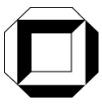
Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

Fourteenth International Workshop on Component-Oriented Programming

Herausgeber: Ralf Reussner, Clemens Szyperski und Wolfgang Weck

Autoren: Colin Atkinson, Ivica Crnkovic, Oliver Hummel, Henning Groenda, Luka Lednicki, Ana Petričić

Interner Bericht 2009-11



Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

ISSN 1432-7864

Fakultät für **Informatik**

Preface of the Workshop on Component-Oriented Programming (WCOP) 2009

WCOP 2009 is the fourteenth event in a series of highly successful workshops. WCOP has been a successful and steady event at ECOOP from 1996 to 2007. Since 2008, WCOP was moved to be part of the CompArch federated event, as the unique opportunity to associate with this established conference in the field of components and architecture, aligning well with the WCOP focus. This move proved successful in gaining submissions of high quality.

COP has been described as the natural extension of object-oriented programming to the realm of independently extensible systems. Several important approaches have emerged over the years, including component technology standards, such as CORBA/CCM, COM/COM+, J2EE/EJB, .NET, and most recently software services and model-driven development. Additionally the increasing appreciation of software architecture for component-based systems, as in SOA, plays an important role including the consequent effects on organizational processes and structures as well as the software development business as a whole.

COP aims at producing software components for a component market and for late composition. Composers are third parties, possibly end users, who are not able or not willing to change components. This requires standards to allow independently created components to interoperate, and specifications that put the composer in a position to decide what can be composed under which circumstances. On these grounds, WCOP'96 led to the following definition:

> A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties.

Where WCOP'96 focused on the fundamental terminology of COP, the subsequent workshops expanded attention to the many related facets of component software. In the future, WCOP will become the Doctoral Symposium of CompArch, as a kind of entry level workshop in the field of software components and architectures.

WCOP 2009 accepted 3 papers covering the broad field of COP: Self-adapting components, component certification, and domain-specific component models. The organisers cordially thank Klaus Krogmann for preparing the proceedings volume.

Ralf Reussner, Clemens Szyperski, Wolfgang Weck

Workshop Co-organizers

Ralf Reussner
Institute for Program Structures and Data Organization
Universität Karlsruhe (T.H.)
Am Fasanengarten 5
D-76128 Karlsruhe, Germany
E-mail: reussner "at" ipd.uka.de
Web: http://sdq.ipd.uka.de

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

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

WCOP Website

http://research.microsoft.com/en-us/um/people/cszypers/events/wcop/

Table of Contents

Reconciling Reusability and Trustworthiness through Self-Adapting Components

Colin Atkinson and Oliver Hummel
Chair of Software Engineering
Faculty for Mathematics and Computer Science
University of Mannheim
{atkinson, hummel}@informatik.uni-mannheim.de

***Abstract*— Assembling applications from reusable components can significantly reduce the time and costs involved in software engineering, but it also raises some significant verification and validation challenges. Traditionally, there has been a tension between the trustworthiness of systems (i.e. the level of confidence that can be placed in their correct execution) and the level of flexibility and ease with which they can be assembled from reusable components, especially at run-time. This effectively made it difficult for developers to strive for both at the same time. In this paper we present a strategy for alleviating this problem based on a combination of technologies developed to address each individually – built-in testing, developed to enhance the trustworthiness of systems, and dynamic interface adaptation, developed to reduce the effort involved in deploying a component in a new context. After first describing the two technologies individually, we then explain the synergy between them and present a vision of how, together, they can be used to make components self-adapting. The overall benefit is to introduce a component paradigm in which the compatibility of components is determined by their behavior (i.e. semantics) rather than the form of their interfaces (i.e. syntax).**

I. Introduction

One of the key problems in assembling new applications from components[1] built by different vendors is ensuring that they "fit" together – that is, that they have the same understanding of the contracts through which they interact. A fully specified contract that includes operation pre- and post-conditions (according to the principles of design by contract as applied in [1]) in theory solves this problem because it specifies both the syntactic form and the semantics of the operations through which components interact. However, real components are rarely if ever accompanied by such complete semantic specifications. For the time being, the most that is usually available is a specification of the syntactic interface offered by a component.

Syntactic interface specifications are a mixed blessing, however. On the one hand they allow the typing integrity of operation invocations to be automatically checked, thus reducing the chance of mismatched calls at run-time, but on the other hand the identifiers they specify make it much more difficult to match the interface provided by a component to that required by a client. If strict name matching is required (as is the case with compiler-driven interface checking) the reused component and using components have to agree on the precise names of operations. In practice, this normally means that the using component has to be written after the used component has been acquired, usually by hand. It also greatly reduces the chance of finding reusable components in the first place since most component search engines still rely on programming language identifiers to find candidate components [13].

In effect therefore, with today's technologies, developers of component-based systems are caught in the dilemma of having to choose between trustworthiness and reusability (i.e. high flexibility in reusing components). If they want high trustworthiness they need to accept syntactic interface matching with the drawback that the interacting components have to agree on the exact names of operations. On the other hand, if they want more flexibility via the ability to invoke components without the strict requirement for a-priori name matching, they need to accept the drawback of a higher chance of runtime invocation errors (and thus lower trustworthiness). As figure 1 exhibits, therefore, with current component technologies high trustworthiness and high reusability tend to be mutually exclusive. The area at the bottom left of the figure illustrates the combination of trustworthiness and reusability that is attainable with current technologies.

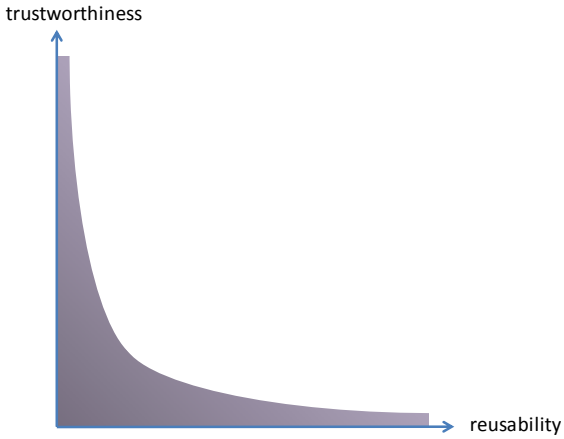


Fig. 1. Tension between trustworthiness and reusability.

[1] By "component" here we mean any well contained subsystem of a system that offers a well defined interface. This also includes services in the sense of service-oriented architectures (e.g. Web services).

We believe it is possible to improve this situation by integrating two technologies that were originally developed to enhance dependability and reusability separately. One is the built-in testing approach [2], enhanced by the test sheet test specification technique [3], which were originally developed to complement syntactic interface checks with semantic ones. The other is a dynamic interface adapter and associated automatic permutation generator, originally developed to support semantic component retrieