed in a specific assembling and runtime environment. In JavaBeans the composition "glue code" is stored in an adapter Java class, in ConQAT the composition information is saved as composed component in an own XML dialect (.cqb). In contrast to JavaBeans, ConQAT supports hierarchical component composition. Unfortunately both component concepts compose components only from one fixed viewpoint, JavaBeans from an *event-driven* viewpoint and ConQAT from a *data-flow* viewpoint.

The next interesting group contains component concepts, that model component-based architectures of complete applications from different viewpoints at design time, which form the CoCoMe category 'formal models, focusing on behaviour and quality properties'. Examples are *SOFA 2.0* [7], *Fractal* [8] and *Palladio* [9]. In all three component

concepts a component is defined equivalent to the definition mentioned above. In SOFA and Fractal structural information about a component-based architecture is modelled conform to an *Ecore*[4]-metamodel and the behaviour of components is described by (extended) *Behaviour Protocols (BP)*. In Palladio four different views to a component-based architecture can be modelled, conforming to an *Ecore*-metamodel, and additionally performance relevant aspects for each of these models can be specified via *Service Effect Specifications (SEFFs)*. SOFA 2 and Fractal explicitly separate functional and control parts of components. All three provide hierarchical component composition. In Fractal and Palladio components are composed via interfaces bindings, in SOFA 2 via connectors offering different communication styles. Fractal provides shared components, component instances that are is subcomponents of several composite components. SOFA 2 and Palladio provide performance prediction at design time. Fractal provides the verification of composition correctness as well as of correspondences between code and behaviour specification. All three component concepts provide modelling tools, SOFA and Fractal additionally provide APIs for supporting the (top down) implementation of designed architectures. However, all three component concepts are originally modelling approaches for complete component-based architectures and do not provide the visual assembling of already implemented and supplied components.

III. A COMPONENT CONCEPT FOR SCIENTIFIC EXPERIMENTS

In this section the targeted component concept for scientific experiments is described in detail. As mentioned before, it offers the *visual assembling of components by a human user, assisted by an assembling environment*. In comparison to the other mentioned visual assembling environments in Section II, the adaptation and especially the composition of components shall be much more versatile. This is achieved by providing *assembling from different viewpoints*, namely data-flow, service-usage and control-flow.

This leads to three different kinds of component composition (Section III-D). On the one hand two components can be composed, if a datum of a component is required by the other component. This component composition constitutes a *data-flow*, similar to component composition in ConQAT. On the other hand, two components can be composed, if a service of a component is required by the other component. This component composition constitutes a *service-usage*, conforming to the common view on component composition, e.g. in UML 2.2 components. Moreover a component or one of its services can be an arbitrarily complex control structure. A component composition with such "control components" constitutes a *control-flow*.

Accordingly, a *component* may require various *input data* from its environment to perform its task and it can provide

---

[3]http://www.vislab.usyd.edu.au/moinwiki/JBeanStudio

[4]http://www.eclipse.org/modeling/emf

distinct *output data*. Moreover a component may require *services of other components* to perform its task and can provide at least one *own service* to other components. So, the *interface of a component* has to offer different kinds of *ports*[5] for receiving input data and service access from the environment and sending output data and access to its own services to the environment. The mentioned control components can be used in the internal structure of a component but they do not have an explicit impact on the interface of a component.

A. Component interface

A component comprises any number of input ports. *Data input ports* facilitate the receiving of input data from the environment. *Service input ports* enable the access to services of other components. Every input port has a *multiplicity* $m..n$ with $m, n \in \mathbb{N}$ and $n > 0$, that decides how much inputs from different sources the port can receive at runtime. Input ports whose lower bound m is 0 are called *optional input ports*, if m is greater than 0 the input ports are called *obligatory input ports*. Input ports whose upper bound n is 1 are called *single input ports*, if n is greater than 1 the input ports are called *multi input ports*.

If a datum/service is obligatory for a component to perform its task, its interface offers an obligatory service port, if the datum/service is optional, the interface offers an optional service port. Usually a component interface offers an optional data input port for a parameter, for which the component has a default value/object, that can but does not have to be changed by a component assembler. If two or more input data/services of the same type have a specific role for a component, its interface offers several single input ports. If the data/services have no specific role, the interface offers a multi input port.

An input datum can be passed directly to a component via component adaptation (Section III-C). Moreover, input data/services can be sent/offered to a component by another component at runtime, where the source component is determined during component composition (Section III-D).

Moreover, a component interface may comprise any number of *data output ports*, each of them facilitating the sending of exactly one output datum to the environment during runtime. One can distinguish between two different kinds of output data and thus between two distinct kinds of data output ports: If an output datum is a constant of a component, its interface offers a *constant data output port*, that is able to send the output datum anytime during runtime. If an output datum is a result of the component's task, its interface offers a *resultant data output port*, that is not able to send the output datum to the environment until the component performed its task.

A component interface comprises one *service output port* for every service of the component, offering access to this service to other components during runtime. A service output port is not able to offer access to its service until the component received all input data and/or services obligatory for the service.

Output data/services can be sent/offered by a component to any number of other components at runtime, where the target components are determined during component composition (Section III-D).

Every port has a *defined type* that is decided during component development by a component developer and changed nevermore. All data ports beside constant data ports have in addition a *current type*. After component instantiation this is the same as the defined type, but during component assembling it can be changed to any subtype.

There can be dependencies between data and thus data ports of one component, affecting its adaptation and/or composition from a data-flow viewpoint. As an example the current type of an data input or output port can depend on the current types of one or more other data input ports. Such dependencies should explicitly be offered by the component's interface to its environment in a formalised form, so that an assembling environment as well as a component assembler can make use of them. A research task in this context is the specification of a suitable concept for the formalisation and persistent storage of those dependencies at development time and their usage at assembling time and runtime.

A component interface shall be able to describe a component and its services as accurately as possible. This information can help an assembling environment to disallow a syntactically wrong assembling, and it can help a component assembler to potentially even avoid a semantically wrong assembling. Moreover detailed information can help to decide, if a component is equivalent to another component and can be substituted by it. So, the assembling environment shall be able to forward as much information about the component as possible to the component assembler. Orth [10] gives a good overview about distinct information for component description.

a) Example.: In the following, some characteristic image processing operations are shortly introduced and conceptually modelled as exemplary components conform to the described component interface. The components are modelled via the use of UML component / composite structure diagrams in a specific way.[6]

An component offers its own services via UML provided interfaces in ball-notation on its right upper corner. A component that requires access to the services of other components possesses UML required interfaces with the names of the services in socket-notation on its left upper corner. For the visualisation of component composition from a data-flow viewpoint, UML ports are modified to data ports, rather functioning like UML pins. Input data ports are always displayed on the left side of a component, data output ports are always displayed on its right side. Constant data output ports are dark grey, in contrast to all other data ports, that are grey.

Figure 1 shows some components, as a base for component composition. `RGBImageReading` is a component, that

---

[5]Here port in terms of entry and exit is meant, not port in terms of UML.

[6]Using the IBM Rational Software Architect http://www.ibm.com/software/awdtools/architect/swarchitect.
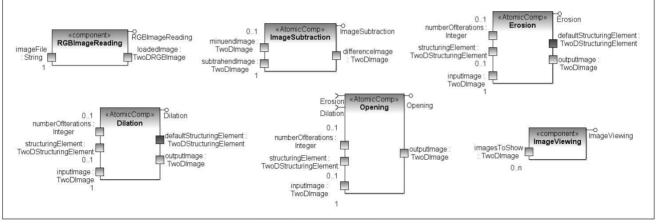
Fig. 1.   Basic components.

loads an image from a file. It requires a String containing path and name of the image as obligatory input datum and delivers it as 2d RGB image as resultant output datum. `ImageSubtraction` is a component that subtracts one image element-wise from another image. It requires two 2d images of the same type and size as obligatory input data and delivers a 2d image of the same type and size as resultant output datum. To deliver an image of the same type, the image values of the resultant `differenceImage` have to be normalised. `ImageViewing` is a component that visualises arbitrarily images. It can get any number of images as input data and delivers no output data. `Erosion` takes for every output image value the minimum of neighbouring values of the input image. `Dilation` performs a contrary operation using the maximum of neighbouring values. The concrete neighbouring values are in both cases determined by a structuring element. `Opening` performs an erosion to an image followed by a dilation. All three components require a 2d image as obligatory input datum and deliver a 2d image of the same type as resultant output datum. Optionally they can get the number of iterations in terms of an integer $n > 0$ and a 2d structuring element as optional input data. `Erosion` and `Dilation` provide their default structuring elements as constant output data.[7] `Opening` additionally requires the services `Erosion` and `Dilation` from its environment.

B. Component instantiation

A component assembler has to instantiate a component, before it can be assembled. The assembling environment should facilitate a *visual component instantiation* via drag & drop from a list of existent components to an assembling canvas.

After it was instantiated a component has a state, that can be changed by component adaptation. A is the decision how instantiated components and their adapted values/objects are stored, directly via serialising component instances or indirectly in specific external files.

C. Component adaptation

Via *visual component adaptation* a component assembler can directly pass input data to instantiated components.

For that purpose the assembling environment should provide specific *adaptation GUIs*. Such an adaptation GUI has to present the current assembling state of a component, including already adapted and/or composed input data, and it has to facilitate the change of input data values regarding potentially existent dependencies between the component's data. Thereby it has to *disallow an incorrect adaptation*, e.g. setting a value of a wrong type or outside the correct range to an input datum and so forth.

An adaptation GUI has to be generated by the assembling environment on demand based on a component interface. To enable the automatic generation of adaptation GUIs, there have to be *predefined GUI elements* for all input data types of a component. These GUI elements can be offered by the component concept as part of the assembling environment or developed by a component developer. The component concept should provide GUI elements for all atomic types as well as for often used complex standard types, like e.g. `File`. A component developer should be able to add GUI elements for any type that is used by a developed component.

One important research question in this context is, if an additional visual component adaptation during runtime shall be enabled. A further research task is the specification of a suitable concept for the development and integration of type-specific GUI elements at development time and their automatic composition to adaptation GUIs at assembling time and runtime. Here related work concerning GUI builders, like the *Eclipse Visual Editor project*[8] or parts of the JavaBeans API, partly examined in [10], has to be regarded.

D. Component composition

Via *visual component composition* a component assembler can determine, which instantiated components send/offer what data/services to which other instantiated components at runtime.

---

[7]It is a design decision of the component developer, that the opening operator has no default structuring element as constant output datum.
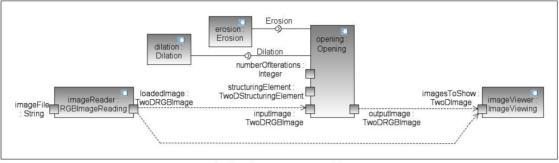
[8]http://www.eclipse.org/vep/

Fig. 2.   Component composition.

The assembling environment should facilitate a component composition via drawing an arrow from an output port of a component to a corresponding input port of another component. Thereby, it is responsible for *disallowing an incorrect composition*, e.g. the composition of two ports of the same component, the composition of two input ports or two output ports, the composition of a data port and a service port, the composition of two ports with incompatible types and/or a cyclic data-flow composition of two or more components and so forth.

A component A can use another component B in two different ways. A can use B *directly* via receiving one or more of its resultant data after B performed its task. Or A can use B *indirectly* via getting access to one or more of Bs services after B received all required input data/services. These two kinds of usage should not be combined, because for an indirect use B has to receive potentially less input data/services than for a direct use. For that purpose an assembling environment could prohibit a simultaneous usage in both ways. This means, if at least one of the resultant output ports of a component is composed, its service output ports can not be composed and vice versa. They can be disabled or even made invisible.

There is a further research task concerning component composition in general, that affects realisation of the component concept but is not only a realisation problem. It has to be decided, if data/services are passed from one component to another during execution by value/copy or by reference. Or rather, in which cases data/services are sent by value/copy and in which cases by reference. And, who determines, if data are sent by reference or by value/copy: the component developer, the component assembler and/or the assembling environment.

b) Data-flow.: During component assembling components can be *composed from a data-flow viewpoint* via their *data ports* (Section III-A). Such a data flow composition means, that at runtime one component sends a specific datum to another component.

Every data output port of a source component can be connected to any number of data input ports of target components, if their types are *compatible*. For a compatibility checking, the data port's *current type* is used and can be changed due to composition. Only for a constant data output port, the defined type is used, because it has no current type. Usually one would expect, that an output port is compatible to an input port, if the output port type is *the same or a subtype* of the input

port type. This statement is correct, but can be *incomplete*. As mentioned before, there can be dependencies between the ports of one component, and these dependencies can lead to further cases of compatibility via changing the current types of depending ports.

Hence, further research tasks are the analysis of the impact of port dependencies on a data-flow component composition and the development of an algorithm that checks the compatibility of two data ports and determines which current types have to be changed in which way to offer the data port composition. Here, related work dealing with *type inference* [11] has to be regarded.

c) Service-usage.: During component assembling components can be *composed from a service-usage viewpoint* via their *service ports* (Section III-A). Such a service-usage composition means, that at runtime a component's service is used by another component.

Every service output port of a source component can be connected to any number of service input ports of target components, if their types are *compatible*, which means that the output port type is *either the same or a subtype* of the input port type.

Usually one would expect, that a component can offer its services anytime during runtime. This is because, a target component uses a service internally and is itself responsible for providing all required input data and/or services to the service before use. But this statement is only true, if the two service ports have exactly the same type. If a service output port type is a subtype of the service input port type, this subtype can specify additional obligatory input data and/or services that the target component does not know. In this case, the source component is not able to provide the service to the target component, until it received those additional obligatory input data and services.

d) Control-flow.: During component assembling, components can also be assembled from a *control-flow viewpoint*, because *every component is itself a control structure*.

Every component affects the successive program flow during runtime *implicitly* in different ways. Usually a component assembler does not really know, how the adaptation and/or composition of a component correlates to the resulting program flow at runtime. But if this knowledge is made explicit during assembling time, it can facilitate an *explicit* control-flow component composition.
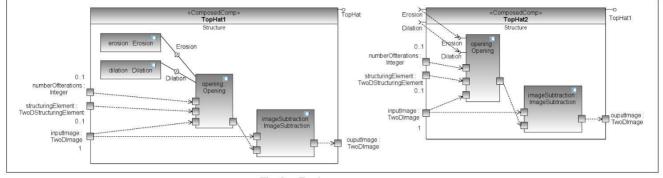
Fig. 3. Top hat components.

On the one hand, an assembling environment can provide *ready-made generic control components* for different kinds of loops and choices. So, a research task is to identify important control structures and develop them in a generic way, conforming to the component concept. Here, existing *visual languages dealing with control flow* like UML activity diagrams [12] and languages for exogenous connectors like Reo [13], where atomic 'channel types' can be hierarchically composed to more complex connectors, have to be regarded.

On the other hand, a component developer can develop *arbitrarily complex domain specific control components* for his research area. Hence, a research task is to analyse the impact of such control structures to the program flow and summarise the results in terms of guidelines for domain specific control component development.

A control component decides, how often the services of its composed components are used or rather how often its composed component are executed.

e) Example.: Figure 2 shows a set of assembled components, not (yet) saved as composed components. A component composition from a data-flow viewpoint is visualised as a connection between an output port of a component and an input port of another component. A component composition from a service-usage viewpoint is visualised as an assembly connection in lollipop notation. Because the RGBImageReader delivers a 2d RGB image, the input image port and output image port of the Opening component changed their types from TwoDImage to the subtype TwoDRGBImage.

E. Component storage

A component assembler can *store a set of assembled components persistently as a new component.*

A set of assembled components is *storable*, if all components are able to either perform their task or offer their services at runtime. This is the case, if at storage time none of them possesses an obligatory input port, that is neither adapted nor composed. If there are still unassembled obligatory input ports, they can be added to the interface of the new component. Then the required data and/or services have to be provided from the environment of the new component at runtime and are internally passed to the contained components.

The assembling environment should facilitate the storage of a component via one click. Thereby it is responsible for *disallowing the storage of incomplete components*. This can be achieved by automatic addition of unassembled obligatory input ports to the new component interface.

Every stored component can again be loaded into the assembling environment for further (re-)assembling or instantiated to be used in another component. Moreover it can be executed, if its interface comprises no unassembled obligatory input ports (Section III-F).

f) Example.: Figure 3 shows two different component variants of the morphological operation TopHat, that subtracts the opening of an image from the original image. Both top hat operators compose the components ImageSubtraction and Opening internally. TopHat1 delivers the services Erosion and Dilation to Opening internally via the components Erosion and Dilation. TopHat2 delivers the services Erosion and Dilation to Opening via requiring them from the environment itself. Both top hat components require a 2d image as obligatory input datum and deliver a 2d image of the same type as resultant output datum. Optionally they can get the number of iterations in terms of an integer $n > 0$ and a 2d structuring element as optional input data. In both components the dependencies between their data input and output ports can be derived from the internal modelling.

F. Component execution

A component user can initiate the execution of an executable component by the assembling environment. A component is *executable*, if its interface comprises no unassembled obligatory input ports. For execution it has to be loaded into the assembling environment.

The assembling environment controls the execution and is responsible for a correct performance. It carries out the following evaluation algorithm:

1. Pass all received input data to contained components.

2. Send constant output data from all contained components to connected contained components.

3.a Execute all contained components, that are able to perform their task now, and send resultant data to connected contained components.

3.b Offer service access from all contained components, that are able to offer their service now, to connected contained components.

4. Repeat step 3 until there are no further components that can be executed or can offer their service.

5. Send resultant data and offer services to connected components.

G. Component concept realisation

The component concept will be realised using *Java* as well as *TGraphs* [14], directed graphs whose vertices and edges are typed, ordered and attributed, and *JGraLab*[9], an API for TGraph processing.

During component development the component concept explicitly distinguishes between two kinds of components. An *atomic component* is developed by a component developer at development time. It can be supplied as .java and/or .class file, packed in a .jar archive, depending on how much information the component assembler/user shall get about the atomic component's internals. A *composed component* is developed by a component assembler during assembling time in the assembling environment. Internally it is modelled as TGraph instance conforming to a specific TGraph schema for composed components. It can be persistently stored as .tg file, packed in a .jar archive, and again loaded from this .tg file. But both kinds of components have *exactly the same interface* to their environment during visual component assembling. The *defined type* of a data port can be *any Java type*. The *defined type* of a service port has to be a subtype of *a specific Java type* constituting the supertype for all components.

An atomic component is instantiated via instantiation of the corresponding Java class and a composed component is instantiated via loading the corresponding TGraph into Java and instantiating all contained components. An atomic component is executed via calling the `execute()`-method of its corresponding Java class. A composed component is executed by executing directly used contained components.

For the development of adaptation GUIs and adaptation GUI elements potentially JavaBeans *property editors and/or customisers* [5] can be used. For the formalisation of dependencies between data ports, potentially the *Java Modeling Language*[10] can be used.

The mentioned image processing components for evaluating the component concept shall use existing image processing libraries. Haas [15] gives a good overview about those libraries.

IV. SUMMARY

As mentioned at the beginning, the intent of the PhD thesis is the specification, prototypical implementation and evaluation of a component concept for scientific experiments. In section III a detailed description of the targeted component concept and predicted research tasks have been introduced. The research tasks comprise:

1. a) The specification of a suitable concept for the formalisation and persistent storage of dependencies between data ports of one component at development time and their usage at assembling time and runtime. b) The analysis of the impact of those port dependencies on a data-flow component composition. c) The development of an algorithm that checks the compatibility of two data ports and determines which current types have to be changed in which way to offer the data port composition.

2. a) The identification of important control structures and their development as ready-made generic control components conforming to the component concept. b) The analysis of the impact of arbitrarily complex domain specific control structures on the program flow and the summarisation of the results in terms of guidelines for the development of such control components.

3. The specification of rules that determine, in which cases data/services are passed from one component to another during execution by value/copy and/or by reference, and/or rules that determine, which actor takes this decision.

4. a) The determination, how instantiated components are stored. b) The decision, if components can additionally be adapted at runtime. c) The specification of a suitable concept for the development and integration of type-specific GUI elements at development time and their automatic composition to adaptation GUIs at assembling and runtime.

In future work, these research tasks have to be performed to complete the specification for the planned component concept for scientific experiments. Based thereon the component concept can be prototypically implemented and evaluated using a large amount of image processing data structures and components[11].

REFERENCES

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, (first edition 1997).

[2] K.-K. Lau and Z. Wang, "Software Component Models," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, 10 2007.

[3] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, *The Common Component Modeling Example: Comparing Software Component Models*. Springer Publishing Company, Incorporated, 2008.

[4] M. H. Selamat, H. Sanatnama, R. Atan, and A. A. Abd Ghani, "Software Component Models from a Technical perspective," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 7, no. 10, pp. 135 – 147, 10 2007.

[5] G. Hamilton (Editor), "JavaBeans," Sun Microsystems, Tech. Rep., 08 1997, version 1.01-A. [Online]. Available: http://java.sun.com/javase/technologies/desktop/javabeans/api

[6] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka, "Tool Support for Continuous Quality Control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, 2008.

[7] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 40–48.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[9] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, 2009.

---

[9]jgralab.uni-koblenz.de

[10]http://www.eecs.ucf.edu/~leavens/JML

[11]The planned component concept is developed in the context of the project *Software Techniques for Object Recognition (STOR)*, http://er.uni-koblenz.de, whose task is the use of software engineering techniques in image processing.

[10] S. Orth, "Entwicklung eines Konzepts zur Selbstauskunftsfähigkeit für STOR-Komponenten," Master's thesis, Universität Koblenz-Landau, 12 2009.

[11] J. Plevyak and A. A. Chien, "Precise concrete type inference for object-oriented languages," in *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*. New York, NY, USA: ACM, 1994, pp. 324–340.

[12] "UML 2.0 Superstructure Specification," Object Management Group (OMG), Tech. Rep., August 2005.

[13] F. Arbab, *Reo: a channel-based coordination model for component composition*. New York, NY, USA: Cambridge University Press, 2004, vol. 14, no. 3.

[14] J. Ebert, V. Riediger, and A. Winter, "Graph Technology in Reverse Engineering, The TGraph Approach," in *10th Workshop Software Reengineering (WSR 2008)*, ser. GI Lecture Notes in Informatics, R. Gimnich, U. Kaiser, J. Quante, and A. Winter, Eds., vol. 126. Bonn: GI, 2008, pp. 67–81. [Online]. Available: http://www.uni-koblenz.de/~ist/documents/Ebert+2008GTI.pdf

[15] J. Haas, "Analyse, Evaluation und Vergleich von Bildverarbeitungsbibliotheken aus Sicht der Softwaretechnik," Master's thesis, Universität Koblenz-Landau, 4 2009.

Performance Prediction for Highly Distributed Systems

Jörg Henss

Institute for Program Structures and Data Organisation

Faculty of Informatics

Karlsruhe Institute of Technology, Germany

Email: Henss@kit.edu

Abstract—Currently more and more highly distributed systems emerge, ranging from classic client-server architectures to peer-to-peer systems. With the vast introduction of cloud computing this trend has even accelerated. Single software services are relocated to remote server farms. The communication with these services has to use uncertain network connections over the internet.

Performance of such distributed systems is not easy to predict as many performance relevant factors, including network performance impacts, have to be considered. Current software performance prediction approaches, based on analytical and simulative methods, lack the support for detailed network models. Hence an integrated software and network performance prediction is required.

In this paper general techniques for the model integration of differently targeted simulation domains are presented. At plus design alternatives for the coupling of simulation frameworks are discussed. Finally this paper presents a model driven approach for an integrated simulation of software and network aspects, based on the palladio component model and the OMNeT++ simulation framework.

I. Introduction

Distributed systems have a long tradition in computer system design. This includes client server, peer-to-peer, and other systems that communicate using networks. We use distributed computer systems in our daily life as for example a super-market point of sale will communicate with the warehouse management system, the credit card clearing house, and some loyalty program system.

Highly distributed systems are distributed systems that use a lot of communication and consume many remote services. A currently emerging form of highly distributed systems is the so called cloud computing. Cloud computing [1] stands for the displacement of actual running computer applications from a users machine or corporate servers to a remote distributed system, providing superior computing power at lower costs. Services that used to reside on the same server are distributed on different machines, that even might be miles away, when using cloud computing.

During software development in general and especially for distributed systems, it is desirable to know whether certain performance constraints, like response time or throughput, will be met. Software performance prediction allows for an ahead of time estimation of these performance characteristics. Furthermore the performance information also assist to choose the right dimensioning of hardware. For distributed systems, the communication between systems can be a major performance factor as data has to be interchanged. This is not only especially true in the high performance computing domain, where large efforts are spent on minimising the communication costs, but also for traditional client server and other distributed software architectures. One of the problems network communication has, is the use of a shared media, as this can lead to collision and contention problems. These contention effects can have a large impact on the overall performance of a system.

Several architectural design decisions can be made to optimise distributed systems. For example the introduction of caches can reduce communication at the cost of memory. Furthermore network compression and quality of service based prioritisation can be applied as well, but require more CPU usage or more expensive hardware. A performance prediction can help to choose the best suited solution and to find a good trade off between alternatives.

It is a matter of discussion how the performance of such distributed systems can be well predicted. On the one hand current software prediction methods only have limited support of network communication prediction. On the other hand network simulation can be used to predict the performance of network communication, but those simulation frameworks have no simple means to model software systems. Thus a combination of both simulative approaches seems to be a solution to this problem. However it is unclear which techniques can be used to accomplish this integration of simulations.

We recognised that some kind of integration has to be made both on the model layer and on the simulation layer. The integration techniques on the model layer have to combine the different simulation models, while the integration on simulation layer describes styles of coupling between the simulation frameworks themselves.

The contribution of this paper is twofold. First we introduce a reference architecture for the model integration and the coupling of simulation frameworks in general and describe certain design alternatives. Secondly we describe a concrete integration architecture that uses model driven techniques to combine a software performance model with a network simulation model. This concrete architecture can be used to predict the performance of highly distributed systems in an

early design stage. In our approach the palladio component model is used as software performance model and is combined with the OMNeT++ network simulation framework that allows to model and simulate detailed network models.

The paper is structured as follows. Section II describes the the foundation of discrete-event simulation in general and software and network performance simulations in special. In Section III several techniques are introduced for a simulation model integration. Methods for coupling of simulation frameworks are introduced in Section IV. Our concrete model driven simulation integration approach is presented in Section V. Related work is presented and discussed in Section VI. Section VII concludes this paper and describes the future work.

II. DISCRETE-EVENT SIMULATION

There exist a large amount of analytical approaches that use formal methods like petri-nets, markov-chains, or queuing networks to calculate different measures of a system. However these analytical methods can not be used for analysing large complex systems without introducing strong constraints (e.g. no mixed scenarios are supported). Thus it is necessary to use a simulative approach for those complex systems.

There are many different techniques for the simulation of systems. Nevertheless the discrete-event simulation approach is among the most popular and wide spread simulation techniques. A large number of discrete-event simulation frameworks are available, both commercially and as open source.

A. Simulation Concepts

Though different simulation frameworks use differing terminologies, the description of discrete event simulation can be stripped-down to the following concepts as described by Banks et al. [2]

A *system* is the set of all entities that interact together over time. These entities usually have a goal they want to achieve. The so called *system state* is a set of variables that are sufficient to describe a system at any given time. As an illustration we will regard a supermarket as a system.

The *model* is an abstraction of a system. The system is described by structural, logical, and mathematical information. This information is represented by entities, events, activities and further state information. A model will usually build upon a meta-model that is fitted to the modelled domain.

An *entity* is an object in a system that is individually represented in the model. Entities can have *attributes* that can change over time. Often entities are organised in a hierarchical way. A supermarket has customers, goods, etc. as entities.

Among the entities, there are special entities that are called *active* and *passive resources*. Active resources are used to execute an *activity*. Usually only one activity can be executed by an active resource at the same time. Thus some kind of scheduling policy is required when several activities are waiting for execution at an active resource. Common scheduling strategies are first-in, first-out (FIFO), last-in, first-out (LIFO), time sharing, or ranked by any attribute of the entities like priority. Some times active resources are also called *service center*.

Passive resources are entities that have to be *acquired* to achieve a goal, e.g. the execution of an activity. The number of passive resources in a system that can be acquired is usually limited, dependent on the kind of resource. Consequently a passive resource has to be *released* after a corresponding goal is achieved. An acquisition of a passive resource blocks until a resource is available.

In the supermarket example, the checkout is an active resource as the payment activity is executed here, while the shopping cart is a passive resource that has to be acquired before a shopping activity can start. After an shopping activity has ended the cart is released and a new customer can use it.

Events are triggers that usually change the state of a modelled system. In discrete-event systems an event can be scheduled to occur in the future. Events succeed each other in time.

As input to the simulation a so called *workload* is used. The workload starts so called jobs, which initially trigger different events and are active until the reception of an event that indicates the end of the job. Usually a stochastic process is used that determines the occurrence of a certain job at an instance of time. Commonly a poisson process is used. Workloads can be open or closed. A closed workload allows only a limited number of jobs to be active and is characterised by a thinking time that determines the time between end of a job and the start of the next job. An open workload is characterised by inter-arrival times of jobs and has no limit on the number of active jobs. Additionally a workload can be based on measured data that is used as input. In our example the workload would generate customers that enter the supermarket.

An *activity* describes a step that takes a certain time to accomplish. This time can be a deterministic value or a statistical value based on a distribution function. Moreover the time can have functional dependencies on the current state.

The time an activity takes is often called *service time*.

A *delay* is a time span that is not priorly specified, but is a results of the current system state. Delays occur when passive resources can not be acquired instantly and when an active resource has several activities that have to wait until they are executed. The average delay is an important measure of a system.

The *clock* in a discrete-event simulation is a variable that represents the simulation time. It is used to express the progress of time while the simulation is running. The progress of time can be synchronous, i.e. the time is constantly increased by a fixed span. However usually an asynchronous time is used, that allows the simulation to skip in time to the next occurrence of an event. A common stop criterion is the arrival at a specific simulation time. For further explanation on discrete-event simulation techniques cf. [2].

There exist a vast amount of discrete-event simulation frameworks, both commercial and open source. Typically every simulation that is to be carried out uses a specific
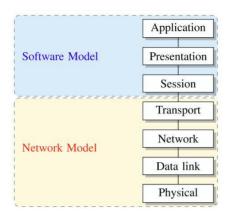
Fig. 1. OSI stack with the network model and the software model.

simulation framework that is fitted to its domain. Mostly theses framework are composed of a simulation core that is used for managing the basic simulation concepts like time, events, and entities. Furthermore the frameworks support the collection of statistical data and event trace data for the evaluation of results.

Another simulation approach is the so called process-oriented simulation, where simulated system activities are mapped to processes. A process represents the life cycle of an entity and can interact with other processes using messages. Shared resources are used to simulate active and passive resources and can cause a process to block and wait for execution or acquisition. This process based view is often regarded as being more intuitive than the event based view. Nevertheless process-oriented simulations often have a limitation on the size of the system that can be simulated and "are known to be slower than discrete-event simulations" [2, p. 537].

On top of simulation frameworks, there usually exist domain specific elements that build upon the basic building blocks and allow for a more convenient definition of simulation scenarios. In most cases one can find an existing simulation framework that already supports most of the required functionality. The domain specific elements usually can be extended additionally, to further customise a simulation. Two distinct domains of discrete-event simulation are the simulation of networking systems and software systems, which will be discussed in the following.

B. Network and Software System Simulation

The domain of network and software systems have much in common, as networks are using software at different levels and software is often using networks for communication. Nevertheless both domains are usually regarded with different focuses. Figure II-B shows the layers of the OSI reference model where software and networks are shown in context. The software domain is mostly situated in layers 5 to 7 while the network domain is settled in layers 1 to 5.

Network simulation usually have a focus on the communication of nodes and the protocols that are used. The communication can be simulated on different levels of abstraction. Common network simulators often simulate on packet or session level. That means the simulation uses packets or sessions as smallest separate entities. Nevertheless there also exist simulators that operate very low on bit level or very high on the system level, where whole networks are communicating. Usually network simulators provide build in high level concepts to model the network nodes. Every node in the network usually has some kind of protocol stack depending on the level it operates on. Stochastic channels are used to model uncertain connections. This allows to simulate loss, jitter, and distortion of transferred data. The main characteristics that are measured are throughput and latency. Though network simulations are also used to validate the pure functionality of protocols.

Prominent network simulation framework are the ns-2 [3] and OMNeT++ simulation frameworks [4]. Both provide a large collection of already implemented network protocols and implementations of network channels. Furthermore rudimentary application entities are built in that represent sinks and sources for protocols like FTP or HTTP. Nevertheless both have in common that advanced behaviour on application layer has to be implemented as code manually.

In this paper the OMNeT++ network simulation framework will be used as ns-2 is not maintained anymore and the new developed ns-3 is not yet matured. OMNeT++ supports discrete-event and process based simulations. Furthermore there exists a GUI for modelling network topologies.

The OMNeT++ network meta-model includes entities that model queues, channels, messages, and packets. So called *cModules* are used to model network nodes. Furthermore cModules can be composed of other modules and are than called *cCompoundModules*. Thus cModules are a kind of simulation components. The support for IP networks in OMNeT++ is given by the INET framework for OMNeT++ [5] that includes the implementations of a large number of IP based network protocols. These protocols are as well defined using the cModule structure. A cModule can have parameters that are used to configure it.

The OMNeT++ framework includes a language for the definition of the network topology, the cModule assembly and the cModule parameters. This language is called *NED*. The NED language is used to define OMNeT++ models for simulation.

An OMNeT++ simulation model is compiled to an executable using a specialised pre-processor for the gcc compiler [6]. When run this executable offers a GUI to control the resulting simulation.

The simulation of software systems to predict their performance and other quality attributes is a well-known research field. The results of simulations can be used to rate and find trade-offs between alternative software architectures, choose an accurate hardware platform, and to predict whether certain quality constraints will be fulfilled. A software simulation model often is based on design-level models like UML, automata, or control flow diagrams and can be used when

annotated with information about the quality attributes. The UML SPT [7] and MARTE [8] profiles are examples for such UML annotations. The design model usually is transformed to a simulation model that only contains simulation relevant information and has the quality attributes embedded.

There also exist specialised models that support the specification of quality attributes directly. One of these models is the Palladio Component Model [9]. It allows for the modelling of component based software systems with a focus on the prediction of performance and other Quality-of-Service attributes. The behaviour of the modelled components can be specified using so called *Service effect specications* (SEFF). For every method in the provided interface of a component a SEFF can be defined. A SEFF includes several constructs that represent the control flow of the method. Moreover a SEFF can make (external) calls to other methods that are part of the component's required interface.

The so called *Resource Demanding Service Effect Specications* (RDSEFF) allows the definition of (active) resource demands in a SEFF. These resource demands can be modelled as stochastic expressions. A stochastic expression usually includes a stochastic distribution function and can have parametric dependencies. A parameter is characterised by a number of so called variable characterisations that define e.g. the size, structure, or value. The variable characterisations for parameters used to solve the parametric dependencies are stochastic expression themselves and are either defined in the SEFF or are part of the method signature and handed over when the method is called. Furthermore there also exist passive resources in the PCM. Thus a RDSEFF can contain acquire and release methods for these resources.

In the palladio component model components can be deployed to systems. A system also provides the active and passive resources that are used in the RDSEFFs. When more than one system is modelled those systems have to communicate using modelled network connection. In the PCM a network connection has attributes that define the throughput and latency.

The palladio component model can be used to make analytical and simulation based predictions. To execute one of these prediction methods the model is transformed using model-to-model or model-to-text transformations. The current simulative prediction approach is called *SimuCom* and uses a process based approach that is described in [10]. SimuCom builds up on the Desmo-J framework [11] for the modelling of queues and events. Currently SimuCom has a very limited support of networks. When a palladio component model includes two components that are communicating over a network connection, SimuCom uses an active resource with a FIFO scheduling strategy to represent this network connection.

The integrated simulation approach presented in this paper will be based on the palladio component model.

Further software simulation approaches using UML and other models can be found in sect. VI.

III. SIMULATION INTEGRATION ON MODEL LAYER

As mentioned before every simulation uses a simulation model to describe entities, resources, events, workload, and activities. The simulation model can additionally have several higher level domain specific elements that ease the definition of simulation scenarios.

For network simulations these elements are network nodes and connections based on some kind of network protocol. At plus common devices will be presented, that represent switches, servers, etc.. Software simulations usually will include elements like software modules, components, and other means to structure software systems. Furthermore the control flow has to be described and how the software system is distributed among servers.

To bring network and software simulation together both simulation models have to be combined. In the following several ways of integration on the model layer will be introduced.

A. Separate Models

A separate models approach is utilised when no direct integration of models is used. This can occur when models are structurally not-connectable or have no direct matching entities. In addition separate models can be required by the utilised simulation framework(s).

B. Monolithic Model

A monolithic model describes the elements of two or more domains as single model. This approach has to be used when a simulation framework only allows for one input model. Nevertheless a monolithic model has the disadvantage that it can be problematic to find a representation that fits the high level domains elements. Thus all domain elements have to be reduced to the supported basic concepts. Usually this will lead to a loss of information, e.g. the context of elements.

C. Federated Model

A federated model integration uses an additional model that links the input models. This extra model is some kind of bridge between domains. It includes basic mapping entities for every element in the intersection of models. Simulation frameworks can use these mapping entities to propagate changes and keep elements in sync. The federated model is commonly used used for federated simulations c.f. Sect. IV-D.

D. Decorated Model

The decorated model approach is based on the usage of a *decoration model* to extend a simulation model with further information. These information can be based on another simulation domain than the model that is decorated. A decoration model has references to the model elements in the model it decorates. In this way the decoration model can add information and annotate elements. For two different domain models usually one domain model will be used to decorate the other domain model.

When a decoration model is only used to add details to existing model elements it is often called *completion*. A
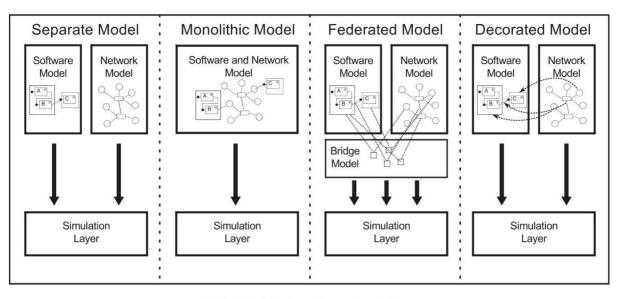
Fig. 2. Simulation integration on the model layer

completion can be seen as a kind of macro mechanism to expand model elements.

The decorated approach has the advantage that no changes to the base model must be made. Furthermore a model can be decorated by more than one decoration model. At plus this technique allows for a modular composition of the simulation model and supports the information hiding principle.

IV. COUPLING OF SIMULATION FRAMEWORKS

There exist several possibilities for coupling two or more simulation frameworks. Coupling of simulations is quite common in the computer aided engineering (CAE) discipline where many relevant aspects have to be simulated at the same time. For instance the simulation of a car crash test uses many different physics and deformation simulations.

The coupling of simulation introduces some Challenges. One major problem is the management and synchronisation of simulation time. Another task is the syncing of entity states and the propagation of events. In the following several approaches for coupling of simulation frameworks will be introduced. All approaches are shown in figure 3.

A. Independent Simulation

An *independent simulation* means that two or more simulations are executed independently from each other such that no simulation result is influenced by running another simulation. For each simulation a separate input model is required that is based on a simulation specific meta model. In special cases these simulation specific models can also be generated from a common model. After all simulations have finished the results are combined. Thus the coupling is solely realised on the result layer.

B. Simulation with Feedback

A simulation with feedback means that the result of one simulation is used as input for another. This input can be

necessary when simulations are dependent. For example

When simulations are interdependent it is necessary to use both all as inputs for each other. This will usually lead to a cycle in the dependency graph. To solve this cycle, simulation results can be used as input iteratively until a fixpoint is reached. I.e. both simulations are repeated with the result of the last iteration as input until no significant changes occur. This method is not very efficient and is not guaranteed to converge when no steady state is reached.

C. Direct Communicating Simulations

Another alternative for coupling simulations is the use of direct communication between simulation frameworks. This approach has the advantage that interdependencies can be resolved directly. A major disadavntage is, that the simulation logic has to be changed to allow this communication. Furthermore, the communicating simulations should be compatible in ways of time and event management, such that only lightweight wrappers and adapters have to be implemented. A large limitation of this approach is that it requires two simulation frameworks at the same time. Thus more CPU and memory is utilised.

D. Federated Simulations

Federated simulations approach is quite similar to the direct communication approach. It requires both simulation frameworks to run at the same time but a centralised co-ordination layer is used. This co-ordination layer is responsible for managing the time and simulation flow. A federated simulation usually also has a federated object model that is used for passing information between simulations.

The major disadvantage of this approach is that the simulation core of the frameworks have to be changed, as time management is externalised. Furthermore a fast co-ordination layer is required for efficient simulation. The federated simulation
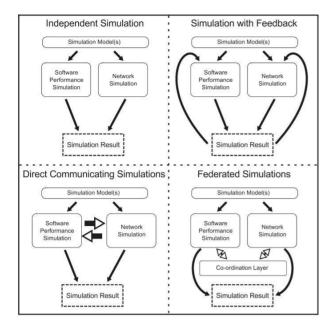
Fig. 3. Coupling of software and network simulations

approach allows for flexible coupling of different simulations. The *High Level Architecture (HLA)* as defined under IEEE Standard 1516 [12] is a standard for distributed federated simulations. It defines an interface for a run-time infrastructure being the co-ordination layer. Moreover the *object model template (OMT)* can be used to define federated models that are used in HLA simulations.

V. AN INTEGRATED SIMULATION APPROACH

In this section we will introduce our approach for an integrated simulation, that is suitable for the prediction of performance and other quality attributes of highly distributed systems. Hence our approach is based on the integration of network and software system simulation. This integration is based on model driven techniques.

On the one hand the basis of our approach is the palladio component model that is used for the modelling of software systems. This model will be extended by using completions for the detailed modelling of network layer entities. On the other hand the OMNeT++ network simulation framework will be used to simulate the network and software systems in an integrated way.

A. Integrated Model

Our approach uses a specialised software simulation model as basis. This model is derived from the palladio component model by using a model-to-model transformation. Thus the distributed system can be modelled using existing editors.

As before-mentioned in the palladio component model the behaviour of component methods is specified as resource demanding service effect specifications (RDSEFFs). For use in the integrated simulation model these RDSEFFs are transformed to a set of instruction entities that represent the

behaviour. Every step of the control flow is represented as a distinct instruction entity. The number of instruction entity types is intentionally kept low to make an implementation of these entities as simple as possible. Furthermore each of these entity types has well defined semantics.

This reduced set of control flow instructions for example includes a beginning of a variable scope, a definition of a variable, an acquire operation, an asynchronous call, waiting for events, and the consumption of an active resource. Many behaviour defining elements originally used in a RDSEFF are replaced by two or more entities. Thus the integrated simulation model usually has a higher number of elements. As an example, the model-to model transformation will substitute every synchronous call in a RDSEFF by a pair of asynchronous calls and a wait operation that waits on the return call. This reduction of control flow concepts can be compared to the use of a reduced instruction set (RISC) [13] architecture.

The basic entities are connected in the order of execution and also includes means to model branches, loops, and other conditional jumps. By passing events between these instruction entities the running of the software is simulated. The events allow to pass variables and state information of the current job between the entities.

Beside the behavioural modelling the intermediate simulation model also includes structural information. This structural information is used to map behaviour to the modelled software components which are adopted from the PCM . Furthermore the component assembly is also represented and composite components are supported. In addition software components are assigned to systems. The services provided by the components assigned to a system can be accessed locally and using network.

A system can have several active and passive resources that can be accessed by the entities using events. By using the before mentioned variables and state information parametric resource demands can be applied when an active resource is called.

Furthermore the network connections between systems are modelled. A network connections has attributes that specify the throughput, latency, and the used protocol.

For advanced modelling of networks, like choosing parameters of protocols, we decided to use the decorated model approach. This allows us to hide domain specific design decisions that are not applicable universally. Thus the used decoration model contains all information that is specific to the utilised network simulator. In this way present editors for networks can be used as well to model network specific parts.

The model integration technique used in our approach can be seen as a mixture of a monolithic model integration and the use of a decorated model.

B. Simulation

As our integrated simulation model is event based, the simulation framework has to be event driven as well. We decided to use the OMNeT++ simulation framework that
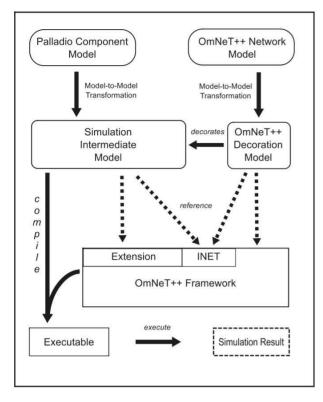
Fig. 4.   Schematic illustration of our integrated simulation approach

offers support for discrete event simulation. Nevertheless our approach can be applied to other simulation frameworks as well in an easy way.

We extend the OMNeT++ simulation framework to support all entities of the integrated simulation model. This extension is quite simple as the semantics of entities is well defined. Furthermore only few entity types have to be implemented. The components in our integrated simulation model can be mapped easily to OMNeT++ cModules and cCompoundModules. In this way the integrated simulation model can be used as direct input. In addition the decoration model can also be used directly as it only contains entities already present in the OMNeT++ meta-model.

The amount of network traffic that is simulated using the INET framework is calculated from the variable characterisations of the parameters used in the signature of a request. In addition an overhead is added that results from the used protocols. Where possible, already existing OMNeT++ protocol implementations are used.

To run a simulation the integrated model is used as input that is compiled, together with the OMNeT++ simulation core, the INET framework, and our extensions, to a run-able executable. Thus the bottom line is that we use an independent simulation that is only coupled by an integrated model layer. Though when further simulation frameworks are utilised, e.g. for simulation of additional quality attributes, other coupling methods can be used easily.

VI. Related Work

There exist several works in the field of performance and quality prediction of software systems. Many of these are based on UML and the UML profiles SPT [7] and MARTE [8].

In [14] Pustina et al. describe a method for the performance evaluation of communication systems including protocols and devices. They use an UML model of the system annotated with either the UML SPT or UML MARTE profile, that is converted into a queuing network based performance model. This performance model is then evaluated using an implementation of queueing networks based on the OMNeT++ simulation framework. Though Pustina et al. also use the OMNeT++ framework, their work is focused on the embedded systems domain and does not yet include detailed network models.

Another approach that makes use of the UML SPT profile is the KLAPER intermediate language introduced by Grassi et al. [15]. Furthermore KLAPER also supports other component based models as input. The KLAPER model is focused on the performance and reliability analysis, but has in common with our approach that a quality aware intermediate model is used. However, KLAPER uses a single monolithic model and uses multiple analysis tools like queuing networks together with a feedback loop. Network specific constructs are not included in the KLAPER language.

A similar analysis approach is the Performance by Unified Model Analysis (PUMA) [16]. It uses the so called Core Scenario Model (CSM) that integrates the SPT profile information with the common UML model. The CSM is similar to the integrated simulation model used in our approach as it also contains entities for every step in a control flow and supports active and passive resources. Nevertheless the CSM is centred on the support of different analysis techniques and only targets simulations tangentially. At plus networks are not modelled in depth.

VII. Conclusion and Future Work

In this paper we present an approach for an integrated simulation of software and network performance. We expect that the use of a fully-fledged network simulator can improve the prediction results of software performance simulation. No existing software performance simulation approache uses a detailed modelling and simulation of network effects that can strongly influence the performance of (highly) distributed systems.

The intermediate simulation model introduced in this paper is generated from a palladio component model and a network model description using model driven techniques. This allows the reuse of existing tools for the palladio model and the application of our approach to existing network models as well. Furthermore the existing tools for the used network simulation framework OMNeT++ can be reused as well.

Our proposed approach has the main advantage that the underlying simulation framework has not to be changed and only has to be extended to support the introduced software simulation elements which have well defined semantics. This allows us to exchange one used simulation framework for

another. Moreover in future work simulation frameworks can be used that predict other quality attributes.

In future work it has to be evaluated whether our approach is really suited to improve the prediction of highly distributed systems. Therefore after the initial implementation of the OMNeT++ extension is finished, experiments will be run that compare our new simulation to existing simulations. At plus our simulation will be compared to the performance of real distributed software systems.

REFERENCES

[1] T. Sridhar, "Cloud computing," *The Internet Protocol Journal (IPJ)*, vol. 12, no. 3, pp. 2–19, September 2009.

[2] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation (3rd Edition)*, 3rd ed. Prentice Hall, 2000.

[3] The network simulator - ns-2. [Online]. Available: http://www.isi.edu/nsnam/ns/

[4] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment." in *SimuTools*, S. Molnár, J. Heath, O. Dalle, and G. A. Wainer, Eds. ICST, 2008, p. 60.

[5] The inet framework for omnet++. [Online]. Available: http://inet.omnetpp.org/

[6] Gcc, the gnu compiler collection. [Online]. Available: http://gcc.gnu.org

[7] Object Management Group (OMG), "UML profile for schedulability, performance, and time specification version 1.1," 2005.

[8] ——, "A UML profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, beta 2," 2008.

[9] R. H. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg, "The Palladio Component Model," Universitaet Karlsruhe (TH), Tech. Rep., 2006.

[10] S. Becker, *Coupled Model Transformations for QoS Enabled Component-Based Software Design*, ser. The Karlsruhe Series on Software Design and Quality. Universitätsverlag Karlsruhe, 2008, vol. 1.

[11] "The DESMO-J Homepage," University of Hamburg, Department of Computer Science, 2007, last retrieved 2008-01-06. [Online]. Available: http://asi-www.informatik.uni-hamburg.de/desmoj/

[12] Simulation Interoperability Standards Committee (SISC), "IEEE standard for modeling and simulation (M & S) High Level Architecture (HLA) - Framework and Rules," pp. i –22, 2000.

[13] A. H. J. Sale, "The risc style of architecture." *Australian Computer Journal*, vol. 21, no. 3, pp. 97–99, 1989.

[14] L. Pustina, S. Schwarzer, M. Gerharz, P. Martini, and V. Deichmann, "A practical approach for performance-driven uml modelling of handheld devices - a case study," *J. Syst. Softw.*, vol. 82, no. 1, pp. 75–88, 2009.

[15] V. Grassi, R. Mirandola, and A. Sabetta, "From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 25–36.

[16] M. Woodside, D. C. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (PUMA)," in *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*. New York, NY, USA: ACM Press, 2005, pp. 1–12.

Towards Secure Services in an Untrusted Environment

Matthias Huber

Institute for Program Structures and Data Organization
Faculty of Informatics
Karlsruhe Insitute of Technology, Germany
Email: matthias.huber@kit.edu

Abstract—**Software services offer many opportunities like reduced cost for IT infrastructure. However, they also introduce new risks, for example losing control over data. While data can be secured against external threats using standard techniques, the service providers themselves have to be trusted to ensure privacy. Cryptographic methods combined with architectures adjusted to the client's protection requirements offer promising methods to build services with a provable amount of security against internal adversaries without the need to fully trust the service provider. We propose a reference architecture which separates services, restricts privilege of the parts and deploys them on different servers. Assumptions about the servers' and adversary's capabilities yield security guarantees which are weaker than classical cryptographic guarantees, yet can be sufficient.**
Keywords: services, cloud computing, security

I. INTRODUCTION

Due to advances in networking and virtualization technology, new paradigms of providing IT infrastructure and software have emerged – among them the so-called *Cloud Computing*. The National Institute of Standards and Technology [21] defines Cloud Computing as "a model for enabling convenient, on-demand network access to a shared pool of computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". Cloud Computing enables clients to use Software as a Service and to outsource their data, thus cutting the cost of maintaining an own computing center.

However, inherent to Cloud Computing are privacy problems [17], [22]: By using services in the Cloud, clients lose control over their data. Clients can not control if their data gets copied or misused on the server. The threat of insider attacks exists and keeps many potential customers from using Cloud Computing in critical or sensitive scenarios (e.g., scenarios comprising business secrets or customer data). Current security mechanisms focus on protecting the data transfer to and from the service provider.

Protecting a pure storage service against insider attacks is easy. Encrypting all data on the client before uploading it to the server provides a sufficient level of protection based on the used encryption [4]. However, this prevents the server from performing any meaningful operation on the data. Hence, more complex services require advanced techniques for providing privacy.

Cryptographic methods like *secure multiparty computation* [7], [13] or *private information retrieval* [11] can in principle solve all privacy problems. Especially since a *fully homomorphic encryption* method [12] was discovered in 2009 which allows calculations on encrypted data. These methods offer strong security guarantees. For example fully homomorphic encryption allows to build services where the service provider does not learn anything about the user input. However, due to high communication and computation costs, these methods are not practical and their costs outweigh all benefits of outsourcing software or data.

Nevertheless, security guarantees for Cloud Computing are needed: Without provable security, outsourcing is not an option for most sensitive scenarios. Since achieving classical cryptographic security guarantees is in most cases infeasible for service scenarios, these guarantees have to be weaker, yet provide a sufficient level of protection. Moreover, to be accepted broadly the security guarantees must be easy to understand.

The contribution of this paper is an idea for solving the privacy problems inherent to Cloud Computing. We suggest partitioning a service on the basis of its duties and deploying its parts on different servers. Combined with cryptographic methods, this approach can provide a provable level of protection. We discuss an architectural style for separating a service and provide examples how this separation can enhance the security of a database service.

We also sketch a new security notion inspired by k-anonymity [8], [24] which can be applied to outsourced databases. This notion has been defined in our previous work [16]. The level of privacy provided by this notion is easy to understand and we show informally that our database example fulfills this notion. In contrast to secret sharing, we respect algorithms and data structures and thus preserve the efficiency of the service. This is vital for services: Outsourcing is pointless if the benefits are canceled by security measures that try to solve problems introduced by outsourcing. Unlike our previous work [16] which focuses on a formal security property, this papers focus is on the architecture implied by a separation of duties.

The rest of this paper is organized as follows: We give a short introduction to approaches for the security of outsourced databases in Section II. We also give an overview

of related cryptographic primitives. In Section III, we discuss the requirements for solutions to the problem motivated in this Section. We present a reference architecture that we call *Separation of Duties* which allows building more secure services in Section IV. In Section V we discuss why we need new security properties to provide provable security. In Section VI, we show with the help of an example of an outsourced database how Separation of Duties can enhance the security of services. We discuss assumptions needed for our approach and its limitations in Section VII. In Section VIII, we discuss future work and conclude in Section IX.

II. RELATED WORK

Since many services rely on databases, in Section II-A, we give an overview of the Database as a Service literature that tries to solve the privacy problems of outsourced databases with the help of cryptographic methods. Apart from the methods used here, cryptography offers additional methods to engage privacy and security issues. We discuss other existing cryptographic methods in Section II-B.

A. Database as a Service

The database community realized the benefits of outsourcing early. The concept of *Database as a Service* was introduced by Hacigümüs et al. in 2002 [15]. They propose to use encryption to enhance privacy and evaluate several different ways to encrypt data in a database. However, the user has to hand the encryption key to the server for query processing. This is a security risk in an untrusted server scenario. Since then the privacy aspects of this concept and the problem of a *searchable encryption* got much attention. In [14], Hacigümüs et al. propose a tuple level encryption scheme and coarse indices to enable the server to execute SQL on the encrypted data. In this scenario, the server does not need to decrypt the data for coarse-grained query execution, and returns a superset of the tuples queried. The client has to decrypt the returned data and execute the exact query on it. There were other papers considering different aspects of this idea: Damiani et al. consider confidentiality in encrypted databases in [9]. They introduce exposure coefficients and examine databases where an index refers to just one attribute value. A more detailed view on exposure coefficients which considers coarse indices is given by Ceselli et al. in [6]. However it remains unclear, what level of privacy can be provided by considering exposure coefficients. In [18], Hore et al. addressed the problem of creating an optimal bucketization scheme under efficiency aspects.

In contrast to the schemes described above, Aggarwal et al. propose to separate a database according to privacy constraints and to distribute it to different providers [1]. They propose to use encryption if a privacy constraint cannot be met and present three encryption schemes, namely one-time pad, deterministic encryption, and random addition. They also propose adding noise to enhance privacy of a distributed database. Moreover, they define a composition as privacy preserving if all privacy constraints are met. However, this definition

depends on the privacy constraints of the actual database. It remains unclear what level of privacy can be provided if the associations between deterministically encrypted attribute values are not hidden.

Kantarcioglu and Clifton showed in [19] that classical cryptographic notions are not applicable to encrypted databases under practical constraints because in general it is infeasible to realize a database that complies with these notions. Therefore, they define new notions and propose an encrypted database with hardware support. In [2] Amanatidis et al. consider searchable encryption from a provable-security methodology point of view. They propose an encryption scheme capable of efficient exact match and range queries while providing provable security. In [3], Bellare et al. propose a new security notion called *PRIV*. In contrast to our notion, a PRIV-secure database provides provable privacy only if the plaintext is drawn from a space of high min-entropy. They emphasize that the proposed schemes enable sublinear search complexity which is important for databases to enable fast query processing.

B. Cryptography

In this Section we discuss cryptographic methods, namely *Multiparty Computation*, *Homomorphic Encryption*, and *Private Information Retrieval* which can in principle solve all the privacy and security problems outlined in Section I. These methods may be applicable in special cases. In general, however, the complexity of these methods cancels the benefits of outsourcing.

1) Multiparty Computation:
There are cryptographic solutions for two or more parties cooperatively computing a certain function over a set of data without any party learning anything about the input of other parties except what is learned by the output. Using an interactive protocol, these *secure multiparty computations* [7], [13] can thus solve all computation related privacy problems. The problem is that for each party, the computation cost is higher than computing the whole function on the complete input without any other party. This makes the concept of multiparty computation for outsourcing services too expensive and in fact pointless if the client is the only one with private input.

2) Homomorphic Encryption:
There are encryption schemes that produce ciphertexts with homomorphic properties: Consider for example Textbook-RSA [23]. Multiplying two ciphertexts and decrypting the result yields the same result as decrypting the two ciphertexts and multiplying the plaintexts. However, Textbook-RSA is not considered as secure [5]. In 2009, Craig Gentry discovered a fully homomorphic encryption that supports multiplication as well as addition [12], and theoretically solves our privacy problem: The client could simply use the proposed encryption scheme, and the service provider could adapt its service to work on encrypted data using this scheme. However, this is not feasible since the size of the key scales with the size of the circuit of the algorithm which the service calculates.

3) Private Information Retrieval:

Cryptography offers a method to retrieve information from an outsourced database without the server learning anything about the information queried [11]. This is a special case of oblivious transfer [25]. However, these methods are also infeasible in most cases: If the database server must not learn anything about a query, the query issued to the database must contain every cell. Otherwise the server learns which cells do not contribute to the result of the query, and thus learns something about the result set, if no special-purpose hardware is involved [19]. The need to iterate over every cell for every query execution makes private information retrieval impractical in most cases.

III. REQUIREMENTS

In this chapter we discuss the requirements of a solution for the privacy problems inherent to most services. Basically, solutions for the presented problem should provide *provable security* and *practicability*. While there are solutions that provide either provable security or practicability (cf. Section II), there is a lack of solutions that provide both.

A. Requirement 1: Provable Security

Formal notions and proofs are provided for cryptographic methods. While providing security in the sense of classical cryptography is infeasible for nontrivial services (e.g. services more complex than pure storage services), solutions for the privacy problem described in Section I should provide provable security. This means that we need a formalization of the level of privacy provided. Furthermore, we need to prove that our solution provides this level of privacy (fulfills the formal notion). A level of privacy one wants a service to provide is for example that an adversary cannot learn relations between attribute values of an outsourced database (cf. Section V).

B. Requirement 2: Practicability

There are cryptographic solutions for all problems mentioned in Section II. However, in most cases they are inapplicable due to their complexity. We strive for solutions that are practical and do not cancel the benefits of outsourcing. Also the provided level of privacy has to be easy to understand, since this is vital to the acceptance of the solution.
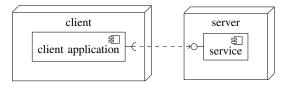

Fig. 1. System and deployment view of a client application deployed on a client machine invoking a service deployed on a server.

IV. SEPARATION OF DUTIES

In order to reduce the information a single provider can learn, we propose to combine the design pattern *Partitioned Application* [20] with the *Need to Know Principle* [27] and apply it to services. Partitioned Application is a security design pattern that suggests to split a large monolithic application into several parts and to restrict the privileges of each part individually. The Need to Know Principle originates from the intelligence service area and suggests to allow access to the data needed, and to deny access to all other data. For services, this means separating a service with respect to its algorithms and data structures and deploying each part on a different server. Each part only gets the data needed to fulfill its duty.

Consider for example a route planning service. This service can be separated into a maps part and a route optimization part. A client using this service can get the maps required for his route, extract the graph data, and send it to the route optimization part, which does not need the names of the locations, but the graph representation of the map to find the cheapest way. With this separation, the service provider does not necessarily learn the location of departure and the destination of the client.

Assumptions about the server's capabilities (like limited or no storage) and the application of well-understood efficient cryptographic methods result in provable and easy to understand security guarantees assuming the adversary only has access to one server. If the separation respects the algorithms and data structures of the service the efficiency of the overall service can be conserved with little overhead. Therefore Separation of Duties can be a solution for the privacy problem of services that fulfills both requirements from Section III. As a side effect, since the code of the service is partitioned and distributed as well, this can also impede software piracy.

Separating a service (c.f Figure 1) into different parts can be done in two ways which we term *serial* and *parallel*. We will explain with the help of examples of outsourced databases how a service can be separated. While these separations for themselves in general does not offer an additional level of privacy, they allow us to deploy them on different servers and to apply cryptographic methods like encryption and provide a level of privacy that can be sufficient while maintaining scalability. Note that the parts even can be deployed on servers of different infrastructure providers.

In the remainder of this Section, we will present serial and parallel separation, and discuss the security implications of these separations. In Section VI we provide an example for a separated database which offers privacy due to the separation and the applied cryptographic methods.

A. Serial Separation

If a service is separated into two (or more) parts and a newly introduced requiring interface of the first part is connected to a newly introduced providing interface of the second part then we call this *serial separation* (cf. Figure 2). This separation is transparent to the client if the providing interface on the first part does not change.
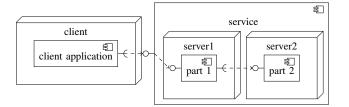
Fig. 2. System and deployment view of a serial separation of a service

Fig. 3. An example for a serial separation: The architecture of a secure outsourced database proposed in [14]. The adaptor provides and requires a standard SQL interface. The overall database service consists of the database service on the server and the adaptor on the client.

Consider for example the architecture of the outsourced database proposed in [14] (cf. Figure 3). The database service is split into two parts: The database part is deployed on a server and the adaptor part is deployed on the client. A sequence diagram for a query is depicted in Figure 4. The adaptor translates queries from the client application to match the scheme of the encrypted database of the database service. The adaptor then decrypts the results, eliminates false positives, and returns the results to the client application.

With serial separation, in order to be connected the components or the servers the components are deployed on have to be aware of each other. If this is a potential security risk, since it elevates the likelihood of malicious cooperation. This can be avoided by using anonymizing proxies. However, since the first part of a service has a requiring interface, it knows that there is a second part somewhere.

B. Parallel Separation

If a service is separated into two (or more) parts which are connected in parallel to an adaptor component or the client application we call this *parallel separation*. In order to be connected, the parts of a serially separated service or the servers the parts are deployed on have to be aware of each other. With parallel separation, this is not necessary. However, for such a separation to be transparent to the client application, an adaptor has to be deployed. Figure 5 shows a parallel separation with an adaptor component deployed on a third server. This adaptor can be as well deployed on the client's machine.

Consider for example the architecture depicted in Figure 6. Here, the database indices are outsourced to another component deployed on a separate server. In order for this separation to be transparent to the client, an adaptor component is deployed on the client's machine. The overall service consists of the adaptor, the database indexing service and the database service. The adaptor or the client application has to call both, the indexing service and the database service in order to
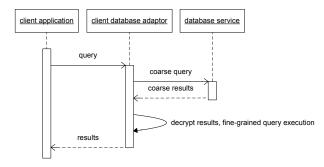

Fig. 4. Sequence Diagram for a query in the serial separated database. The adaptor translates queries from the client application to match the scheme of the encrypted database of the database service decrypts the results and returns them after removing false positives to the client application.
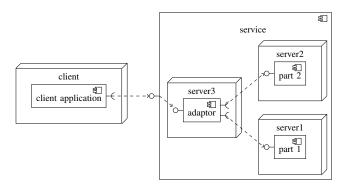

Fig. 5. System and deployment view of a parallel separation of a service, and an adaptor. The composition structure and the deployment relationship may cross each other. Consider the adaptor deployed on the client. This is not in conflict with UML and conceptionally sound, as long as the inner components are bundled by an *assembly* where inner components remain visible, exactly for the reason of possibly distributed deployment.

execute a query. In Figure 7 a sequence diagram for a query is depicted. A client calls the adaptor. The adaptor calls the indexing service and then queries the database using the results from the indexing service. Then, it returns the result from the database service to the client application.

Parallel separation can reduces the likelihood of malicious collaboration. The components do not have to be aware of each other, and the adaptor can be deployed on the clients' machine. However, parallel separation of services may be harder to
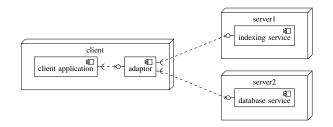

Fig. 6. An example for a parallel separation: The overall database service is separated in an indexing service and a database service deployed on different servers. An adaptor deployed on the client's machine provides a standard SQL interface to the client application.

achieve since it requires a new component and can require an adaption of existing components and the calling party as well. This may be counterintuitive for software engineers.

C. Security of Separations of Duties

Separating a service does not enhance the security in general. Consider for example the route planning service from the beginning of this Section. If the route optimization part can match the input graph unambiguously to an area, privacy is lost.

However separating a service allows to restrict the privilege of individual parts, or apply cryptographic methods that also can be seen as a restriction of privilege. If the route optimization service does not get graph data which is unambiguously mappable (e.g. noisy data), intuitively there is a privacy gain.

V. Security Properties

For cryptographic methods, there are security notions that formally describe the security provided by these methods. Classical cryptographic security notions are inspired from indistinguishability. For example, an encryption scheme is considered secure if two different ciphertexts are indistinguishable for an adversary. If an adversary is given a plaintext, the encrypted plaintext and another ciphretext, she should not be able to relate one of the two given ciphertexts to the plaintext correctly with a probability significantly greater than $\frac{1}{2}$. Otherwise the encryption scheme is considered insecure. This is a very strong notion. It implies, that a ciphertext yields no information about the plaintext.

However, classical style cryptographic notions are not applicable in a Service scenario under practical constraints: If you transfer indistinguishability-inspired notions to a Service scenario, you would want the server not to be able to distinguish between two different sets of data sent to the service. Neither would you want the server to gain information about the input from an execution of the service (to be able to distinguish two executions of the service). For Database as a Service, it is neither possible to obtain *database indistinguishability* nor *query indistinguishability* under practical performance constraints [19].
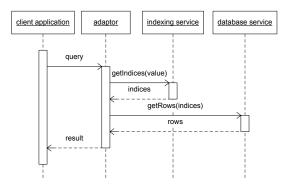

Fig. 7. Sequence diagram for a query on an example of a parallel separated database service: The client queries the adaptor, which calls the indexing service for the indices needed for the query of the database.

To provide provable security, we need a formulation of the level of security we want to provide. In order to be applicable in a service scenario, we strive for cryptographic security notions that are weaker than classical notions, but provide a sufficient level of protection:

Consider a system that hides relations between attribute values while allowing attribute values itself to leak. Such a system would not considered as secure by classical cryptography, but the level of privacy it provides can be sufficient. In some cases it may be even sufficient to allow to leak that an arbitrary value of an attribute is related to k distinct rows in the database This is inspired from the anonymity property *k-anonymity* [8], [24]. As the level of privacy provided by hiding relations is easy to understand, it is not trivial to formalize and is out of the scope of this paper. However, formal security notions are in the scope of our work (cf. [16]).

Note that there are different levels of privacy between no security and perfect security. These levels in general cannot be compared to each other, since they form a half order. For example, you cannot compare the levels of privacy provided by hiding relations to the level of privacy provided by hiding attribute values.

Hiding relations, but allowing attribute values to leak is clearly weaker than database indistinguishability [19], yet achievable with reasonable overhead. In the next Section we will present an example for a separated database that hides relations.

VI. Example

In Section IV we introduced parallel and serial separation of a service and provided examples. While these separations do not provide an additional level of privacy in general, they allow us to apply cryptographic methods while maintaining scalability. In this Section we provide an example for a separated database that offers privacy due to the separation and the applied cryptographic methods.

Consider the toy CRM database depicted in Figure 8 containing the names, surnames and bank account numbers of individuals. We want to outsource this database without an

row	name	surname	bank account
1	Alice	Smith	573535635
2	Bob	Smith	436343346
3	Alice	Jones	343462624
⋮	⋮	⋮	⋮

Fig. 8. An toy example of a CRM database.

adversary learning the relations between the attribute values (cf. Section V). For example, we allow an internal adversary to learn that our database contains a tuple with the name value "Alice", but we do not want the adversary to learn that there is a tuple in the database with the name value "Alice" *and* the surname value "Smith". Additionally, one may want to hide the number of occurrence of attribute values. For example, we may want to hide that the attribute value "Alice" occurs two

row	ENC_{prob}(name, surname, bank account)
1	ENC_{prob}(Alice, Smith, 573535635)
2	ENC_{prob}(Bob, Smith, 436343346)
3	ENC_{prob}(Alice, Jones, 343462624)
$\vdots$	$\vdots$

Fig. 9. The toy CRM database from Figure 8 with tuples encrypted probabilistically.

times in our database. In this database, we want to be able to search by name and surname values, but we never search by the bank account number. Furthermore, we only want to be able to execute exact match queries efficiently.

Creating explicit index tables for the attributes "name" and "surname" allows (cf. Figure 10) to apply encryption to the original database. These indices can either be created bit by bit if the database is empty in the beginning, or in one step by the client before the database gets outsourced. In this example we propose to apply probabilistic encryption on tuple level. Figure 9 depicts the resulting table. Here, the tuples are encrypted probabilistically with ENC_{prob}. The attribute values can only be read with the appropriate encryption key. Of course, this prevents the execution of most queries. However, exact match queries still can be executed by using the previously created index tables (cf. Figure 10). Note that efficient execution of other queries e.g. range queries or sum and avg aggregates can be provided with more sophisticated index structures [1], [14]. For the simplicity of this example, we focus on exact match queries.

If the index tables and the encrypted CRM table are deployed on the same server, attribute value relations may still leak even if the values in the rows column are encrypted (cf. Figure 12). An adversary who has access to logs can infer that rows often queried successively are related. Therefore we deploy each index and the relations database on different servers. Figure 11 depicts the system and the deployment view of the resulting database service. The overall service consists of the adaptor deployed on the clients machine, the indices and the relations database.

To hide the number of occurrence of individual attribute values, the entries in the rows column can be padded to a fixed length and encrypted probabilistically (cf. Figure 12).

Note that search time for exact match queries for this database system is still sublinear.

name	rows
Alice	1, 3
Bob	2
$\vdots$	$\vdots$

(a)

surname	rows
Smith	1, 2
Jones	3
$\vdots$	$\vdots$

(b)

Fig. 10. Separated indices for the attributes name and surname of the database in Figure 8
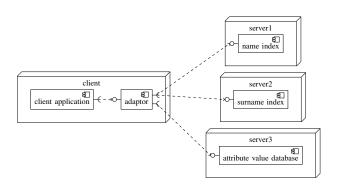

Fig. 11. Structural and deployment view of our separated CRM database. The indices are deployed on separate servers as well as the encrypted attribute values database. An adaptor deployed on the clients' machine provides a standard SQL interface to the client application. The overall service consists of the indices, the adaptor and the attribute values database.

name	rows
Alice	ENC_{prob}(1, 3)
Bob	ENC_{prob}(2)
$\vdots$	$\vdots$

(a)

surname	rows
Smith	ENC_{prob}(1, 2)
Jones	ENC_{prob}(3)
$\vdots$	$\vdots$

(b)

Fig. 12. Encrypted separated indices for the attributes name and surname of the database in Figure 8. The first column contains the attribute value, the second column contains a probabilistically encrypted list of the rows where the attribute values occur in the original database.

Consider for example the query:

```
query := SELECT * FROM table
        WHERE name = "Alice"
        AND surname = "Smith"
```

A sequence diagram for this query is depicted in Figure 13. After receiving this query the adaptor has to get the appropriate tuple IDs from the indices. These queries can be executed in parallel:

```
subquery1 := SELECT rows FROM name index
            WHERE name = "Alice"
subquery2 := SELECT rows FROM surname index
            WHERE surname = "Smith"
```

After decrypting the results, the adaptor gets the tuple IDs 1 and 3 from the name index and the tuple IDs 1 and 2 from the surname index. The client can intersect the two id sets and query the attribute value database for the appropriate tuples:

```
get tuples := SELECT * FROM
            attribute value database
            WHERE row = 1
```

After decrypting the result, the adaptor returns it to the client application. Note since the adaptor is deployed on the client's machine, the server does not need the encryption keys, and relations between attribute values are hidden assuming an adversary only has access to one server.

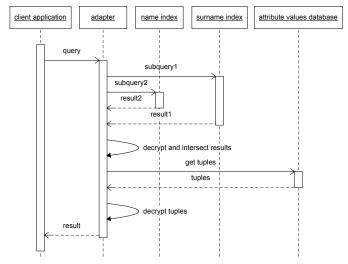This example shows that it is possible to separate a database

Fig. 13. Sequence diagram for a query in the separated database. After receiving a query from the client application, the adaptor queries the indices for tuple IDs, then the attribute value database and returns the decrypted results to the client application.

and to apply cryptographic methods in order to hide relations between attribute values. For additional levels of privacy further methods may be used. For example dummy queries can shroud the frequency distribution of attribute values involved in queries. Deterministic encryption of the keywords of the indices still allows for efficient execution of exact match queries, but can hide even single attribute values.

VII. ASSUMPTIONS AND LIMITATIONS

For our approach we need some assumptions.

We assume that adversaries cannot solve problems in feasible time for which no efficient algorithm has been found yet. This is a reasonable assumption and broadly accepted.

We currently assume that the adversary is honest but curious. This means that the adversary does not change data or influence calculations on the server, but just observes, and tries to learn as much as possible. Consider for example a system administrator who dumps data and tries to sell it. We assume that the client can detect malicious behavior beyond eavesdropping. This is reasonable since in most cases such a behavior can be detected, for example by occasionally comparing the output of the service to the expected output, or by using hash codes for storage.

We also assume that adversaries only have access to one server. This is also reasonable since the parts of the service can be distributed to different servers or even different infrastructure providers. This makes malicious cooperation highly unlikely.

VIII. FUTURE WORK

For future work we plan to investigate more complex index structures that allow efficient execution of other types of queries. We plan to evaluate different alternatives for databases

with respect to the level of privacy provided and the performance of the overall system.

We also plan to investigate if and how Separations of Duties can be applied to more complex services and how it can enhance the privacy in other scenarios than outsourced databases.

We also need formalizations of the level of privacy we want to provide. We need security notions, that are easy to understand, and achievable in a service scenario, yet provide sufficient privacy guarantees.

We want to understand new methods such as adding noise or dummy data that intuitively enhance privacy (cf. the example from Section IV-C). There has been some work [10], [26] considering the effects of noise or artificial data. However, the privacy aspects in a service scenario are still unclear. Investigating the privacy properties of adding noise or dummy data seems a promising direction.

IX. CONCLUSION

Privacy problems are inherent to services. Since classical cryptographic security notions are not applicable to services, we need new security notions which are easy to understand and are weaker than classical notions but provide a sufficient level of protection, and means to realize them. We presented a reference architecture that we called Separations of Duties which can potentially solve the privacy problems by i) separating a service into several parts and restricting privilege of individual parts and therefore ii) allowing the application of practical cryptographic methods to single parts. We presented an example of a separated database service and showed that this service hides relations of attribute values while providing efficient execution of exact match queries.

X. ACKNOWLEDGEMENT

REFERENCES

[1] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR 2005*.
[2] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *DBSec*, pages 14–30, 2007.
[3] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.
[4] Matt Blaze. A cryptographic file system for unix. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, New York, NY, USA, 1993. ACM.
[5] Dan Boneh, Antoine Joux, and Phong Nguyen. Why textbook elgamal and rsa encryption are insecure (extended abstract), 2000.
[6] Alberto Ceselli, Ernesto Damiani, Sabrina De Capitani Di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Modeling and assessing inference exposure in encrypted databases, 2005.
[7] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, New York, NY, USA, 1988. ACM.
[8] Valentina Ciriani, Sabrina De Capitani di Vimercati, Sara Foresti, and Pierangela Samarati. k-anonymity. pages 323–353, 2007.

[9] Ernesto Damiani, S. De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs, 2003.

[10] Josep Domingo-Ferrer, Francesc Sebe, and Jordi Castella-Roca. On the security of noise addition for privacy in statistical databases. In *Privacy in Statistical Databases 2004*, pages 149–161. Springer, 2004.

[11] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[12] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[13] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM.

[14] Hakan Hacigümüs, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM, 2002.

[15] Hakan Hacigümüs, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 29, Washington, DC, USA, 2002. IEEE Computer Society.

[16] Christian Henrich, Matthias Huber, Carmen Kempka, Jeorn Mueller-Quade, and Ralf Reussner. Technical report: Secure cloud computing through a separation of duties. https://sdqweb.ipd.kit.edu/huber/reports/sod/technical_report_sod.pdf, 2010.

[17] Christian Henrich, Matthias Huber, Carmen Kempka, Jörn Müller-Quade, and Mario Strefler. Towards secure cloud computing. In *Proceedings of the 11th International Symposium on Stabilisation, Safety, and Security of Distributed Systems (SSS 2009)*, 2009.

[18] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 720–731. VLDB Endowment, 2004.

[19] Murat Kantarcioglu and Chris Clifton. Security issues in querying encrypted data. Technical report, 2004.

[20] Darrell M. Kienzle, Matthew C. Elder, Ph. D, Ph. D, David Tyree, and James Edwards-hewitt. Security patterns repository, version 1.0, 2006.

[21] NIST. NIST - cloud computing. http://csrc.nist.gov/groups/SNS/cloud-computing/, 2009.

[22] Sinai Pearson. Taking account of privacy when designing cloud computing services. *HP Laboratories*, 2009.

[23] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[24] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[25] Wen-Guey Tzeng. Efficient 1-out-of-n oblivious transfer schemes with universally usable parameters. *IEEE Trans. Comput.*, 53(2):232–240, 2004.

[26] Jonathan Ullman and Salil Vadhan. PCPs ans the hardness of generating synthetic data. Technical report, 2010.

[27] H. Wedekind. Ubiquity and need-to-know: two principles of data distribution. *SIGOPS Oper. Syst. Rev.*, 22(4):39–45, 1988.

Architectural Design with Visualization Patterns

Markus Knauß

Institute of Software Technology, Software Engineering Research Group
University of Stuttgart, Germany
knauss@informatik.uni-stuttgart.de

Abstract—The design of the architecture of a software is a crucial step in the development process. The architecture defines the components of the software and the functions and responsibilities of each component. Also, the architecture determines the maintainability of the software, and the techniques that are used in the implementation. Software architecture visualizations support the architect when she designs the software because they facilitate communication, analysis, and development of the architecture. Unfortunately, no visualization is available that allows for efficient visualization of a software architecture while the architecture is designed.

This paper introduces NOTAVIS. A visualization concept that claims to efficiently support the design of a software architecture. NOTAVIS will provide a visualization that can be used as a medium for efficient communication about an architecture, and that is helpful in architecture analysis and development.

Keywords-software architecture; architecture visualization; architecture design

I. INTRODUCTION

Designing the architecture of a software is a crucial step in the software development process. The architecture defines the components from which the software is build, it decides on the technologies that must be used for implementation, and it determines the maintainability of the software. The design of a software architecture is done by the software architect.

In order to efficiently design an architecture visualizations are needed [23]. A visualization will support the analysis, development, and communication of an architecture. When designing a software architecture the architect develops a mental picture of the architecture [1], but the architecture itself is invisible. A visualization of this mental picture supports the analysis of the architecture, because the reviewers share a common picture of the architecture that is induced by the visualization [8]. Also, the visualization supports the development of the architecture, because additions and modifications to the architecture become visible. In addition, a visualization serves as a medium for communicating an architecture to their stakeholders, e.g. developers and project managers.

Therefore, visualizations are needed that efficiently support the architect when she designs an architecture, but unfortunately currently available and used visualizations do not support the architect when she designs an architecture. That is because existing visualizations are not integrated in the design process, the widely used UML diagrams do not efficiently support the design of an architecture, and the frequently used ad-hoc architecture visualizations are of little use for the further software development.

Existing architecture visualizations are not integrated in the design process. Architecture visualizations are created in one step after the architecture is designed, but the architecture design process is iterative, and an architecture is developed in several ever-changing directions, e.g. top-down, bottom-up, inside-out, or outside-in [20]. While an architecture is designed changes are common, but changes to visualizations often destroy them, e.g. the layout of the symbols. Visualizations typically do not define methods and tools that support the iterative, change-intensive design process. Therefore, architecture visualizations are not integrated in the design process.

UML diagrams are widely used for visualizing an architecture, but they do not efficiently support the design of an architecture. First, UML doesn't define a method that will allow the architect to efficiently turn his mental picture of the architecture into a UML diagram representing her mental picture. Second, most of the UML diagrams require that implementation details are known, e.g. the methods of classes. When designing an architecture this details are initially of no importance. Third, UML doesn't provide a single overview of an architecture because the elements of an architecture are scattered over different diagrams whose relationships aren't obvious. For instance, architectural structures like components, their interfaces, and their deployment are scattered over the the component, the class, and the deployment diagram. Fourth and finally, UML doesn't support the effective visualization of modern architectural concepts like plug-in frameworks for instance. This is why UML diagrams do not efficiently support the design of an architecture.

When designing an architecture architects frequently use ad-hoc architecture visualizations to overcome the shortcomings of UML [6]. Unfortunately, the ad-hoc visualizations are of little use for the further software development. The informal symbols that are used in ad-hoc visualizations have no obvious connection to the source code, therefore they are of little use for the later implementation. Ad-hoc visualizations don't serve well as communication medium, because the meaning of the symbols is not defined, and therefore the architect must describe their meaning to any stakeholder. At last, ad-hoc visualizations are of little use for the project management, because the missing connection to the source code makes it hard to use the visualization as a basis for planning and controlling the implementation phase. Thus, ad-hoc visualizations are of little use.

Although visualizations are needed for efficient design of a software architecture, actually there is no visualization in use
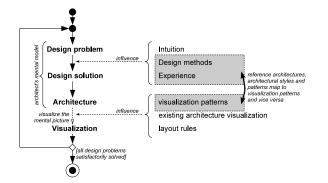
Fig. 1: The software architecture design process of NOTAVIS

Fig. 2: Refinement of a usage relationship

or available that efficiently supports the architect while she designs a software architecture. That is because visualizations are not integrated in the architecture design process, the widely used UML diagrams are unsuitable to support the design of an architecture, and ad-hoc visualizations are of little use for the further software development.

This paper presents the NOTAVIS visualization concept. It provides a method for efficient visualization of the mental picture of an architecture that the architect builds when she designs the architecture. This is done by combining design decisions, architectural styles and patterns, and architecture visualizations. In order to support the method, NOTAVIS defines a visualization tool and a graphic notation. The visualization tool complements the architecture design process and allows for an iterative visualization of an architecture while it is designed. The graphical notation is derived from typical elements of ad-hoc architecture visualizations, and is therefore easy to comprehend. Though it is derived from ad-hoc visualizations, it links to the source code, and therefore will be useful in later software development.

The structure of this paper is as follows. The next section introduces the NOTAVIS concept. NOTAVIS comprises a visualization method, a graphical notation, and a visualization tool. Following this, section III shows an example of how NOTAVIS is used in practice. Section IV shows the relations of NOTAVIS to the areas of architecture visualization, architecture documentation, and visualization. This section also compares and contrasts NOTAVIS with existing architecture visualization approaches. Finally, in section V a conclusion is drawn and an outlook to future work is given.

II. THE NOTAVIS CONCEPT

The NOTAVIS concept comprises three parts. These parts are a visualization method, a graphical notation, and a visualization tool. The visualization method is based on the architecture design process. The graphical notation is based on ad-hoc visualizations, design rules for efficient visualizations, and the connection to the implementation. The visualization tool is developed with respect to design rules for efficient visualization tools. All these parts are described in detail in the sections that follow.
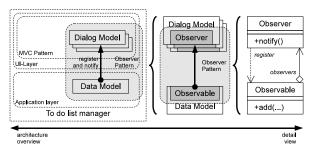
A. Visualization Method

The visualization method of NOTAVIS supports the architect in turning her mental picture of the architecture into a visualization and in refining the visualization in order to be useful for later development phases. The method is based on the architecture design process, the visualization of the mental picture, and the refinement of usage relationships between components.

The architecture design process is iterative [24]. In the beginning the architect takes the specified requirements that must be realized by the software that is to be developed. Her job is to design an architecture that allows for realizing the specified requirements [3]. In order to design a suitable architecture the architect first formulates a design problem, e.g. what architecture will allow for the implementation of a graphical user interface. For the design problem she decides on a design solution. For instance, the graphical user interface is implemented using the model-view-controller pattern (MVC). The design solution in turn leads to an architecture, continuing with the example this is a MVC architecture [15]. At last, in the NOTAVIS architecture design process the architect visualizes her mental picture of the architecture, namely the MVC architecture. This process is repeated until all design problems are satisfactorily solved, and the architecture allows for realizing the specified requirements [15]. Figure 1 shows the described design process.

A crucial step in the design process is the visualization of the architect's mental picture of the architecture [23]. NOTAVIS supports the architect in visualizing her mental picture by providing visualization patterns. Design decisions an architect takes are based on her intuition, design methods she knows, and her experience [15], [19]. Reference architectures, architectural styles and patterns, and design patterns, are called solution patterns. Solution patterns are part of design methods the architect knows and are part of her experience. Therefore, when solving a design problem an experienced architect often uses solution patterns. These solution patterns modify the designed architecture, and in turn modify the architect's mental picture of the architecture [1], [15]. To visualize her modified mental picture the architect selects the matching visualization pattern from the visualization pattern catalog of NOTAVIS and adds it to the visualization. A visualization pattern is simply a visual representation of a solution pattern.

56

TABLE I: NOTAVIS' symbols

Symbol		Semantics	Symbol		Semantics
	Component	Part of a software that realizes a function of the software and is usable through an interface.	→	Usage relation	Shows a relationship between two components. The arrow points to the used component.
	Multiple component	A component that exists multiple times in different forms.	•→	Inverse usage relation	Shows an inverse relationship between two components. For instance, this is a callback relation. The using component defines an interface that is implemented by the used component. To be used the component that implements the interface must register with the component that defines the interface. The dot of the arrow points to the component that defines the interface, and the arrow points to the component that implements the interface and is used.
	Nested component	A component that is nested into another component. The nested component can be used through the interface of the nesting component, or it can share an interface with the nesting component.			
	Concept, Style, Pattern	Encloses components that are part of a concept or an architectural style or pattern.			
	Application boundary	A software application has boundaries. Inside these boundaries the application's components are placed. Relationships crossing the application boundary are dependencies to the environment.			

In order to be useful for the later development phases the architecture visualization must link to the elements of the source code, e.g. classes, interfaces, or methods. The link is established by refining the usage relationships between the components and the components and the environment. When a relationship is refined the syntactical interface is defined. The syntactical interface comprises the parts of the implementation that make up the relationship. For instance, if Java is used the syntactical interface of a relationship between two components is defined by designing the Java interface that is provided or used by the components respectively. Figure 2 shows the refinement of an inverse usage relationship between two components.

B. Graphical Notation

The architecture of a software is its structure that comprise components, the externally visible properties of those components, and their relationships to each other and to the environment [3]. Also, components can be part of logical concepts, and of course they are part of a software application.

A component encapsulates functions, data, and possibly other components. It realizes parts of the functionality of the software. The encapsulated functions, data, and components can be used through interfaces that the encapsulating component provides. If one component uses another component a relationship between them is established. All components of a software together make up the application in which the requirements of the stakeholders are realized.

A logical concept in an architecture is, for instance, a solution pattern. Components that make up a solution pattern, e.g. the model, view, and controller components in a model-view-controller pattern, are part of the logical concept model-view-controller.

NOTAVIS provides symbols to visualize the elements of an architecture. Table I shows the symbols and describes the meaning of each symbol. The symbols were derived from ad-hoc visualizations because they are familiar to architects, widely used, and easy to use. Typically rectangles and arrows are used in ad-hoc visualizations [6]. Iconic representations of real-world objects are also frequently used in ad-hoc visualizations. Therefore, NOTAVIS allows to add icons to component, application, and concept symbols to give the viewer a hint of the functions and responsibilities of the visualized element.

For the visualization of the refined relationships between components and between components and their environment UML diagrams are used. UML diagrams were chosen because they are familiar to the developers and they directly map to the source code, e.g. a class in a class diagram represents a class in the source code. Also, UML diagrams integrate the NOTAVIS visualization with the UML diagrams that are regularly used in the architecture documentation [6].

An efficient visualization allows the viewer to quickly catch on the meaning of the visualization and its symbols. The comprehension process is supported by colors and by arranging the symbols in a meaningful way [25]. NOTAVIS provides color schemes that can be applied to the used symbols. The color schemes define colors that are easily distinguishable. Also, NOTAVIS defines layout rules for the arrangement of the symbols. There are two kinds of layout rules: Inter-symbol layout rules, and visualization patterns.

Inter-symbol layout rules define how symbols are arranged relative to each other. First, symbols do not overlap each other except for the concept symbol. Second, the lines of symbols should not adjoin to make them distinguishable except for nested components that share an interface with their nesting component. Third, symbols are arranged with respect to their relationships. Relationships should flow from top to bottom and from left to right. This arrangement matches the typical reading direction. Finally, bends of relationships are always right-angled, and crossings should be avoided.
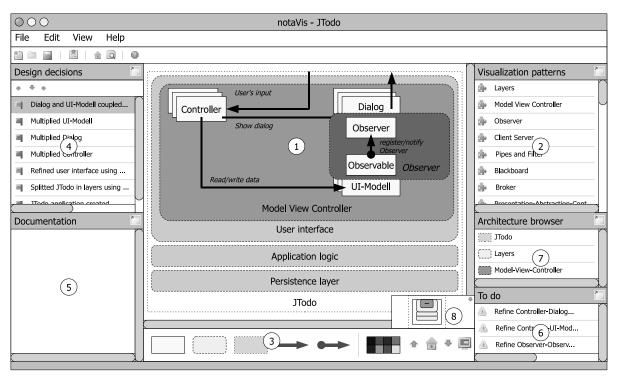
Fig. 3: The user interface of the NOTAVIS tool

Visualization patterns are graphical representations of a solution pattern. For instance, the visual representation of the model-view-controller pattern. Therefore, visualization patterns define the symbols and their arrangement to be used when a solution pattern is visualized. When a visualization pattern is added to the existing visualization the layout of the visualization pattern is preserved. The visualization pattern itself is treat like a single symbol that is positioned according to the inter-symbol layout rules in the visualization.

The functions, responsibilities, roles, and meanings of the symbols are documented. To support the architect in documenting the elements of the architecture she designs, NOTAVIS defines documentation templates. The templates define the content of the documentation for a symbol. For instance, for a component the sections name, role, function, responsibilities, dynamic behavior, states, and references are defined. The sections are either obligatory or optional. If a component originated from a visualization pattern a documentation example is provided along with the template. The example documentation facilitates the documentation task because it shows the architect how the documentation could look like [18].

C. Visualization Tool

A visualization tool is also part of NOTAVIS. The tool is designed with respect to Shneiderman's rules for the design of efficient visualization tools [22]. Figure 3 shows a prototype of the user interface (UI) of the tool. The elements of the UI are described in the following paragraphs. Each of the described elements is numbered, e.g. ①. The description below refers to that number.

The visualization panel ① is the main element of the UI. In this panel the architecture is visualized using the symbols described in section II-B. The whole visualization can be zoomed, and elements in the visualization can be refined by using an element zoom function. The element zoom function allows to vertically zoom in or out of a symbol, for instance to refine a relationship or to hide nested components. The element zoom takes place in the visualization. Surrounding symbols that are not zoomed make place for the zoomed elements. Figure 2 shows the principle of the element zoom when refining a relationship.

The visualization pattern catalog ② provides visualizations for solution patterns. A visualization pattern can be inserted into the visualization either by dragging or by double-clicking on it. When the visualization pattern is inserted the currently selected element is refined by the pattern. The visualization tool manages the insertion of the visualization pattern by rearranging the surrounding symbols to make room for the added visualization pattern. Rearranging the symbols is done in a way that preserves the existing layout as far as possible.

The toolbox ③ provides a symbol palette, the color scheme selector, a zoom function, and access to a bird's-eye view on the architecture. An architecture can be visualized freely, that is, without using visualization patterns. Free visualization is done by using the symbols from the symbol palette.

For each visualization pattern and for each symbol the architect adds to the visualization it is assumed that it resulted
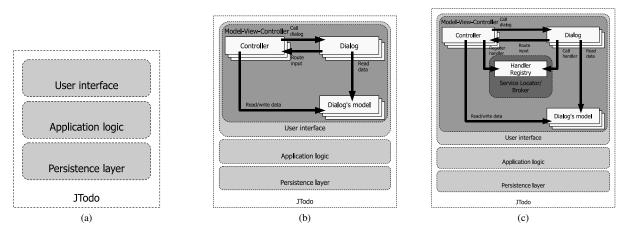
Fig. 4: Visualization of the architecture of JTodo with NOTAVIS

from taking a design decision. Therefore, additions to and modifications of the visualization are recorded in a protocol ④. The protocol can be used for analysis and documentation of the design decisions. The recorded design designs can be combined and renamed. It is also possible to undo a design decision. Actually, the undo mechanism thought about is very simple. It just removes the undone design decision from the protocol and all design decisions that were taken after the undone one. By removing the design decisions from the protocol all symbols that were added by the design decisions are removed from the protocol too.

The documentation view ⑤ shows the documentation of the currently selected symbol. The documentation view is an editor for structured documents. The structure of a document is defined by the documentation template that is provided for each symbol. If a symbol originated from a solution pattern the role of the symbol and eventually its function is known. Therefore, the documentation template of the symbol is filled with an example documentation.

Additionally, the UI provides an architecture browser ⑦. This browser lists the symbols in the visualization with respect to their nesting relationships. The to-do view ⑥ lists items that must be done to complete the visualization, e.g. refine a usage relationship or document a symbol. To-do items are retrieved from the meta model of NOTAVIS that is beyond the scope this paper.

III. EXAMPLE

This example shows the design of the architecture of JTodo. JTodo is a software application that can be used for managing to-do lists. The application is used in teaching Java programming to undergraduate students. Therefore, the application is simple and small, and it is implemented in Java. JTodo consists of about 350 lines of code that are spread across 10 classes and interfaces respectively. Java Swing components are used to implement the graphical user interface (GUI). The implementation of JTodo contains some design patterns that are typically used in Java desktop applications with a GUI.

The following sections describe the design decisions that the architect took when she designed the architecture. The description of the design process is top-down only to illustrate the refinement of the visualization using visualization patterns. The resulting NOTAVIS visualization is shown in figures 4 a–f. Elements in the text that reference elements in the visualizations are printed in *italics*.

When the architect designed the architecture of JTodo she first decided to structure the components using a layered style [21]. This design solution is visualized using the visualization pattern for the layered style (figure 4a).

Then the architect decided to structure the implementation of the GUI according to the model-view-controller pattern (MVC) [11]. The MVC pattern defines a model, view, and controller component. All these components appear in multiple forms, e.g. different views each one with its own model. In order to visualize the design solution the *User interface* layer is refined with the visualization pattern representing the MVC pattern (figure 4b).

Next, the event-handling mechanism is designed. The *Controller* components of the GUI realize the event handlers. Each event handler is used from multiple places in the source code. Therefore, the architect decides to implement the event handlers using the action technique of Swing. Access to the handlers is provided through a *Broker* component [4]. The design solution is visualized using the visualization for the *Broker* pattern (figure 4c).

The *Application logic* layer is refined after the design of the GUI is finished. The architect takes the decision to realize all use cases in one component that is accessible through a façade. The façade is realized as a singleton. The central *Application data model* component is used to store the data with which the application works. In order to visualize this design solution the architect refines the *Application logic* layer by adding the components *Uses cases* and *Application data model* to the *Application logic* layer. The *Use cases* component is refined with the visualization patterns for the singleton and the façade patterns (figure 4d).
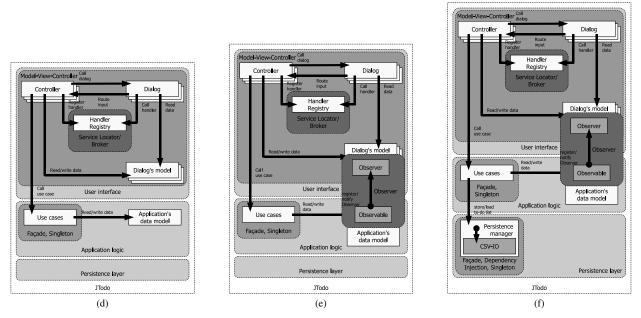
59

Fig. 4: Visualization of the architecture of JTodo with NOTAVIS

After this step, the connection of the GUI data model with the application data model is designed. The architect decides to use the observer pattern. Therefore, she refines the *Dialog model* and the *Application data model* using the visualization pattern for the observer pattern. The observer pattern overlays the *User interface* and the *Application logic* layer. Figure 4e shows the visualization of this design solution.

At last, the architect designs the *Persistence layer*. She decides to provide access to the persistence layer through a façade. The implementation of the storage provider is defined using the dependency injection pattern [10]. The architect visualizes the design solution by refining the *Persistence layer* using the visualization patterns for singleton, façade, and the dependency injection pattern (figure 4f).

After this steps the components of JTodo and their relationships are designed. In order to finish the NOTAVIS visualization of the architecture the architect has to a) refine all relationships, and b) document all symbols. After doing this, the designed architecture for JTodo can be analyzed, it can be documented, and it can be implemented.

The example showed the top-down direction of architecture design only. If the design followed a bottom-up direction components and solution patterns on a lower level of the architecture are added to solution patterns or components on a higher level. For instance, if at a low level an observer pattern is used to connect two components these two components can be added to the same layer or different layers of a layers style solution pattern that is applied later at a higher level. If the design direction is inside-out or outside-in the existing elements of the architecture are connected with the new elements.

IV. RELATED WORK

NOTAVIS is in the context of the topics software architecture visualization for design, software architecture documentation, the creation of efficient visualizations and visualization tools, and architecture design methods that use visualizations. Each of these topics is briefly introduced and the relation to NO-TAVIS is described in the sections that follow.

Today, many visualizations for software architectures exist. The visualizations are needed to give the architect and the stakeholders of the architecture a visible presentation of the invisible architecture. UML diagrams are the most prominent visualization for a software architecture. In contrast to NO-TAVIS, UML doesn't define a method to efficiently visualize an architect's mental picture of the architecture. Also, UML doesn't define a tool that allows for efficient step-by-step development of an architecture visualization. At last, UML does not define any rules for the presentation of an architecture visualization, e.g. coloring or layout rules.

The Unified Process [14] and the architecture design methods Catalysis [9] and UML Components [5] are based on the UML as graphical notation for visualizing an architecture while it is designed. These approaches define how to design an architecture. NOTAVIS in contrast doesn't define how to design an architecture. It defines how the mental picture of an architecture is efficiently visualized regardless of the process. The design methods define that UML is to be used for visualization, but they do not define how to use UML efficiently.

FMC (Fundamental Modeling Concepts) is an architecture visualization approach that is more related to NOTAVIS. FMC was developed by Keller and Wendt [16]. In FMC an architecture is visualized using three distinct views: the

concept, block, and petri-net view. The visualizations support the communication about a software architecture. Gröne [12] enhanced FMC by adding conceptual patterns. Conceptual patterns are used during the design of an architecture. Additionally, Gröne defines sample visualizations for typical architectures. Apfelbacher et al. [2] extended Gröne's work by adding a standard layout and layout rules for architecture visualizations. In contrast to NOTAVIS, FMC doesn't define visualization patterns, and therefore doesn't define a method to support the architect in visualizing his mental picture of the architecture. Also, documentation and refinement of the elements of an architecture isn't defined by FMC.

State of the art in software architecture documentation is to use multiple views. The views that are used are defined by the stakeholder's viewpoints. The views complement each other, and each one visualizes a special facet of the architecture [7], [13]. NOTAVIS provides an overview of a software architecture. Therefore, a NOTAVIS visualization can be used as one view of the architecture in an architecture documentation. Through the refined relationships of the components a NOTAVIS visualization connects to other views, e.g. a static view showing the classes of the software. Additionally, NOTAVIS records design decisions and defines the documentation of the elements in the architecture. Both, recorded design decisions and the documentation of the symbols, can be reused in the architecture documentation.

NOTAVIS builds upon concepts for the design of efficient visualizations and visualization tools. Shneiderman provides rules for efficient visualization tools [22]. Rules for good layout provides Wertheimer's classic work on the Gestaltgesetze [26]. Koning provides layout rules especially for architecture diagrams [17]. Ware provides rules to create efficient visualizations [25]. All these rules are reflected in the visualization method, the graphical notation, the visualization tool, and the meta model of NOTAVIS.

V. Conclusion and Future Work

This paper introduced the NOTAVIS concept. NOTAVIS is used to visualize a software architecture while it is designed. The motivation for NOTAVIS is the lack of a visualization concept that efficiently supports the architect in visualizing a software architecture while she designs the architecture. The NOTAVIS concept defines a visualization method, a graphical notation, and a visualization tool in order to efficiently support the architect in visualizing an architecture while it is designed. The sections above described the parts of the concept in detail.

We claim that NOTAVIS will contribute to the work of the software architect and the scientific community a concept that allows for efficient visualization of a software architecture while it is designed. The concept will contribute the following three main parts:

- Provide a method for efficient visualization of the mental picture of a software architecture that the architect builds when she designs the architecture.
- Provide a graphical notation that is designed according to the rules for good visualization design, and that can

easily be understood by the stakeholders of the visualized architecture.
- Provide a visualization tool that complements the architecture design process, and that allows for iterative architecture visualization while the architecture is designed.

Additionaly, NOTAVIS will support the documentation of the architectural elements that are designed using NOTAVIS. Thus, NOTAVIS will add to exisiting architecture design, visualization, and documentation approaches by providing a concept for efficient visualization of a mental picture of an architecture while it is designed.

More work has to be done to prove that the claims stated above are true. First of all, the elements of the NOTAVIS concept and their foundations must be developed and refined. Among other things, these elements are the graphical notation, the visualization tool and the pattern catalog. All this work is underway.

The NOTAVIS concept must be evaluated after relevant elements of the concept are finished. Two evaluation approaches are considered. One is to show that NOTAVIS can be used to visualize any architecture. The other is to prove that NOTAVIS efficiently supports the architect in visualizing her mental picture. For this purpose a controlled experiment is planned. Participants in the experiment are told to visualize an architecture that is known to them. The experiment will try to answer questions about the support NOTAVIS provides for visualizing the architecture and the quality of the created visualizations.

References

[1] B. Adelson and E. Soloway. The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360, 1985.

[2] R. Apfelbacher, A. Knöpfel, P. Aschenbrenner, and S. Preetz. FMC Visualization Guidelines. http://www.fmc-modeling.org (last access: 08/26/2009).

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley Longman, Inc., 1998.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Ltd., 1996.

[5] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[6] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's Go to the Whiteboard: How and Why Software Developers use Drawings. In *Proceedings of CHI*, pages 557–566, 2007.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Pearson Education, Inc., 2003.

[8] S. Diehl. Softwarevisualisierung. *Informatik Spektrum*, 26(4):257–260, 2003.

[9] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley, 1999.

[10] M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern. http://martinfowler.com/articles/injection.html (last accessed: 04/15/2010), 2004.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-oriented Software*. Addison Wesley Longman, Inc., 1995.

[12] B. Gröne. *Konzeptionelle Patterns und ihre Darstellung*. Dissertation, Universität Potsdam, 2004.

[13] IEEE Std. 1471. *Recommended Practice for Architectural Description of Software-intensive Systems*. IEEE Standards Association, 2000.

[14] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, Inc., 1999.

[15] R. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood. *The Process Involved in Designing Software*, chapter 8, pages 255–283. Lawrence Erlbaum, 1981.

[16] F. Keller and S. Wendt. FMC: An Approach Towards Architecture-centric System Development. In *Proceedings of the 10th IEEE International Conference and Workshop on Architectures for Software Systems*, pages 198–205, 1995.

[17] H. Koning. *Communication of IT-Architecture*. Dissertation, Universität Utrecht, 2008.

[18] S. Krauß. *Verfahren der Software-Dokumentation*. Dissertation, Universität Stuttgart, 2007.

[19] P. Kruchten. Mommy, where do software architectures come from? In *Proceedings of the 1st Intl. Workshop on Architectures for Software Systems*, pages 198–205, 1995.

[20] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag GmbH, 2nd edition, 2010.

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

[22] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[23] S. Sonnentag. Expertise in Professional Software Design: A Process Study. *Journal of Applied Psychology*, 83(5):703–715, 1998.

[24] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun. *Softwarearchitektur: Grundlagen, Konzepte, Praxis*. Spektrum Akademischer Verlag, 2009.

[25] C. Ware. *Information Visualization - Perception for Design*. Academic Press, 2000.

[26] M. Wertheimer. Untersuchung zur Lehre von der Gestalt. *Psychologische Forschung*, 4:301–350, 1923.

Towards Simulative Environment for Early Development of Component-Based Embedded Systems

Marin Orlić
Faculty of Electrical
Engineering and Computing,
University of Zagreb,
Croatia
marin.orlic@fer.hr

Aneta Vulgarakis
Mälardalen Real-Time
Research Centre,
Mälardalen University,
Sweden
aneta.vulgarakis@mdh.se

Mario Žagar
Faculty of Electrical
Engineering and Computing,
University of Zagreb,
Croatia
mario.zagar@fer.hr

Abstract—As embedded systems become more and more complex the significance of predictability grows. The particular predictability requirements of embedded systems, call for a development framework equipped with tools and techniques that will guide the design and selection of system software. Simulation and verification are two complementary techniques that play a valuable role in achieving software predictability already at early design stage. Simulation is scalable and can be very useful in debugging and validating the system design. Moreover, it can be used as a supplement to verification for visualizing diagnostic traces produced by the verification tool and for rerunning counterexamples in cases when the verification property is not satisfied.

In this paper we introduce an idea of a simulative environment for early development of component-based embedded systems. By using it, the designer can navigate and debug the design and behavior of such systems at early stages of the system lifecycle.

I. Introduction

As the complexity of embedded systems grows their development becomes more and more difficult. An appealing approach to manage the embedded systems software complexity, reduce time-to-market and decrease development costs lies in the adoption of component-based development [1]. The specific predictability demands of embedded systems, require the designer to employ a framework equipped with tools and techniques that can be applied to deal with requirements such as dependability, timing, and resource utilization, already at early-stage of development. Modeling, simulation and verification play increasingly important roles in achieving predictability, since they can help us to understand how systems function, validate the design and verify some important properties.

Simulation validates the behavior of a system for one execution path. Being relatively inexpensive in terms of execution time compared to verification, simulation is a valuable fault detection technique in early stages of system development. In general, it can be used to quickly verify a system prototype for desired properties and behavior and it can contribute to our studying of system design alternatives, in a controlled environment. Moreover, with simulation one can explore system configurations that are difficult to physically construct, and observe interactions that are difficult to capture in a live system. The ability of the simulation can be applied as a complementary activity to verification, which covers the exhaustive dynamic behavior of the system. A simulator can be used for visualizing diagnostic traces generated by the verification tool and for replaying counterexamples in cases when the verification property does not hold.

In this paper we introduce a simulative environment for development of component-based embedded systems. The simulative environment allows the designer to navigate the behavior of possibly complex and multilayered systems with respect to time and resource consumption and check behavior compliance to resource constraints. Here, we use the ProCom component model for describing the architecture of our embedded systems [2]. Additionally, we use the Remes dense-time state-based language [3] for modeling resource-wise behavior of ProCom components. Our main goal for the simulator is to be developer-friendly and usable by system modelers, engineers and developers with no prior knowledge of formal verification methodologies and tools. Finally, our intent is to present this environment to the user as a debugger with a familiar interface that will reduce the user learning effort.

The remainder of the paper is organized as follows. Section II reviews the ProCom component model and the associated behavioral model Remes needed to comprehend the rest of the work. Section III introduces our simulative environment and finally, Section IV discusses our and related approaches and concludes the paper.

II. Preliminaries

A. The ProCom component model

The ProCom component model [4] is designed to address the key requirements and modeling issues coming from the embedded system domain. In particular, ProCom considers the need for the design of a complete system consisting of both complex and distributed functionalities on one hand, and small low-level control-based functionalities on the other. Therefore,

ProCom is a hierarchical component model structured into two layers: ProSys and ProSave. The upper layer, ProSys, serves for modeling a system as a collection of active and distributive *subsystems* that execute concurrently, and communicate by asynchronous messages sent and received at typed output and input *message ports*. The lower layer, ProSave, models the internal design of subsystems as interconnected passive components with small functionality, whose communication is based on the pipe-and-filter paradigm with an explicit separation between data- and control flow. The former is represented by *data ports*, and the latter by *trigger ports*. The functionality of a ProSave component is captured by a set of *services*, which may execute concurrently while sharing only data, but no triggering. Components may be interconnected by simple connections from output- to input ports or by *connectors* that provide detailed control over data- and control flow. A ProSave component can be activated by a special type of construct, *clock*.

The ProSys and ProSave layer can be related to each other only in the lowest level of a ProSys hierarchy, where a ProSys component can be modeled out of ProSave components. For more details, see [2].

B. The REMES behavioral modeling language

The REsource Model for Embedded Systems REMES [3] is a dense time state-based behavioral modeling language, which is primarily intended to provide a basis for capturing resource-constrained and timing behavior of embedded systems. It introduces resources as first-class modeling entities that are characterized by their discrete (e.g., memory, access to external devices) or continuous (like energy) nature.

For formal analysis purposes, REMES models can be transformed into timed automata (TA) [5], or priced timed automata (PTA) [6], depending on the analysis type. We use REMES for modeling and (when translated to TA or PTA) for formally analyzing, the behavior of ProCom component-based systems.

The internal behavior of an embedded component is described by a REMES *mode* that can be either *atomic* (does not contain submodes), or *composite* (contains submode(s)). The discrete control of a mode is captured by a *control interface* made up of *entry*- and *exit* points, whereas the data transfer between modes is done through a *data interface*. Similar to other languages, each REMES mode may contain *local* or *global* variables that can be of types integer, natural, boolean, array, or clock.

Assuming that a component consumes resources, its REMES mode can be annotated with the corresponding resource-wise continuous behavior. The consumption is expressed by the first derivatives of the variables that denote resources, and which evolve at positive integer rates.

The control flow is given by *edges* (i.e., a set of directed lines) that connect the control points of (sub)modes. The continuous behavior of a mode is captured by *delay/timed* actions and their execution does not change the current mode. The discrete behavior is given by discrete actions (represented as edge annotations), which execution changes the mode. A

discrete action can be executed only when the corresponding boolean *guard* that prefixes the action body holds. A REMES composite mode may contain *conditional connectors* that enable nondeterministic selection of one discrete outgoing action to be executed, out of many possible ones. A mode may also be annotated with *invariants* that bound from above the current mode's execution time. For more details about the REMES model, we refer the reader to [3].

III. AN IDEA FOR A SIMULATIVE ENVIRONMENT

Starting from a given specification of a system (architecture-, behavior- and platform specification) we want to simulate the system behavior, with respect to timing and resource utilization. In order to achieve this goal we propose to build a simulator that would be able to accept a model fed in by a developer and allow the user to track the changes in component behavior, component activation and resource utilization. Ideally the user interface should be provided in a fashion that the user is comfortable with, in order to avoid the resistance associated with "learning yet another tool".

A. The 2+1 view of a system

To prepare a specification of an embedded system, we propose a three-fold view of the system. Architecture- and behavior specification define the desired system, and the third component - the platform specification, describes the execution platform for the system. The first two models are typically specified by the system designer, while the last one comes from the platform designer and is a common artefact shared between all systems or products based on the same platform.

Architecture specification comprises of systems, components and their connections conforming to the PROCOM component model, and the behavior specification of the system is specified with REMES models, where each REMES model is corresponding to a PROCOM component.

Platform specification declares available platform resources – CPU, available memory, energy consumption etc., within a platform profile. Once declared, resource consumption is modeled within component behavior – the resources are referenced as variables that cannot be read, only incremented and decremented. Platform profile also specifies constraints over resources. We propose to define constraints as minimum, maximum and average functions on either concrete resource values or resource changes (differences), as defined by the following grammar:

$$rc \quad ::= \quad (\ max \mid min \mid avg\ )$$
$$\text{'('} \quad (resource \mid resource')\ \text{')'}$$
$$(\ < \mid \leq \mid = \mid \geq \mid >\ ) \quad value$$

For example, the platform profile can define the constraints for CPU and memory resources such as: $\max(\text{CPU}) < 200, \max(\text{mem}) < 16384$, to define memory size to 16 Ki units and CPU usage to 200 units (or 200% usage, assuming two available CPU cores). In case of available energy the constraints could be: $\max(\text{eng}') < 50, \max(\text{eng}) < 15000$, to limit usage peaks to 50 units, with maximum total energy
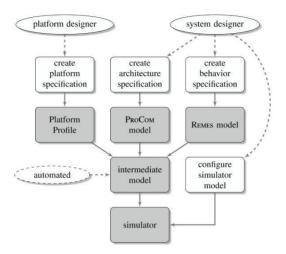
Fig. 1. Workflow steps involved in setting up the simulator

reserve of 15000 units. The choice of operators max, min and avg allows tracking and detecting peaks and spikes, as well as average resource usage.

To allow some degree of behavior parametrization, the platform profile can also define values for constants declared in REMES models. If a REMES model declares constants with no values assigned, it is assumed that such constants will finally be assigned values when a profile is added. This allows component behavior to use platform-dependant constants to declare resource usage, e.g. component initialization overhead.

During development of a system in compliance to a specific platform profile, the profile can ideally be replaced with another. Applying a new profile allows to check conformance with constraints of a different platform configuration, or a different platform version.

B. Generating the intermediate model

In order to prepare the simulation, the architecture- and behavior specifications are combined to form an integral intermediate model of the system. The purpose of the intermediate model is similar to that of object files obtained by compiling the source code of a programming language – it contains syntax-checked model information and resolved variable references. As a part of this process, expressions contained in component behaviors are translated to their corresponding abstract syntax trees and type-checked. Intermediate model joins the architecture and behavior using predefined mappings between components and behavior. Architecture and behavior are both copied to a single model namespace with the addition of a platform profile, forming a simulation specification. For example, architecture-behavior mappings map REMES variables used in REMES behavior models to input and output data ports of a ProSave/ProSys component. Connections between such data ports are converted to variable renamings (mappings between REMES variables in different behavior models) in the intermediate model.

The intermediate model is the input model for the simulator, therefore it should be complete and well-defined – references

to unknown variables or type-invalid expressions would make simulation impossible.

The process of generating the intermediate model should be hidden from the user. Whenever an architecture- or behavior specification for a component changes, it's corresponding intermediate model should be automatically generated, simplified and checked. Figure 1 gives an overview of the model translations needed to prepare for the simulation. Actors are represented with ellipses, processes with white boxes, and artifacts with gray boxes. Platform- and system specification are essentially separate design processes. Platform designer specifies platform resources and constraints through a platform profile. System designer (possibly a team of engineers) is responsible for architecture and behavior of the system and creates PROCOM and REMES models, respectively, to specify the two. An automated process transforms the three models (platform profile, architecture- and behavior models) to an intermediate model. Finally, system designer configures the simulation process with a simulator model.

C. Code generation and simulation

To simulate the system, we have utilized code generation from the intermediate model, using common model-to-text transformation tools. Compared to interpreter, generated code is simpler – the structure of the system is mirrored in generated code, and the simulator core can manipulate program objects directly, using language facilities, instead of manipulating model elements using model API or reflection. The core simulator process was designed after the time and action successor functions for timed automata [7]. As mentioned before, an alternative to this approach would be to perform the simulation using interpretation with a model visitor.

The simulator is configured with its corresponding simulator model. The simulator model contains links to intermediate model of a system, platform profile and, optionally, one or more simulator sensors. Sensors monitor data points (REMES variables or component data ports) and record value changes
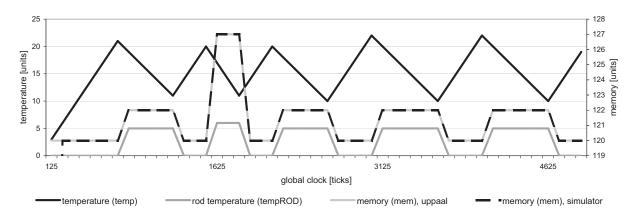
Fig. 2. An example run of a simulator for Temperature Control System

TABLE I
Mapping between common debugger objects and ProCom/Remes objects

Debugger object	ProCom/Remes object	Comment
Process	System	Container of execution for all objects
Thread	Active subsystem (ProSys)	Basic unit of parallel execution
Stack frame (method)	Component (in a hierarchy of components)	Unit of (hierarchical) sequential execution
Current instruction pointer	Active mode of component behavior	Smallest unit of execution
Variable	Mode variables	Variables and resources

when triggered. Sensors can be triggered on Remes variable change or on component trigger port activation. Data collected from sensors is displayed in the simulator environment and stored for later analysis.

To show an example of a simulator run, we can look at a temperature control system (TCS) [3]. TCS models a cooling controller for a reactor system that has two cooling rods which are used to absorb excessive reactor heat thus maintaining reactor temperature within predefined boundaries. The primary purpose of the Remes behavior model of the TCS system is to illustrate resource consumption (e.g., CPU, memory and energy) during TCS system lifetime.

Figure 2 illustrates changes in core and rod temperatures and memory consumption for a sample run in both our simulator and the one in the Uppaal * tool. Both simulators were forced to follow the same execution trace when selecting transitions to perform. Slight differences in the results can be noted for memory consumption at the very beginning of the simulation. This is due to different resource initialization strategies – TCS model in Uppaal performs resource initialization at the time the components (Uppaal processes) are activated, while our simulator adheres to the Remes execution model and performs initialization the first time a component is activated and its corresponding behavior mode is entered.

The main benefit of simulating the TCS system is the ability of the simulator to track changes for each resource separately. The current implementation of Uppaal Cora † is somewhat limited – model checking or simulation can be performed over a single monotonically rising cost variable, with occasional

*For more information on Uppaal , please visit http://uppaal.com/
†For details, visit http://www.cs.aau.dk/~behrmann/cora/

errors in the simulator. To track resource changes on Figure 2 we have manually tracked memory resource change using Uppaal and its simulator. Note that in Uppaal Cora all resources need to be combined to a single cost variable. This approach does not allow to track each resource separately. Therefore, we have used the Uppaal simulator to track memory changes for comparison with our simulator. The downside of this approach is that only discrete model transitions can update resources, as Uppaal cannot model continuous variable change. However, in the sample TCS system memory resource consumption is not affected by delay transitions but only discrete transitions of the automata.

D. Simulative environment from a user's perspective

Our main goal for the user interface is to reuse the existing UI as much as possible, and reduce the effort needed to use the simulator facilities. With this in mind, we propose to integrate the simulator with a well-known IDE platform, similar to what was done with SaveIDE [8] (Eclipse-based) and Uppaal Port [9], but reuse the platform even further and present the simulator as a debugger.

Figure 3 illustrates the user's perspective. Architecture and behavior models are created using graphical editors, as seen on the left. These models are then automatically translated into their intermediate model counterparts (in the middle). Platform profile (top right) is linked with the two using a simulator configuration model (middle right) which is used to generate the simulator classes (bottom right). The intermediate model consists of several submodels, as both architecture and behavior models can be split over several submodels, e.g. for each component in the system.

Users accustomed to modern IDEs are also familiar with the concept of debuggers – every programming language comes with one, and users are familiar with core debugging concepts. Debuggers deal with objects that model execution elements like processes, threads, stack frames, current instruction pointers and variables, and features of modern IDEs are built to support manipulation of these objects. In a system modeled by PROCOM and REMES we can distinguish active elements such as subsystems, components and modes of behavior that have some similarity to traditional execution elements. An example of relation between some common debugger objects and PROCOM/REMES objects is given in Table I. During simulation, we can manipulate these objects in the same fashion as during debugging of a program process – pause execution, switch between active elements, inspect current state and so on.

In this way we can reuse metaphors of threads, stack frames and current instruction pointers to follow the active model elements (subsystems, components and modes). Mode variables correspond to debug variables. This allows the user to navigate possibly complex and multilayered system through both its architecture and behavior in a familiar fashion – as debugging equally complex structures defined in traditional programming languages. It is our hope that this approach will increase the appeal of the simulative environment to a wider audience.

During debugging, the user needs to be able to navigate the system model(s). To enable this, the simulative environment needs to be integrated with the development environment for PROCOM – the Progress-IDE [10], [11]. Progress-IDE is built on Eclipse Platform [12] which provides rich editors and a Debug platform among other facilities. The environment is component-centric, and system and component structure are modeled in PROCOM. Work on support for behavior modeling with REMES, simulator-debugger interface and automating tasks required to generate intermediate models is in progress.

IV. DISCUSSION

A. Assumptions

There are several assumptions (or limitations) built into the simulation process, which we list in the following.

Static architecture specification – the architecture specification is implied to be static. Although PROCOM component model doesn't explicitly prohibit dynamic reconfiguration of components and their connections, the simulation assumes that components cannot migrate to different underlying hardware resources or change connections.

Simplified view of system runtime environment – processing elements such as CPUs are described by their processing speed/rate in a simple resource model. This is a simplification as modern CPUs cannot be characterized with just their clock frequency when many parameters such as processor architecture, cores, cache, pipelining, instruction dispatch and general platform architecture come into play. To some extent this assumption originates in REMES, with an intent that behavior models stay platform-independent.

Limited support for concurrency – execution parallelism in

target hardware platform and concrete component allocation to hardware nodes is not taken into consideration. It is instead left for analysis in later system design stages when deployment/allocation models are introduced.

B. Simulation strategy

In section III-A we introduced platform resource usage as incrementing and decrementing referenced resources (that behave as scalar variables). The simulator actually manipulates resources as intervals with open or closed bounds (endpoints). When a discrete transition and its corresponding action is performed, it can increment or decrement the resource variable – in effect, translating the resource interval by a specified amount. When a delay transition is performed, the resource update is calculated depending on the duration of the transition – in effect, arbitrarily changing the resource interval bounds. Resource updates therefore depend on the timed execution of model, as described in [3].

Simulation is performed in steps guided by minimum time intervals for next discrete transition, similar to global execution strategy described in [13]. In essence, the list of active modes and possible transitions is traversed to calculate time intervals till next discrete transition. From the list of intervals, a minimum interval is selected, time is let pass within this interval and the system state is updated accordingly. Transition prioritization and selection (perhaps on user intervention) can easily be performed during mode list traversal in each round.

C. Trace visualization

Simulator can be used to visualize traces generated by the verification tools and inspect counterexample states in detail. Our approach of presenting the simulator as a debugger can easily be adapted to this purpose – the process of transition selection for the next simulation round should be guided by the generated diagnostic trace, instead of usual selection rules. When following a trace, the designer can monitor state change, and in any moment divert from the generated trace to investigate a different dynamic execution path. In combination with model-checkers for verification, a proposed tool could be used for both quick prototyping at an early system design stage, and system verification of a complete system model.

D. Partially-specified systems

An interesting topic for further consideration is the possibility to simulate and analyze partially-specified systems. To illustrate, imagine a system designer working on an early system design. She has specified overall structure, but has yet to define behavior specifics of each component. However, when designing for a concrete platform, some details (e.g. component overhead resource consumption) are already known as they are dictated by the platform. With this in mind, it should be possible to simulate the early system model based on *default* component behaviors provided within platform profile. We intend to extend the platform profile with the description of default behaviors for components, the support for this in REMES remains a topic for discussion.
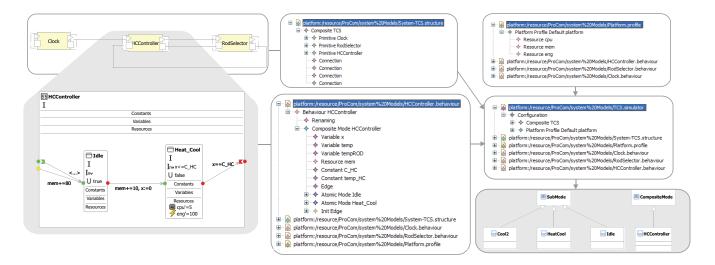
Fig. 3. From architecture- and behavior specification to simulator code: architecture- and behavior models (left) are translated into intermediate models (middle) and combined with platform profile to form the simulator configuration (right), which is then used to generate simulator source code (bottom right).

E. Related approaches

Several approaches, based on simulation models derived from UML diagrams, have beed suggested. Extended UML can be used to specify system models directly, and de Miguel et al. [14] propose extensions (with UML profiles) to express temporal requirements and resource usage. Annotated diagrams are then automatically transformed to scheduling and simulation models using Analysis- and Simulation Model Generators, respectively. Similar to our approach, application element models are transformed to simulation submodels which are combined to form an integrated simulation model. A second approach, proposed by Arief and Spiers [15] uses UML to specify system details needed for simulation with a process-oriented simulation model. System simulation is built using a predefined Java-based Simulation Modeling Language (SimML) framework with key elements such as components, processes, queues and messages.

Balsamo and Marzolla in [16], [17] propose a similar tool for simulation of performance for process-oriented systems. Annotated UML diagrams, such as Use Case, Activity and Deployment diagrams, are used to describe system performance parameters. UML model elements are closely related those of the simulator, and simulator structure and behavior closely follow the structure and behavior of the UML model. A discrete-event simulation model is automatically extracted from the diagrams, and simulation results are reported back as tagged values in diagrams.

A notable approach is that of Palladio Component Model (PCM) [18], [19]. PCM describes component-based systems with structure, behavior, allocation and usage models and derives a simulation model from them. PCM can model resource demands of discrete component actions and provide statistical results, such as processing rate, throughput and response time per component. Simulation workload is generated using domain-specific experts' knowledge contained in the usage model. A development and analysis environment is provided.

When discussing embedded systems, we should not forget to consider approaches using Matlab and Simulink, as these tools have established themselves as standard tools for embedded system design and analysis. COMDES [20], [21], [22] is a framework for hard real-time distributed control systems that uses actor diagrams representing subsystems and signals exchanged between them, and state-machines or functional block diagrams to specify behavior. COMDES translates the model to Simulink for simulation.

REFERENCES

[1] I. Crnkovic, "Component-based Software Engineering for Embedded Systems," S. G. Jean-Phillipe Babau, Jol Champeau, Ed. IESTE, Ltd, 2006, pp. 71–90. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1381

[2] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis, "ProCom – the Progress Component Model Reference Manual, version 1.0," Mälardalen University, Technical Report MDH-MRTC-230/2008-1-SE, June 2008. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1508

[3] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "Remes: A resource model for embedded systems," in In Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009). IEEE Computer Society, June 2009. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1741

[4] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A component model for control-intensive distributed embedded systems," in Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008). Springer Berlin, October 2008, pp. 310–317. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1525

[5] R. Alur and D. L. Dill, "A theory of timed automata," Theoretical Computer Science, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: citeseer.nj.nec.com/alur94theory.html

[6] R. Alur, Optimal paths in weighted timed automata, in *In HSCC'01: Hybrid Systems: Computation and Control*. Springer, 2001, pp. 49–62.

[7] W. Penczek and A. Połrola, *Advances in verification of time petri nets and timed automata: a temporal logic approach*. Springer-Verlag New York Inc, 2006. [Online]. Available: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle: Advances+in+verification+of+time+petri+nets+and+times+automata\ #0

[8] S. Sentilles, J. Håkansson, P. Pettersson, and I. Crnković, Save-IDE An Integrated development environment for building predictable component-based embedded systems, in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[9] Uppsala University, Aalborg University, UPPAAL Port, http://www.uppaal.org/port/ accessed 14/4/2010.

[10] J. Feljan, L. Lednicki, A. Petricic, and S. Sentilles, Requirements on the system design phase for Progress-IDE, Dices technical report, http://www.fer.hr/dices/resources accessed 14/4/2010.

[11] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic, Integration of extra-functional properties in component models, in *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*, I. P. Christine Hofmeister, Grace A. Lewis, Ed. Springer Berlin, LNCS 5582, June 2009. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1634

[12] Eclipse, http://www.eclipse.org/.

[13] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, Modular Specification of Hybrid Systems in Charon, *Hybrid Systems: Computation and Control, Third International Workshop*, vol. LNCS 1790, pp. 6–19, 2000.

[14] M. de Miguel, T. Lambolais, M. Hannouz, S. Betge-Brezetz, and S. Piekarec, UML extensions for the specification and evaluation of latency constraints in architectural models, in *Proceedings of the second international workshop on Software and performance - WOSP '00*. New York, New York, USA: ACM Press, 2000, pp. 83–88. [Online]. Available: http://portal.acm.org/citation.cfm?doid=350391.350411

[15] L. B. Arief and N. A. Speirs, *A UML tool for an automatic generation of simulation programs*. New York, New York, USA: ACM Press, 2000, vol. 21, no.]. [Online]. Available: http://portal.acm.org/citation.cfm?doid=350391.350408

[16] S. Balsamo and M. Marzolla, A simulation-based approach to software performance modeling, in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2003, pp. 363–366. [Online]. Available: http://portal.acm.org/citation.cfm?id=940071.940122

[17] M. Marzolla and S. Balsamo, UML-PSI: the UML performance simulator, *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 340–341, 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1348057

[18] S. Becker, H. Koziolek, and R. Reussner, Model-Based Performance Prediction with the Palladio Component Model, in *Proceedings of the 6th international workshop on Software and performance*. ACM, 2007, p. 65. [Online]. Available: http://portal.acm.org/citation.cfm?id=1217006

[19] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg, The Palladio component model, Karlsruhe, 2007. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/pdfs/reussner2007a.pdf

[20] C. Angelov, K. Sierszecki, and N. Marian, *Component-Based Design of Embedded Software: an Analysis of Design Issues*. Springer Berlin / Heidelberg, 2005, vol. 3409, pp. 1–11. [Online]. Available: http://www.springerlink.com/index/FNL8TWBABACHQ3C0.pdf

[21] C. Angelov, K. Sierszecki, N. Marian, and J. Ma, *A Formal Component Framework for Distributed Embedded Systems*. Springer Berlin / Heidelberg, 2006, pp. 206–221. [Online]. Available: http://www.springerlink.com/index/644W21160610014R.pdf

[22] N. Marian and Y. Ma, Translation of Simulink Models to Component-based Software Models, in *Proc. of the 8th International Workshop on Research and Education in Mechatronics REM'2007*, Talinn, Estonia, 2007, pp. 262–267. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1490\&rep=rep1\&type=pdf

Towards Performance Prediction of Large Enterprise Applications Based on Systematic Measurements

Dennis Westermann*, Jens Happe*

*SAP Research, Vincenz-Priessnitz-Strasse 1, 76131 Karlsruhe, Germany
Email: {dennis.westermann|jens.happe}@sap.com

Abstract—**Understanding the performance characteristics of enterprise applications, such as response time, throughput, and resource utilization, is crucial for satisfying customer expectations and minimizing costs of application hosting. Enterprise applications are usually based on a large set of existing software (e.g. middleware, legacy applications, and third party services). Furthermore, they continuously evolve due to changing market requirements and short innovation cycles. Software performance engineering in its essence is not directly applicable to such scenarios. Many approaches focus on early lifecycle phases assuming that a software system is built from scratch and all its details are known. These approaches neglect influences of already existing middleware, legacy applications, and third party services. For performance prediction, detailed information about the internal structure of the systems is necessary. However, such information may not be available or accessible due to the complexity of existing software. In this paper, we propose a combined approach of model based and measurement based performance evaluation techniques to handle the complexity of large enterprise applications. We outline open research questions that have to be answered in order to put performance engineering in industrial practice. For validation, we plan to apply our approach to different real-world scenarios that involve current SAP enterprise solutions such as SAP Business ByDesign and the SAP Business Suite.**

I. INTRODUCTION

The performance (timing behavior, throughput, and resource utilization) of a software system is one of its key quality attributes. Performance is directly visible to the user and it heavily affects the total cost of ownership (TCO) for the system provider. Software Performance Engineering (SPE) [1] helps software architects to ensure high performance standards for their applications. However, applying performance engineering for large enterprise applications is a challenging task. Today's enterprise systems are usually built on a large basis of existing software (middleware, legacy applications, and third party services) and rarely developed from scratch. Furthermore, companies continuously adapt their applications to changing market requirements and technological innovations. Thus, performance analysts have to evaluate performance aspects during the whole lifecycle of the system. Although many approaches have been published in the context of software performance engineering, none of them has achieved widespread industrial use [2]. In many cases, the sheer size and complexity of a software system hinders the application of performance engineering in practice. Especially in large enterprise applications, the performance of a system is affected by a variety of parameters. Often, these parameters are

distributed across various layers (infrastructure, virtualization, database, application server, etc.) involving many different technologies. Thus, evaluating such systems is a time and resource consuming process.

Most existing approaches use established prediction models ([3], [2]) to estimate the performance of a software system. Most of them aim for predicting the performance in early lifecycle phases, especially before system implementation. This can avoid substantial costs for redesigning the software architecture. Concerning the evaluation of already existing components, the main focus of existing approaches lies on (i) the derivation or extraction of appropriate models and (ii) the estimation of resource demands / quantitative data needed to parameterize prediction models. Approaches focusing on the first issue analyze call traces [4] or apply static code analyses [5] to extract models of software systems. Approaches focusing on the second issue (e.g. [6], [7]) use benchmarking and monitoring of systems to derive model parameters. The general drawback of these approaches is that they are bound to the assumptions of the underlying prediction model [8]. For example, if a network connection is modeled with FCFS scheduling, it won't capture the effect of collisions on the network. Another important issue is scalability. The existing approaches do not scale with respect to size and complexity of today's enterprise applications. Creating performance prediction models for those systems requires considerable effort and can become too costly and error-prone as much work has to be done manually. For the same reason, many developers do not trust or understand performance models, even if such models are available. Concerning legacy systems and third party software, the required knowledge to model the systems may even not be available at all. Here, re-engineering approaches (e.g. [9], [5]) can help. However, the large and heterogeneous technology stack of such systems makes re-engineering often infeasible.

The approach presented in this paper, handles the complexity of large enterprise applications by abstracting those parts of the system that cannot be modeled or only with high effort. The goal is to capture the dependencies between the system's usage (workload and parameters) and performance (timing behavior, throughput, and resource utilization). The technical core of the approach is the Performance Cockpit, a framework for systematic performance evaluations. Around that technical core, there are four conceptional blocks: experiment definition, automated measurements, statistical inference, and model inte-

gration. The combination of the aforementioned blocks allows the performance analyst to evaluate the performance of large systems with reasonable effort.

The contribution of this paper is an approach that combines measurement based and model based performance engineering techniques to evaluate the performance of large enterprise applications. Furthermore, we outline open research question in order to put the approach into practice. We plan to validate the approach in different scenarios that involve current SAP solutions such as SAP Business ByDesign and the SAP Business Suite.

The paper is structured as follows. Section II illustrates the research challenges that arise owing to the considered systems and the chosen approach. In Section III we present the building blocks towards meeting these challenges. Section IV describes some application scenarios of the approach. In Section V, we outline related research work. Finally, Section VI concludes the paper.

II. RESEARCH QUESTIONS

In this section, we describe the main research questions addressed by our research. The general challenge is to understand the dependencies between the system's usage (workload and parameters) and performance (timing behavior, throughput, and resource utilization). The goal is to predict the performance behavior of the system in productive operation under real-world customer load. The following questions describe the steps towards accomplishing that goal using our systematic measurement approach.

A. How can we automatically identify the performance relevant parameters?

The sheer size of the considered systems bears a big challenge when answering the question how to identify the performance relevant parameters from a set of potential candidates. In large enterprise applications, a variety of potential performance relevant parameters and parameter combinations exist which span a huge search space. Thus, we have to develop an intelligent and efficient search algorithm that combines statistical analyses with the experiment setup. However, due to the huge cause of dimensionality such an algorithm might require too much time and resources. Thus, manual reduction of the search space may be necessary before starting the actual measurements. This can be accomplished by an explicit configuration of the measurements with appropriate heuristics based on expert knowledge. Another problem is that these parameters are distributed across various layers involving many different technologies. Thus, measuring the impact of the parameters can be a time and resource consuming process. Hence, there is a need for a powerful measurement framework that can be easily adapted and applied to different systems under test. Moreover, it should support/guide the performance analyst as far as possible in order to reduce the effort and the error rate.

B. How can we efficiently quantify the influence of specific parameters on software system performance?

Once we are able to identify the performance-relevant parameters of a software system, the derivation of their actual impact on the overall performance of the system is still a complex task. Especially in large enterprise applications, a variety of parameters affect the performance of a system. As a consequence, we have to find a trade-off between providing enough measurement data for the analyses and minimizing the period of measurements. Due to the sheer size of the considered systems it is not feasible to measure each possible parameter assignment. However, the number of measurements has to be large enough to provide statistically significant results and to cover all possible effects. Thus, we have to find an optimal mixture of intelligent experiment determination and efficient statistical analysis techniques to reduce the number of measurements. Another problem is that the system under test might not deliver the necessary monitoring information for all required parameters (e.g. due to the impact of monitoring on the performance of the system). Therefore, we have to infer this information from the available monitoring data.

C. How to deal with interdependencies?

One of the strengths of our goal-oriented measurement based approach is that it abstracts from system internals, meaning it abstracts the internal resources and behavior from prediction modeling. However, this black-box approach of course involves the risk that important dependencies between components and resources inside the abstracted system are not captured. For example, consider two otherwise independent web services share the same database. Here, we have to identify the interdependencies that influence performance without detailed knowledge about the system internals. Moreover, we have to find a solution that allows the explicit integration of these interdependencies in our resulting functions.

D. How to integrate our measurement based approach with existing model based approaches?

The combination of measurement based and model based performance prediction methodologies is a promising research field towards better applicability of software performance engineering in industrial practice [8]. This combination leverages from the benefits of both approaches. However, the results derived by the measurements and statistical analyses are basically mathematical functions that capture the dependency between a set of performance relevant input parameters (independent parameters) and certain performance metrics (dependent parameters). Thus, these functions cannot be directly combined with classical performance prediction approaches (such as queuing networks or stochastic process algebras [10]), or model-driven approaches (such as PCM [11] or CB-SPE [12]). As a consequence, we have to find a way to adjust the existing solutions in order to integrate our measured functions and combine the strength of both approaches.

E. How can we apply the approach using live monitoring instead of systematic benchmarks?

An important problem of performance prediction techniques concerning their applicability in industrial practice is that in most cases they are only practicable in early phases of the software lifecycle, particularly before the system goes live. However, in practice customers do not provide the necessary information about their expected workload which is an important parameter for detailed performance predictions ([13], [14]). Even if customers provide that information, it will for sure change over time which obsolesces the predictions made in early lifecycle phases. Furthermore, the system itself evolves over time or might integrate different third party services which can only be observed after system deployment. These are all effects that are hard to consider in early lifecycle performance evaluations. Thus, we have to develop a methodology that allows us to derive/adapt our performance models during productive operation of the system. In the course of this, we have to answer the questions discussed earlier in this section in consideration of the entailed restrictions in productive operation (e.g. that measurements must not affect the performance visible to the customer or the availability of the system). This causes additional challenges to the algorithms and analysis procedures for example due to the noisy data for statistical analyses or the reduced control over the system compared to a test setup.

III. APPROACH

In the following, we present our approach that aims at understanding the performance behavior of large enterprise applications in real customer environments. The main idea is to abstract from system internals by applying a combination of systematic goal-oriented measurements, statistical model inference, and model integration. Figure 1 illustrates the major building blocks of the approach.
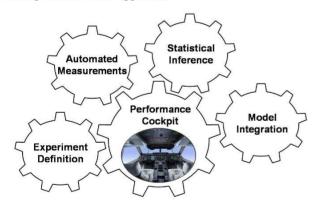

Fig. 1. Goal-oriented Systematic Measurement Approach

The technical core of the approach is our *Performance Cockpit*, a framework for systematic performance evaluations. Around that technical core, there are four conceptional blocks: *Experiment Definition*, *Automated Measurements*, *Statistical Inference*, and *Model Integration*. In what follows, we describe the building blocks of the approach in detail.

A. Performance Cockpit

Today's enterprise applications are rarely developed from scratch. On the contrary, in most cases these applications are built on a large basis of existing components such as middleware, legacy applications, or third-party services. Besides the sheer size of these systems, the resulting complexity and heterogeneity in terms of technology, distribution, and manageability complicates the application of performance evaluations. Since the performance of a system is affected by multiple factors on each layer of the system, performance analysts require detailed knowledge about the system under test. Moreover, they have to deal with a huge number of tools and techniques for benchmarking, monitoring, and data analyses. In practice, performance analysts try to handle this complexity by focusing on certain aspects, tools, or technologies within the system. However, these isolated solutions are inefficient due to the small reuse and knowledge sharing and do not provide reliable performance predictions for the overall system. The goal of the Performance Cockpit is to encapsulate knowledge about performance engineering, the system under test, and statistical analyses in a single application. Therefore, the framework implements best practices and guides the performance analyst in conducting systematic performance evaluations [15]. Moreover, the framework provides a flexible, plug-in based architecture that allows the separation of concern and supports the reuse of performance evaluation artifacts. Figure 2 illustrates the idea of the Performance Cockpit.
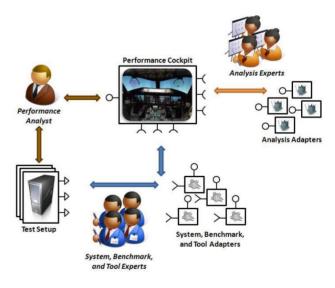

Fig. 2. The Idea of the Performance Cockpit

Each stakeholder contributes to those parts of the performance evaluation he is an expert in. The basic functionality to control the performance evaluation is provided by the framework. This plug-in based approach enables the *Performance Analyst* to reuse the adapters implemented by the *System, Benchmark, and Tool Experts* or the *Analysis Experts*, respectively. Moreover, the *Performance Analyst* can reuse adapters in multiple scenarios. Furthermore, if a component in the

system under test is changed one can easily switch the plug-ins without changing the actual measurement application. The resulting benefits are (i) less effort for setting up performance tests, (ii) better knowledge transfer, (iii) flexible measurement environment, (iv) better usability, and thus making performance evaluations available to a broader target group.

B. Experiment Definition

The approach introduced in this paper requires a huge number of measurements. Moreover, the approach should be applicable for various systems. In order to keep the approach feasible, we have to abstract from the concrete system under test and automate the measurements as far as possible. The Model-Driven Architecture (MDA) [16] is a design approach that allows to meet these challenges. We implement the MDA approach by designing a platform-independent meta-model for the definition of experiments. Experiment includes the system under test, workload, monitoring, analysis, measurement procedures, evaluation goals, etc. The definition of a platform-independent meta-model allows us to provide a single point of configuration to the performance analyst. Based on the meta-model, we can automatically create configurations for different parts of the performance evaluation (e.g. via model-to-model or model-to-text transformations). Figure 3 illustrates the idea.
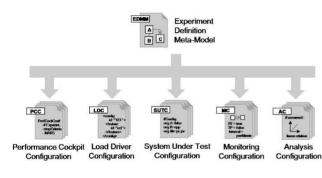

Fig. 3.   ExperimentDefinitionMetaModel

The generic *Experiment Definition Meta-Model* allows us to perform multiple evaluations in a consistent and integrated way without having effect on the flexibility of the approach. Garcia, Mora, and others successfully applied such a meta-model for software artifact and process measurements [17], [18], [19]. In our approach, we focus on the configurations necessary to perform measurements of performance metrics. This includes the following points:

- *Performance Cockpit Configuration*: Information concerning the execution of measurements by the Performance Cockpit, e.g. number of experiment runs, stop criteria for the experiments, notification event receiver, and plug-in selection (load driver, system control, monitoring, analysis, etc.).
- *System Under Test Configuration*: Information concerning the setup of the system under test, e.g. system parameters, system topology including addresses, and system control information.

- *Load Driver Configuration*: Information concerning the generation of load on the system under test, e.g. the number of concurrent users, and the variation of parameters.
- *Monitoring Configuration*: Information concerning the monitoring infrastructure and behavior, e.g. monitored metrics, sampling intervals, and hold-back time of monitoring data.
- *Analysis Configuration*: Information concerning the statistical analysis of the monitored data, e.g. analysis technique, assumptions about the expected functions, expected accuracy of the results and desired output format.

C. Automated Measurements

The experiment definition meta-model described in the previous section is an approach to automate configuration and setup of measurement environments. In this section, we focus on the automated execution of measurements. Due to the size of the considered systems and the resulting huge number of necessary measurements, the automated execution is a critical success factor. In order to automate the measurements, we have to link the different areas of performance measurement by an intelligent and efficient algorithm. If setup and configuration of the system under test and the measurement environment are completed, the following steps remain for the actual measurements: determining the actual experiment setup (i.e. how to vary the parameters in each experiment run), running the experiment and measure, and analyzing. Typically, these steps are triggered manually. For example, if performance analysts want to evaluate the performance of a middleware component, they generate or adopt a certain load profile (such as provided by the SPEC benchmarks [20]) as the experiment setup and execute it, monitor the relevant metrics and parameters during execution, and finally analyze the monitored data. Often, this process is not only manually triggered but also executed only once due to the effort involved. In our approach, we will automate this process as depicted in Figure 4.
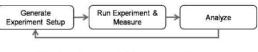

Fig. 4.   Automated Measurement Process

The Performance Cockpit generates the experiment setup, automatically deploys the load drivers on the corresponding nodes, and starts the measurements. During the measurements, the Performance Cockpit captures information about the parameters and performance metrics of interest provided by existing monitoring infrastructures. The information is aggregated and saved in the cockpit's measurement data repository. The Performance Cockpit uses the data to run its statistical analyses in predefined intervals. Depending on the results of the analysis the Performance Cockpit (i) reruns the load profile analyzed in that interval (e.g. because of insufficient monitoring information) or (ii) generates and executes new load profiles (e.g. in order to detect effects not covered by the actual load profile). The presented procedure allows us to

implement highly dynamic and efficient algorithms (such as introduced by Reussner et al. [21]). This is an essential issue towards the feasibility of our approach in large, real-world enterprise applications.

D. Statistical Inference

In the previous section, we described the automated measurement process used in our approach. In the analyses phase of the process we use statistical inference [22] to capture the dependencies between the system's usage and performance. The data collected by the monitoring is used to infer (parameters of) a prediction model. In one of our recent work [23], we derived the dependencies between the usage and the performance of a message-oriented middleware using Multivariate Adaptive Regression Splines (MARS) [24] and genetic optimization [25]. While statistical inference does not require specific knowledge on the internal structure of the system under test, it might require assumptions on the kind of functional dependency between independent and dependent variables. The main difference between the multiple inference approaches is their degree of model assumptions. For example, the nearest neighbor estimator makes no assumptions on the model underlying the observations, while a linear regression makes rather strong assumptions (linearity). Most other statistical estimators lie inbetween both extremes. In general, methods with stronger assumptions need less data to provide reliable estimates, while methods with less assumptions need more data, but are also more flexible. In our approach, the concrete technique used depends on the considered problem. For example, the identification of performance relevant parameters requires other techniques than the derivation of the actual impact of a certain parameter on a certain performance metric. Additionally, the chosen technique might differ depending on the system under test, as in some cases we might have good estimators for the underlying model while in other cases the system under test is a complete blackbox. Furthermore, the available monitoring information can influence the selection of an appropriate technique. In our approach, we aim for automatically selecting the appropriate method by the Performance Cockpit. However, in many cases this is not possible since expert knowledge is required. Thus, we enable the performance analyst to extensively configure the analyses using our experiment definition model (see Section III-B).

E. Model Integration

Model-driven performance prediction approaches (such as surveyed in [2]) allow to evaluate the performance of a software system prior to its implementation. The evaluation at design time has the advantage that it can avoid performance problems during implementation and testing, which can raise substantial costs for redesigning the software architecture. However, the drawback of these approaches is that they require detailed knowledge about the system under test in order to provide reliable results. The advantage of our measurement based approach is that we can apply it to nearly any system without detailed information about its internal structure and behavior. However, the drawback of such measurement based approaches is that the system has to be available in order to conduct the measurements. This excludes the approach from answering questions like "How would the system perform if we buy another server?".

With the combination of model based and measurement based techniques, we leverage from the benefits of both approaches while overcoming the drawbacks. We target at integrating our measured functional dependencies in the Palladio Component Model (PCM). PCM is a model based performance prediction approach that targets component based, distributed systems. It is parameterizable for parameter values as well as for the deployment platform. Moreover, PCM supports the use of performance completions. Performance completions allow software architects to annotate an architecture model [26]. Model-to-model transformations refine the annotated elements by injecting low-level performance influences, e.g. of a certain middleware [27]. The completions are parametric with respect to resource demands of the annotated element. For each implementation and each execution environment the demands have to be determined explicitly. The integration of the inferred models in PCM allows us design-time performance predictions for systems that build on a large basis of existing components. In [23], we applied our approach to build a performance completion for the message-oriented middleware ActiveMQ 5.3.

IV. Application Scenarios & Validation

The Performance Cockpit supports software architects and developers in different stages of the software lifecycle. During design time, software architects can use the Performance Cockpit to derive performance models of middleware platforms and legacy systems. In the implementation phase, developers can use the Performance Cockpit to conduct regular performance tests of their system. Furthermore, service providers can determine reliable and flexible SLAs for their services using the Performance Cockpit. In the following, we discuss possible application scenarios in more detail.

Evaluation of Design Alternatives: During design time, software architects often face architectural choices that are equivalent with respect to functionality but heavily differ with respect to performance (or extra-functional properties in general). Proper estimates of the influence of such decisions on the system's performance are essential. Design time performance predictions can avoid costly refactoring of the whole system in later development stages. However, such predictions require a detailed understanding of the middleware, third party and legacy software on which a new system is built. The Performance Cockpit enables software architects to automatically derive performance models and dependencies from systematic measurements of the systems used. Software architects can focus on the evaluation of the design decisions at hand. They can use performance completions [26] in combination with the Performance Cockpit to add low-level performance details of underlying middleware to their system under study. The

Performance Cockpit instantiates performance completions for specific implementations and configurations of a middleware platform.

In case of composable (third-party) services, software architects can use the prediction models inferred by the Performance Cockpit to assess performance characteristics of higher level services. For example, if a customer requires a special composition of business services, software architects can use the resulting model to estimate the response time and throughput of the composed business process.

Sizing and Adaptation: Sizing the underlying IT infrastructure is a critical task when deploying an enterprise application. On the one hand, the infrastructure has to provide enough resources to run the application fluently. On the other hand, purchasing and operating infrastructure resources are a substantial factor with regard to the cost-effectiveness of a companies IT landscape. Especially the trend towards providing enterprise applications as on-demand services increases the requirements on scalability and cost-effectiveness of software systems. The combination of measurement based and model based performance prediction proposed in this paper helps software architects to tailor the infrastructure to the specific needs. Thus, the approach can prevent companies from over- or undersizing their systems. In combination with appropriate cost models (such as proposed in [28]), software architects can find an efficient trade-off between costs, performance, and scalability. Moreover, the targeted performance prediction along the whole lifecycle of software systems allows early and efficient adaptations to changed workload requirements. Hence, applying the Performance Cockpit performance analysts are able to answer questions like "'Can we start another instance of application X on server Y without violating existing performance agreements?'".

Regression Benchmarking: Regression Benchmarking [29] (analogously to regressions testing) automatically executes a series of performance tests. The performance tests are executed on a regular basis (for instance after each nightly build). The results are summarized in a set of reports accessible to the developers. Thus, developers receive regular feedback on the performance of their system. They can directly assess the effect of changes in the implementation on software performance. The Performance Cockpit with its high degree of automation is well suited to support regression benchmarking. Its infrastructure allows the automatic execution of a series of measurements. The results of the measurements can be automatically analyzed and exported. Developers can specify the performance tests using the measurement configuration. In addition, the regularly executed benchmarks can be used to generate up to date performance models of the system under study.

Flexible SLA Specifications: In recent years, Service-level Agreements (SLAs) are gaining more and more attention. However, the specification of quality attributes in SLAs is still limited to fixed values (e.g., the response time is smaller than 2 seconds in 90% of all cases). A specification of dependencies between a service's usage and its performance

has not yet been established. In our approach, we propose a black-box specification of performance characteristics, i.e., the performance of a system is captured by a function of its usage. These black-box performance models do not contain any information about the system's internal structure. Including such models (e.g. as functions) in SLAs allows more fine-grained performance evaluations of service compositions and thus better service selections. For example, customers who require a scalable service can evaluate the available offers with respect to their expected usage profile and load. Furthermore, service providers can use the Performance Cockpit to determine reliable SLAs. The Performance Cockpit allows them to automatically execute the necessary measurements and derive the parameters for their SLAs in their environment. Based on the results, they can assess what performance they can provide. Additionally, they can use the models for internal capacity planning. For example, if a new customer requests one of their services, service providers can use the integrated prediction models to decide how the additional load will affect other customers.

For validation, we apply our approach on (i) a multi-tenant SAP ByDesign system and (ii) an SAP ERP system comprising a set of already existing SAP Enterprise Services which are composed to customer specific business processes. For each selected scenario, we will provide a validation of the prediction model (i.e. we will compare our predictions with observations). We also envision to evaluate the applicability by realizing one scenario in a larger case study. However, this depends on the feasibility and required overhead which we cannot assess in this early phase of the work.

V. RELATED WORK

This section presents current research dealing with measurement based performance analysis of software systems. Approaches that combine the different building blocks presented in this paper to an integrated, practice-oriented solution are rare. Liu et al. [30] build a queuing network model whose input values are computed based on benchmarks. The goal of the queuing network model is to derive performance metrics (e.g. response time and throughput) for J2EE applications. The approach applied by Denaro et al. [31] completely relies on measurements. The authors estimate the performance of a software system based on measurements of application specific test cases. However, both approaches simplify the behavior of an application, and thus, neglect its influence on performance. Jin et al. [32] introduce an approach called BMM that combines benchmarking, production system monitoring, and performance modeling. Their goal is to quantify the performance characteristics of real-world legacy systems under various load conditions. However, the measurements are driven by the upfront selection of a performance model (e.g layered queuing network) which is later on built based on the measurement results. Thakkar et al. [33] describe a framework that targets the derivation of software performance

models by a series of tests. In order to reduce the required number of actually needed test runs the authors suggest to use domain knowledge or statistical analyses techniques such as Main Screen Analysis [34] and two-way ANOVA [35]. However, the authors remain open how to design such a framework and how well the suggested statistical analyses worked in their scenario. In [36], Bertolino et al. introduce an approach that verifies QoS properties of Web service implementations before their deployment. The approach is based on the automatic generation of test-beds for the Web Service under development. The authors focus on testing a single Web service while generating mock-ups for the rest of the system. In [37] they included the test-bed generation tool in a framework called PLASTIC. PLASTIC aims at enabling online and offline testing of networked applications by providing a set of tools for generating and executing tests as well as for monitoring different metrics. An approach to generate customized benchmark applications for Web service platforms is described by Zhu et al. in [38]. The approach is based on their benchmark generation tool MDABench [39].

Many measurement based approaches rely on statistical inference techniques to derive performance predictions based on measurement data. Zheng et al. [40] apply Kalman Filter estimators to track parameters that cannot be measured directly. To estimate the hidden parameters, they use the difference between measured and predicted performance as well as knowledge about the dynamics of the performance model. In [6] and [7], statistical inferencing is used for estimating service demands of parameterized performance models. Kraft et al. apply a linear regression method and the maximum likelihood technique for estimating the service demands of requests. The considered system is an ERP application of SAP Business Suite with a workload of sales and distribution operations. Pacifici et al. [6] analyze multiple kinds of web traffic using CPU utilization and throughput measurements. They formulate and solve the problem using linear regressions. Kumar et al. [41] and Sharma et al. [42] additionally take workload characteristics into account. In [41], the authors derive a mathematical function that represents service times and CPU overheads as functions of the total arriving workload. Sharma et al. [42] use statistical inferencing to identify workload categories in internet services.

VI. Conclusions and Future Work

In this paper, we proposed an approach for performance evaluations of large enterprise applications. Our approach focuses on the observable data and does not consider the internal structure of the underlying system. We propose a combination of model configuration, automated measurements, statistical inference and model based performance prediction in order to support performance evaluations along the whole lifecycle of a software system. The approach is realized by a flexible framework called Performance Cockpit. So far, we implemented a first prototype of the Performance Cockpit

to evaluate the performance of message-oriented middleware [23].

The approach allows software architects to create performance models for applications that include components (e.g. middleware, legacy systems, third-party services) of which they do not know or understand all performance relevant internals. Performance analysts can apply the approach to answer sizing questions, to provide guarantees in SLAs, support decisions for adaptation scenarios (e.g., moving an image from one node to another), run regression benchmarks on nightly builds, and so on. Due to the separation of concern principle and the flexible, plug-in based architecture of the Performance Cockpit, the effort to execute the aforementioned tasks is kept feasible.

In our future work, we are going to gradually answer the research questions outlined in Section II. Moreover, we will further enhance the architecture and the prototype of the Performance Cockpit framework. Currently, we are applying the approach to evaluate the performance of different types of middleware (message-oriented middleware and application servers) as well as for predicting the performance of web services and web service compositions.

References

[1] C. U. Smith, *Performance Engineering of Software Systems.* Addison Wesley, 1990.

[2] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. In Press, Corrected Proof, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/B6V13-4WXC21F-1/2/602bed8a6bd384b5516b8f84ac82c672

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

[4] F. Brosig, S. Kounev, and K. Krogmann, "Automated Extraction of Palladio Component Models from Running Enterprise Java Applications," in *Proceedings of the 1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009). In conjunction with Fourth International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2009), Pisa, Italy, October 19, 2009.* ACM, New York, NY, USA, Oct. 2009.

[5] K. Krogmann, M. Kuperberg, and R. Reussner, "Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction," *IEEE Transactions on Software Engineering*, 2010, accepted for publication, to appear.

[6] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, "Dynamic estimation of cpu demand of web traffic," in *Valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools.* New York, NY, USA: ACM, 2006, p. 26.

[7] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating service resource consumption from response time measurements," in *Valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools.* New York, NY, USA: ACM, 2006.

[8] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Proceedings of ICSE 2007, Future of SE.* IEEE Computer Society, Washington, DC, USA, 2007, pp. 171–187.

[9] T. Poch and F. Plasil, "Extracting behavior specification of components in legacy applications," in *CBSE*, 2009, pp. 87–103.

[10] M. Bernardo and J. Hillston, Eds., *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, ser. Lecture Notes in Computer Science, vol. 4486. Springer, 2007.

[11] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[12] A. Bertolino and R. Mirandola, "Cb-spe tool: Putting component-based performance engineering into practice," in *Proc. 7th International Symposium on Component-Based Software Engineering (CBSE 2004)*. Springer, 2004, pp. 233–248.

[13] H. Koziolek, "Parameter Dependencies for Reusable Performance Specifications of Software Components," Ph.D. dissertation, University of Oldenburg, 2008. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/koziolek2008g.pdf

[14] H. Li, "Workload characterization, modeling, and prediction in grid computing," Doctoral thesis, 2008.

[15] R. Jain, *The art of computer systems performance analysis*. New York: Wiley Interscience, 1991.

[16] "OMG model driven architecture," Apr. 2010, http://www.omg.org/mda/.

[17] F. García, M. A. Serrano, J. A. Cruz-Lemus, F. Ruiz, and M. Piattini, "Managing software process measurement: A metamodel-based approach," *Inf. Sci*, vol. 177, no. 12, pp. 2570–2586, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2007.01.018

[18] B. Mora, F. Garcia, F. Ruiz, and M. Piattini, "Model-driven software measurement framework: A case study," *Quality Software, International Conference on*, vol. 0, pp. 239–248, 2009.

[19] B. Mora, M. Piattini, F. Ruiz, and F. Garcia, "Smml: Software measurement modeling language," in *Proceedings of the 8th Workshop on Domain-Specific Modeling (DSM'2008)*, 2008.

[20] SPEC, "SPEC's benchmarks and published results," Standard Performance Evaluation Corporation, Apr. 2010. [Online]. Available: http://www.spec.org/benchmarks.html

[21] R. Reussner, P. Sanders, L. Prechelt, and M. Mueller, "SKaMPI: A detailed, accurate MPI benchmark," *Lecture Notes in Computer Science*, vol. 1497, pp. 52–??, 1998.

[22] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data mining, Inference ,and Prediction*, 2nd ed., ser. Springer Series in Statistics. Springer, 2009.

[23] J. Happe, D. Westermann, K. Sachs, and L. Kapova, "Statistical inference of software performance models for parametric performance completions," in *6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, , June 23-25, 2010, Proceedings. To Appear.*

[24] J. H. Friedman, "Multivariate adaptive regression splines." *Annals of Statistics*, vol. 19, no. 1, pp. 1–141, 1991.

[25] D. Beasley, D. R. Bull, and R. R. Martin, "An overview of genetic algorithms: Part 1, fundamentals," 1993.

[26] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric Performance Completions for Model-Driven Performance Prediction," *Performance Evaluation*, vol. In Press, Corrected Proof, pp. –, 2009.

[27] L. Kapova and T. Goldschmidt, "Automated feature model-based generation of refinement transformations," in *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society, 2009, pp. 141–148.

[28] H. Li, G. Casale, and T. Ellahi, "Sla-driven planning and optimization of enterprise applications," in *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, pp. 117–128.

[29] T. Kalibera, L. Bulej, and P. Tuma, "Generic environment for full automation of benchmarking," in *SOQUA/TECOS*, ser. LNI, S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, Eds., vol. 58. GI, 2004, pp. 125–132.

[30] Y. Liu and I. Gorton, "Performance prediction of J2EE applications using messaging protocols," in *CBSE*, ser. Lecture Notes in Computer Science, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, Eds., vol. 3489. Springer, 2005, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/11424529_1

[31] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," *SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 94–103, 2004.

[32] Y. Jin, A. Tang, J. Han, and Y. Liu, "Performance evaluation and prediction for legacy information systems," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 540–549.

[33] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A framework for measurement based performance modeling," in *WOSP '08: Proceedings of the 7th international workshop on Software and performance*. New York, NY, USA: ACM, 2008, pp. 55–66.

[34] C. Yilmaz, A. S. Krishna, A. M. Memon, A. A. Porter, D. C. Schmidt, A. S. Gokhale, and B. Natarajan, "Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 293–302. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062515

[35] M. Sopitkamol and D. A. Menascé, "A method for evaluating the impact of software configuration parameters on e-commerce sites," in *WOSP*. ACM, 2005, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/1071021.1071027

[36] A. Bertolino, G. D. Angelis, and A. Polini, "Automatic generation of test-beds for pre-deployment qoS evaluation of web services," in *WOSP*, V. Cortellessa, S. Uchitel, and D. Yankelevich, Eds. ACM, 2007, pp. 137–140. [Online]. Available: http://doi.acm.org/10.1145/1216993.1217017

[37] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini, "The plastic framework and tools for testing service-oriented applications," pp. 106–139, 2009.

[38] L. Zhu, I. Gorton, Y. Liu, and N. B. Bui, "Model driven benchmark generation for web services," in *SOSE '06: Proceedings of the 2006 international workshop on Service-oriented software engineering*. New York, NY, USA: ACM, 2006, pp. 33–39.

[39] L. Zhu, N. B. Bui, Y. Liu, and I. Gorton, "Mdabench: Customized benchmark generation using mda," *Journal of Systems and Software*, vol. 80, no. 2, pp. 265–282, 2007.

[40] T. Zheng, C. M. Woodside, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Trans. Software Eng*, vol. 34, no. 3, pp. 391–406, 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.30

[41] D. Kumar, L. Zhang, and A. Tantawi, "Enhanced inferencing: Estimation of a workload dependent performance model," in *Valuetools '09: Fourth International Conference on Performance Evaluation Methodologies and Tools*, 2009.

[42] A. Sharma, R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, and G. M. Voelker, "Automatic request categorization in internet services," 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.1977;http://www.cs.ucsd.edu/~voelker/pubs/insight-hotmetrics08.pdf

Event-Driven Verification
in Dynamic Component Models

Claas Wilke
Institut für Software- und
Multimediatechnik,
Technische Universität
Dresden, Germany
Email: claas.wilke@tu-dresden.de

Jens Dietrich
Massey University,
Institute of Information
Sciences and Technology,
Palmerston North, New Zealand
Email: j.b.dietrich@massey.ac.nz

Birgit Demuth
Institut für Software- und
Multimediatechnik,
Technische Universität
Dresden, Germany
Email: birgit.demuth@tu-dresden.de

Abstract—**Modern component models enable software designers to design, vary and reuse multiple implementations of specific components in their software systems. Besides the implementation of specific interfaces, these components often have to fulfill contracts that are given implicitly or hidden in the interfaces' documentation. In this paper, we present *ActiveTreaty*, a novel contract and verification framework suitable to define component contracts explicitly. ActiveTreaty is an extension of our previously presented *Treaty* framework and combines aspects of static and dynamic verification by triggering component contract checks whenever configuration changes occur. We introduce a role model that supports the flexible, context-dependent management of different aspects of contract management and enforcement. Furthermore, we discuss the changes that were necessary to extend Treaty as a dynamic verification framework. Finally, a proof-of-concept implementation for the OSGi/Eclipse component model is presented.**

I. INTRODUCTION

In recent years, dynamic component models supporting the dynamic (re-)configuration of applications have become very successful. Examples include *OSGi* [1], its derivatives and the *.NET* framework [2]. Some of the largest and most complex systems are now based on these models, including application servers such as *IBM WebSphere* [3]. The question arises how these systems can be effectively verified. In practise, static verification techniques are still dominant. This includes the use of compilers and unit testing. Verification is typically performed for single components and test assemblies at buildtime. Component containers and runtime environments have little support to verify the integrity of actual *runtime* assemblies. For instance, OSGi containers merely check the type safety of classes providing services described by interfaces, and the version compatibility of the components composed. This is based on the implicit assumption that compatibility information between components can be completely described by the combination of service interfaces and dependencies between versioned packages and components.

In our previous work [4], we have argued that a more expressive contract language is required to precisely describe the relationship between collaborating components. This has resulted in *Treaty*, an expressive, component-model independent contract language. Treaty supports alternative contract arrangements (disjunction), component collaborations based on resources other than program language types, and therefore contracts that describe different aspects of component compatibility [5] including interface, behavioural and quality of service contracts. It turns out that these features are appropriate to simplify and improve the contract management in applications based on dynamic component systems such as *Eclipse* [6].

In the first version (1.*) of Treaty, contracts were represented by integrity rules. Once a contract is instantiated when collaborating components pair up, contracts are ready to be used for verification. However, Treaty makes no assumptions about *when* verification is actually performed. On the other hand, dynamic component models like OSGi have well defined lifecycle models that describe the various lifecycle states of components, and the transitions between these states. It is therefore possible to use the events signalling state transitions to trigger contract verification.

In this paper, we propose an extension to make Treaty *active* by integrating both the events that trigger contract verification, and the actions that are to be performed upon verification into the contracts. This implies that contracts are represented using *event-condition-action rules* (*ECA rules* for short) [7].

The rest of this paper is organised as follows. In the next section, we review related work. We then sketch the Treaty framework in section III. For a detailed description, the reader is referred to [4]. In section IV, we introduce *ActiveTreaty*. Afterwards, we present a prototypical implementation of ActiveTreaty for Eclipse as a case study in section V. Section VI concludes our contribution.

II. RELATED WORK

The idea of having contracts between collaborating software artifacts has been made popular by Meyer's work on *design by contract* (DBC) [8]. Contracts are represented by invariants, pre- and postconditions and added directly to the programming language source code. Several authors have adopted these ideas to other programming languages and runtime environments, including the *Java Modeling Language (JML)* [9] and *Spec#* [10]. In contrast to ActiveTreaty, these approaches have no explicit event handling, contract checks are performed when the methods are invoked. Furthermore, these DBC approaches are bound to specific programming languages,

while ActiveTreaty is programming language independent. Arnout and Meyer [11] have used "A Posteriori Contracting" to add contracts to existing software artifacts. This is similar to the introduction of legislator components in ActiveTreaty as discussed below.

The *Tamago platform* [12] supports design by contract during component development and execution. In Tamago, contracts are based on assertion logic combined with state transitions. Similar to ActiveTreaty, contracts are defined in an annotational and not-intrusive manner and the contracted components can be deployed and composed at runtime. In contrast to Tamago, ActivteTreaty focuses on an extensible contract language that can be extended to check both functional and non-functional properties.

In [13], a framework that enhance components programmed in the object-oriented language Creol with contracts described in deontic logic, is presented. Besides the definition what is obliged, permitted or forbidden, the contract language also contains statements to express what has to happen if contracts were violated. Similar to ActiveTreaty, the supported contracts can be regarded as ECA rules. But the approach is limited to components modeled in Creaol whereas ActiveTreaty provides an abstract component model in Java that can be applied to multiple component models and can be implemented in other programming languages as well.

The *Component Interaction Property Validator (CIPV)* [14] allows to define interaction contracts on CORBA components. The interaction contracts are defined using the *specification pattern system (SPS)* which allows to define orders in which specific events have to occur at runtime. For validation, the contracts are represented as finite state automata and are validated by intercepting the interactions between components during runtime. In contrast to ActiveTreaty, the CIPV focuses on interaction contracts whereas ActiveTreaty allows to express and validate multiple kinds of contracts.

Other formal contract languages have been used successfully in order to define several kinds of contracts in business computing, including *SLA contracts* [15], *business contracts* [16], [17] and the provisions for exception handling in *electronic contracts* [18]. These languages support complex event handling, in particular through the use of ECA rules.

III. THE TREATY CONTRACT LANGUAGE

Treaty is a framework to manage different types of contracts in dynamic component systems. It is based on the idea that components interact through connectors by providing and consuming services and resources.[1] Examples for (dynamic) components are Eclipse and OSGi bundles, examples for connectors are Eclipse extensions and extension points, respectively. Contracts describe this interaction. The requirements in these contracts are themselves expressed through resources that are usually provided by the consumer components. Examples are

[1] In this paper the term *resource* has the same meaning as in the OSGi component model [1]. Everything that is part of an OSGi component can be regarded as a resource. E.g., a Java class, an XML file or the component's manifest file.

Java interfaces, OCL constraint sets, JUnit test suites and XML Schemas. Suppliers of services provide other resources that have to have certain typed relationships with consumer side resources. For instance, they provide classes that must *implement* (consumer-side) interfaces and/or *pass* JUnit test suites and XML documents that must *instantiate* XML Schemas. Treaty contracts consist of these typed relationships or boolean expressions built from these relationships. Simple properties (comparisons against values) and mere resource existence conditions can also be expressed. It turns out that this representation is appropriate to express many types of contracts in existing component models. In particular in the case of Eclipse, both logically complex conditions and contracts using resource types other than Java types are required. The type system used by Treaty is defined using the Web Ontology Language (OWL) [19]. I.e., resource, property and relationship types are represented by URIs and reasoning features such as subtype and subproperty reasoning are supported. The set consisting of types, relationships and properties is called a *contract vocabulary*.

Treaty is suitable for dynamic component models using late binding. I.e., contracts can be defined without explicit reference to a supplier or a consumer. The missing resources can be referenced by function symbols representing queries to component meta data. This allows the components to advertise services by listing the respective resources in their meta data. Once the supplier and consumer are known, the contract is instantiated by executing these queries against the meta data of these components and replacing the functions by the actual resource references. This process is called *binding*. For instance, in the Treaty proof-of-concept implementation for Eclipse, these functions are *XPath* [20] expressions that are resolved against the `plugin.xml` meta data file of a supplier bundle when binding occurs.

A sample contract is shown in Listing 1. This contract defines the relationship between a component that prints dates (clock view) and a component that provides a date formatting service (date to string). The contract can be separated into three sections: a *consumer*, a *supplier* and a *constraints section*. The consumer and supplier sections define the resources provided by the components playing the consumer and the supplier role in the contract, respectively. The constraints section defines the constraints that shall be checked during the contract's verification. In this contract, suppliers can provide the service either by providing a Java class implementing a certain interface and passing two test suites, or by providing a formatting template XML document instantiating a given XML Schema. The references to consumer resources (lines 25 and 29) are XPath queries that are resolved against the `plugin.xml` meta data of the supplier component when binding occurs. The example application is an Eclipse bundle that can be installed from an Eclipse update site following the instructions on the Treaty project home page [21]. For further details the reader is referred to [4].

Figure 1 depicts the Treaty meta model. This meta model is simplified, in particular, the type hierarchy of `Conditions`
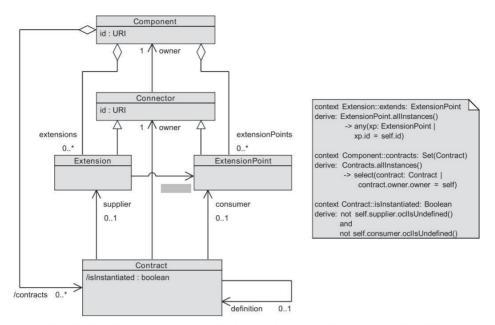
Fig. 1: The Treaty meta model (derived fields are defined using OCL [22]).

that can be used in contract definitions is missing for space reasons. Treaty supports the posteriori association of contracts with components. This is achieved by using special *legislator* (contract owner) components that provide contracts for other components. There are numerous use cases for this, including context-dependent contracts and retrofitting existing component-based systems for verification.[2]

The Treaty implementation for Eclipse also supports the dynamic composition of contract vocabularies [4]. The use case used to motivate this feature is the following. Assume a company provides a business reporting tool based on the *Velocity* template engine. They want to allow customers to add in their own reporting templates by providing components supplying resources of the respective type. The type is `MyReportingTemplate`, a subtype of `VelocityTemplate` with restrictions on the variables that can be used in the templates. These are exactly the variables the application can bind when the report is generated. To capture this in a contract, a new resource type `MyReportingTemplate` must be introduced. This would be part of the reporting tool, and used to safeguard the tool against faulty extensions. In Treaty, this is called a *vocabulary contribution*. A vocabulary contribution defines new types, properties and relationships, and their semantics. For instance, the vocabulary contribution would provide the semantics represented as a script that loads a resource and checks it by parsing it with a velocity parser and checking the variables found in the template against a predefined static list of symbols. If the `MyReportingTemplate` was declared as subtype of `VelocityTemplate` provided by another contribution, the

type reasoner could be used to delegate part of the check to the contribution defining `VelocityTemplate`.

The fact that different components can play different roles with respect to a given contract creates dependencies that need to be carefully taken into account when verification in dynamic component models is considered. For instance, if a component C_1 provides a contract for another component C_2, then the contract can only be checked if C_1 is present. I.e., there is a runtime dependency between C_2 and C_1. This dependency is implicit, but can be made explicit when the components are designed. E.g., OSGi supports direct dependencies between bundles[3] as well as (less explicit) dependencies through versioned packages.[4] A similar dependency exists between components providing contracts and components providing vocabularies for these contracts.

IV. ACTIVETREATY

One of the main challenges in verifying systems based on dynamic component models is that assemblies change at runtime. It is therefore difficult to predict the behaviour of these systems by using static verification (such as unit testing) on pre-deployment snapshots. These snapshots can only cover a small subset of possible runtime configurations, and will therefore not be able to detect many faults resulting from integration problems.

The main issue is the assumption that the behaviour of assemblies can be predicted from the known behaviour of components in other (test) assemblies. However, this is difficult as long as the contracts governing compositions are weak and ignore aspects such as semantics as this is often the case. Treaty is an approach to improve the situation by *facilitating*

---

[2]See http://www-ist.massey.ac.nz/jbdietrich/treaty/treatyout/index.html for an experiment where Eclipse has been retrofitted with contracts and verified against these contracts.

[3]Declared using the `Require-Bundle` header.

[4]Declared using the `Import-Package` and `Export-Package` headers.

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <contract>
3    <consumer>
4      <resource id="Interface">
5        <type>http://code.google.com/p/treaty/java
6          #AbstractType</type>
7        <name>clock.DateFormatter</name>
8      </resource>
9      <resource id="QoSTests">
10       <type>http://code.google.com/p/treaty/junit
11         #TestCase</type>
12       <name>clock.DateFormatterPerformanceTests</name>
13     </resource>
14     <resource id="FunctionalTests">
15       <type>http://code.google.com/p/treaty/junit
16         #TestCase</type>
17       <name>clock.DateFormatterFunctionalTests</name>
18     </resource>
19     <resource id="DateFormatDef">
20       <type>http://code.google.com/p/treaty/xml
21         #XMLSchema</type>
22       <name>/dateformat.xsd</name>
23     </resource>
24   </consumer>
25   <supplier>
26     <resource id="Formatter">
27       <type>http://code.google.com/p/treaty/java
28         #InstantiableClass</type>
29       <ref>serviceprovider/@class</ref>
30     </resource>
31     <resource id="FormatString">
32       <type>http://code.google.com/p/treaty/xml
33         #XMLInstance</type>
34       <ref>serviceprovider/@formatdef</ref>
35     </resource>
36   </supplier>
37   <constraints>
38     <xor>
39       <and>
40         <relationship resource1="Formatter"
41           resource2="Interface"
42           type="http://code.google.com/p/treaty/java
43             #implements"/>
44         <relationship resource1="Formatter"
45           resource2="FunctionalTests"
46           type="http://code.google.com/p/treaty/junit
47             #verifies"/>
48         <relationship resource1="Formatter"
49           resource2="QoSTests"
50           type="http://code.google.com/p/treaty/junit
51             #verifies"/>
52       </and>
53       <relationship resource1="FormatString"
54         resource2="DateFormatDef"
55         type="http://code.google.com/p/treaty/xml
56           #instantiates"/>
57     </xor>
58   </constraints>
59  </contract>
```

Listing 1: The contract for the clock example.

the definition of more expressive contracts in order to make assemblies more predictable. But the basic problem persists - Treaty itself can not make assemblies completely predictable.

The situation can be further improved by repeating verification on snapshots taken whenever the configuration changes. This is facilitated by the fact that many dynamic component models support component lifecycle events indicating configuration changes. An example lifecycle model for OSGi components is shown in Figure 2 [1]. OSGi containers fire `BundleEvents` to notify observers about bundle lifecycle changes. Eclipse has an additional event mechanism for bundles on top of the OSGi mechanism.
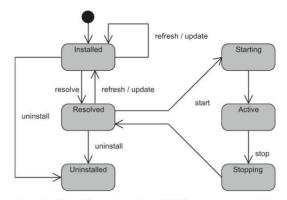

Fig. 2: The lifecycle of an OSGi component [1].

The `IExtensionRegistry` fires events when extensions and extension points of Eclipse bundles are added or removed to/from the registry. These events can be captured by `IRegistryEventListener` implementations. Furthermore, the *Eclipse Equinox* [23] OSGi implementation provides the `BundleWatcher` interface to capture similar events.

These kinds of events can be used to trigger contract verification. In particular, the events can be added to contracts themselves to make the verification policy configurable. In a similar manner, actions can be added to contracts to define behaviours being executed upon verification results. This amounts to the use of ECA rules to represent contracts. Therefore, the following issues need to be addressed:

1) An extended syntax for Treaty contracts,
2) Semantics of events and actions,
3) Dependency management between components.

A. Syntax

The contract syntax of Treaty is formally defined by an XMLSchema.[5] To support ActiveTreaty contracts, additional `trigger` and `action` elements were added to the schema. The types of both elements is the XMLSchema built-in type `anyURI`. Listing 2 shows an extended contract for the clock example (the already presented sections are not shown completely, see Listing 1 for the missing elements).

In the `trigger` section, events can be defined that shall trigger the verification of the contract. E.g., an event can represent the change of the consumer or supplier component's state such as activation (lines 3-4) or installation. ActiveTreaty allows to define multiple events for the same contract. Event sequences are interpreted as disjunction, i.e. they are implicitly connected via `OR` connectives. Thus, the contract will be verified if *any* of the described events occur. Currently, ActiveTreaty does not allow the use of a full event algebra to compose events. This is to keep the contract language and its verification as simple as possible to address core use cases.

The action section allows the definition of multiple actions to be performed either if the contract is violated or verified successfully. ActiveTreaty supports *onSuccess* and *onFailure*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <contract>
3    <trigger>http://code.google.com/p/treaty/trigger/bundle
4        #SupplierBundleStarted</trigger>
5    <onSuccess>http://code.google.com/p/treaty/action/logger
6        #LogInfo</onSuccess>
7    <onFailure>http://code.google.com/p/treaty/action/logger
8        #LogWarning</onFailure>
9    <onFailure>http://code.google.com/p/treaty/action/bundle
10       #StopSupplierBundle</onFailure>
11   <consumer>...</consumer>
12   <supplier>...</supplier>
13   <constraints>...</constraints>
14 </contract>
```
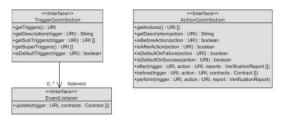
Listing 2: A contract for the clock example including triggers and actions.

Fig. 3: Trigger and Action Contribution in ActiveTreaty.

actions. E.g., an action can be the logging of a message, warning or exception (lines 5-8) or the deactivation (lines 9-10) or uninstallation of a component. In ActiveTreaty actions are implicitly connected via AND connectives. Thus, if the contract is verified successfully, all onSuccess actions are performed. If the contract's verification fails, all onFailure actions are performed. The actions are executed according to the order in which they are defined in the XML contract definition.[6] This is important as different actions are generally not orthogonal. I.e., executing the same actions in a different order could have different effects.

B. Semantics

Contracts declare actions and events using the URIs of the respective action and event types. The semantics of these types are defined in action and event contributions, respectively. This design is analogue to the design of vocabulary contributions in Treaty [4]. The contributions provide the semantics for the actions and events through classes that are associated with the respective URIs. The model comprising the interfaces that must be implemented by these classes is shown in Figure 3. The action semantics is given by the perform method defined in the ActionContribution interface. The semantics of the event is given by the implementation of TriggerContribution interface that is an *observable* (event source) [24].

Action examples are actions that log or display messages, or uninstall components. Event triggers usually act as adapters to lifecycle events of the underlying component system, which are first captured, and then registered observers are notified by calling the update methods. Another example for a trigger implementation are user interface events, allowing verification to be triggered by GUI interaction (e.g., a button to verify a selected set of contracts).

Note that ActiveTreaty does not have a "declarative semantics". In particular, by mapping actions to methods, actions cannot be guaranteed to be free of side effects. An example, where this can become problematic, is when actions themselves perform component lifecycle changes like uninstalling

or passivating components. This may trigger further contract checks. This cascading verification could even result in infinite loops and threaten system stability. While those scenarios are unlikely, they are possible. Some problems could be prevented through meta verification (e.g., checking the ECA contract rules for potential circular dependencies). This problem is currently not addressed in ActiveTreaty.

Events can be defined hierarchically. E.g., an event BundleStarted could have the subevents Consumer-BundleStarted and SupplierBundleStarted. This means that contract verification is triggered if any of the subevents occurs. The use of event hierarchies is similar to the handling of hierarchical exception events in languages like Java.

ActiveTreaty supports events to be defined as *default events*. Default events are global and used to specify events that can cause contract verification but have not to be declared in contracts. This is useful to define events that can be used to verify all selected contracts–e.g., caused by a user interface component that displays a set of contracts and provides such an operation. In ActiveTreaty, *default actions* can be defined as well. Actions can have up to four different kinds of default states:

1) Actions can be declared as default before the verification of a set of contracts (*before action*). E.g., an action to log a message how many contracts will be verified caused by a specific event.

2) Actions can be declared as default after the verification of a set of contracts (*after action*). E.g., to log a message how many contracts have been violated during a triggered verification or to show a similar message in a GUI dialog.

3) Actions can be declared as default after a contract's successful verification (*default onSuccess*). E.g., to log a message that the contract has been verified successfully.

4) Actions can be declared as default after a contract's failed verification (*default onFailure*). E.g., to log a warning including the verification's stack trace.

Please note, that before and after actions cannot be defined in a contract itself but in an action contribution. In a contract only onSuccess and onFailure actions can be declared.

C. Dependency Management

As briefly discussed earlier, the management of dependencies between components having different roles with respect

---

[6] Actions defined as default actions are performed after performing all actions defined explicitly (default actions are introduced and explained below).

to a contract is a problem that must be addressed when managing contracts. The main problem is to ensure that actions, events, types, relationships and properties used in contracts are available. For instance, consider the clock example introduced above. The contract defines a relationship between an XML document and an XML schema (line 45-47 in Listing 1). This relationship (http://code.google.com/p/treaty/xml#instantiates) is defined by a vocabulary. The vocabulary is itself provided as a resource by an Eclipse bundle extending a vocabulary extension point of the ActiveTreaty implementation. If this contract is to be verified, the bundle must be available. In particular, the bundle knows the semantics of this relationship and can compute a boolean indicating that the respective XML document indeed instantiates the schema. The bundle would use a validating XML parser for the actual implementation of this functionality.

This example shows that there is a runtime dependency between the component that defines the contract and the component that defines the vocabulary used in the contract. In OSGi, this dependency would imply that the bundle defining the vocabulary is visible (its classloader accessible) to the bundle defining the contract.

Abstracting from this example, contracts have additional (implicit) references to extensions providing the building blocks for contracts. Figure 4 depicts the resulting extended Treaty meta model for ActiveTreaty.

To define dependencies between components, we first associate components with roles with respect to components. The use of roles is similar to how roles are used in *design patterns* [24], [25]. Here, a class can play a role like factory or product in the context of a certain (factory) pattern instance. However, a class can be a factory in the context of a factory instance, and a product in the context of *another* factory instance. Formally, roles are subsets of $Component \times Contract$ defined as follows:[7]

1) $(com, con) \in CONSUMER \Leftrightarrow$
   $con.consumer.owner == com$
2) $(com, con) \in SUPPLIER \Leftrightarrow$
   $con.supplier.owner == com$
3) $(com, con) \in LEGISLATOR \Leftrightarrow$
   $com.contracts.includes(con)$
4) $(com, con) \in$
   $VOCABULARY\_CONTRIBUTOR \Leftrightarrow \exists x \in$
   $Extension \wedge con.vocabularies.includes(x) \wedge$
   $extension.owner == com$
5) $(com, con) \in EVENT\_CONTRIBUTOR \Leftrightarrow \exists x \in$
   $Extension \wedge con.events.includes(x) \wedge$
   $extension.owner == com$
6) $(com, con) \in ACTION\_CONTRIBUTOR \Leftrightarrow \exists x \in$
   $Extension \wedge con.actions.includes(x) \wedge$
   $extension.owner == com$

A component can have multiple roles with respect to the

[7]We use a compact syntax combining PL1 with navigational OCL for simplicity.

same contract. For instance, in the clock example, the contracts are defined in the component that has the extension point, and therefore the $LEGISLATOR$ and the $CONSUMER$ roles overlap.

Using these definitions, each contract gives rise to a legislator component and sets of consumers, suppliers, legislators, and vocabulary, event and action contributors. To execute a contract, the following dependencies must be satisfied for each component in the respective role:

1) $\forall c_1, c_2 \in Component : (\exists con \in Contract :$
   $(c_1, con) \in LEGISLATOR \wedge (c_2, con) \in$
   $VOCABULARY\_CONTRIBUTOR \Rightarrow (c_1, c_2) \in$
   $DependsOn$
2) $\forall c_1, c_2 \in Component : (\exists con \in Contract :$
   $(c_1, con) \in LEGISLATOR \wedge (c_2, con) \in$
   $EVENT\_CONTRIBUTOR \Rightarrow (c_1, c_2) \in$
   $DependsOn$
3) $\forall c_1, c_2 \in Component : (\exists con \in Contract :$
   $(c_1, con) \in LEGISLATOR \wedge (c_2, con) \in$
   $ACTION\_CONTRIBUTOR \Rightarrow (c_1, c_2) \in$
   $DependsOn$
4) $\forall c_1, c_2 \in Component : (\exists con \in Contract :$
   $(c_1, con) \in LEGISLATOR \wedge (c_2, con) \in$
   $CONSUMER \Rightarrow (c_1, c_2) \in DependsOn$
5) $\forall c_1, c_2 \in Component : (\exists con \in Contract :$
   $(c_1, con) \in LEGISLATOR \wedge (c_2, con) \in$
   $SUPPLIER \Rightarrow (c_1, c_2) \in DependsOn$

In many cases, the supplier also depends on the Consumer. An example where this is the case is if the consumer describes a service to be provided by a supplier using a (Java) interface. The supplier needs to provide a class implementing these interface, and needs access to the interface to do this. E.g., in OSGi, the supplier bundle would need access to the class path of the consumer bundle.

V. CASE STUDY: ACTIVETREATY FOR ECLIPSE

To prove the concepts of ActiveTreaty, the framework has been implemented and tested as a contracting language for Eclipse bundles and their extension points. Some implementation details and challenges are depicted below.

A. Application Structure

The Eclipse implementation of ActiveTreaty consists of three different features, the contract framework (net.java.treaty.eclipse.feature.group), the clock example (net.java.treaty.eclipse.example.clock.feature.group), and a system example that retrofits some standard Eclipse extension points with contracts (net.java.treaty.eclipse.example.system.feature.group). The framework feature also contains views to monitor contracts, vocabularies and event and action contributions. The application can be easily installed using the Treaty update site [21].
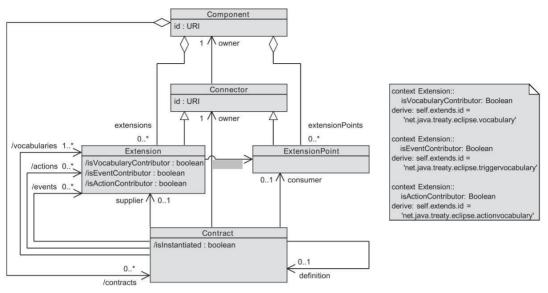
Fig. 4: The ActiveTreaty meta model with additional derived associations and attributes. The derived associations Contract.vocabularies, Contract.events and Contract.actions are not defined using OCL because these expressions would be too complex. A Contract references all vocabulary, event and action contributions whose types or other elements are used in the contract's definition.

B. Contracts and Vocabularies

The Eclipse application includes nine contracts, including four "system contracts" used by ActiveTreaty itself for contracts, vocabulary, event and action contributions, one contract for the clock example, and another four contracts for several standard Eclipse extension points. The number of instantiated contracts depends on the Eclipse configuration and is significantly higher. The retrofitted standard Eclipse extension points have been choosen because we know that the respective contracts are often violated by extensions.

The contracts use five different vocabularies - Java, XML, JUnit, OWL and a vocabulary for basic types such as strings and numerical types.

C. Events

The application defines two default events that are used to trigger all or a selected sets of contracts. The clock example uses the event http://code.google.com/p/treaty/trigger/bundle#SupplierBundleStarted. This trigger is activated when the supplier bundle (i.e., the Eclipse bundle that has an extension extending the data formatter extension point) is activated. This means that the contract is checked automatically if a new bundle providing a date formatting service is installed and started. The respective event basically acts as an adapter between the OSGi bundle event mechanism and ActiveTreaty's internal event system.

D. Actions

The Eclipse application contains an action vocabulary that adds default actions to pop up information dialogues showing verification results. Furthermore, there is an action vocabulary that defines actions to log the results of contract verifications to the standard Eclipse error log. A third experimental vocabulary provides actions to deactivate or uninstall OSGi bundles.

E. Lazy Loading and Verification Side Effects

Eclipse bundles can be activated lazily. If an Eclipse bundle is resolved, it remains in the lifecycle state `resolved`, since a class from the bundle is required (e.g., shall be instantiated). The class loading process causes a change of the bundles's lifecycle state. The bundle becomes `active`. This mechanism–known as *Lazy Activation Policy* [1]–is one example for a side effect that can occur during the verification of ActiveTreaty contracts.

Consider two bundles `A` and `B`. `A` defines an extension point `A.xp1` with a contract and `B` extends this extension point with an extension `B.x1`. The contract defined on `A.xp1` contains a trigger that states that the contract shall be verified if one of the contracted bundles (the consumer `A` or the supplier `B`) becomes active. Furthermore, the contract states that a class provided by `B` must implement an interface provided by `A`. First, both bundles are deactivated but `resolved` by the runtime environment. At a certain point during runtime, `A` becomes active and the contract is verified. ActiveTreaty has to load the implementation class from `B` and thus, the bundle becomes activated and changes its state to `active`. This change triggers the verification of the contract again.

In this simple example the side effects are harmless. The same contract is checked twice and the bundle `B` changes its lifecycle state. But the example illustrates that ActiveTreaty's contract verification can have side effects that can alter the system's state, can cause further contract verification and can even lead to endless verification loops.

F. Dependency Management

The dependency problems illustrated in subsection IV-C are solved by using explicit bundle dependencies. For instance, the legislator component containing contracts for standard Eclipse extension points declares explicit dependencies to ActiveTreaty, to the bundles providing vocabularies, actions and events used in the contracts, and to the (standard) bundles owning the extension points representing the consumers.

VI. CONCLUSION

We have presented ActiveTreaty, a contract framework for software components based on ECA rules. The presented implementation based on Eclipse shows such that this framework can be easily added to an existing component model and can be used to safeguard evolving component assemblies by triggering verification whenever assemblies change. A detailed case study contracting existing Eclipse extension points [26] proofed that Eclipse lacks the support for such a contract mechanism and that such a contract mechanism is indeed sensible.

There are some open questions though. The actions used in our implementation so far are fairly simple - user notification and logging. While this is useful, it would be desirable to go one step further and to use corrective actions. In particular actions such as uninstalling faulty components and roll-back component upgrades. However, this causes another problem. Actions interfering with the lifecycle of components will indirectly trigger lifecycle events. These events can trigger new contracts and the system could go into infinite loops. Therefore, verification of contracts is needed. One way of doing this is to build a dependency graph between actions and events and check this graph for circular dependencies. The events used so far in ActiveTreaty are flat. The expressiveness of the contract language could be further increased by allowing complex events such as events composed using event algebras.

In our current research we plan to implement an OCL vocabulary [22] for ActiveTreaty. The major challenge is to develop an action contribution that triggers verification when the actual service invocation occurs. We have conducted a first case study to investigate a suitable extension to support such an OCL vocabulary in ActiveTreaty [27].

VII. ACKNOWLEDGEMENTS

REFERENCES

[1] "OSGi Service Platform Core Specification - Release 4, Version 4.2," June 2009, http://www.osgi.org/Download/Release4V42.

[2] "MS .NET Framework," http://msdn.microsoft.com/netframework/.

[3] "IBM Software - Websphere," http://www.ibm.com/websphere.

[4] J. Dietrich and G. Jenson, "Components, contracts and vocabularies - making dynamic component assemblies more predictable," *Journal of Object Technology*, vol. 8, no. 7, pp. 131–148, 2009, http://www.jot.fm/issues/issue_2009_11/article4/index.html.

[5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[6] "The eclipse project," http://www.eclipse.org/.

[7] K. R. Dittrich, S. Gatziu, and A. Geppert, "ACT-NET - the active database management system manifesto: A rulebase of ADBMS features," *SIGMOD Record*, vol. 25, no. 3, pp. 40–49, 1996.

[8] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[9] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *FMCO*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 342–363.

[10] M. Barnett, K. R. M. Leino, and W. Schulte, "The spec# programming system: An overview," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., no. 3362. Springer-Verlag, Berlin/Heidelberg, Germany, January 2005, pp. 49–69.

[11] K. Arnout and B. Meyer, "Finding Implicit Contracts in .NET Components," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., vol. 2852. Springer, 2003, pp. 285–318.

[12] H. Belhaouari and F. Peschanski, "A lightweight container architecture for runtime verification," in *8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008, Selected Papers*, ser. Lecture Notes in Computer Science, M. Leucker, Ed., no. 5289. Springer-Verlag, Berlin/Heidelberg, Germany, October 2008, pp. 173–187.

[13] O. Owe, G. Schneider, and M. Steffen, "Components, objects, and contracts," in *Proceedings of the Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*. New York, NY, USA: ACM, September 2007, pp. 95–98.

[14] Y. Jin and J. Han, "Runtime validation of behavioural contracts for component software," in *Proceedings of the Fifth International Conference on Quality Software (SQIC'05)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2005.

[15] A. Paschke, J. Dietrich, and K. Kuhla, "A logic based SLA management framework," in *Proceedings of the Semantic Web and Policy Workshop (SWPW) at 4th Semantic Web Conference (ISWC 2005)*, 2005, pp. 68–83.

[16] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal, "A unified behavioural model and a contract language for extended enterprise," *Data Knowl. Eng.*, vol. 51, no. 1, pp. 5–29, 2004.

[17] G. Governatori and Z. Milosevic, "A formal analysis of a business contract language." *Int. J. Cooperative Inf. Syst.*, vol. 15, no. 4, pp. 659–685, 2006.

[18] B. Grosof and T. Poon, "SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions," in *Proceedings of the 12th international conference on World Wide Web (WWW'03)*. New York, NY, USA: ACM, 2003, pp. 340–349.

[19] D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview," W3C, W3C Recommendation, Feb. 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[20] "XML Path Language (XPath) 2.0. W3C Recommendation 23 January 2007," 2007, http://www.w3.org/TR/xpath20/.

[21] "The Treaty Project," http://code.google.com/p/treaty/.

[22] "Object constraint language, version 2.2," Object Management Group (OMG), Needham, MA, USA, February 2010. [Online]. Available: http://www.omg.org/spec/OCL/2.2/

[23] "The eclipse equinox project," http://www.eclipse.org/equinox/.

[24] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 1993, pp. 406–431.

[25] D. Riehle and T. Gross, "Role model based framework design and integration," in *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 98)*, 1998, pp. 117–133.

[26] J. Dietrich and L. Stewart, "Component contracts in eclipse - a case study," in *Submitted and accepted for the 13th International Symposium on Component Based Software Engineering (CBSE2010)*, June 2010.

[27] C. Wilke, "Model-based Run-time Verification of Software Components by Integrating OCL into Treaty," Diploma Thesis, Technische Universität Dresden, Germany, September 2009.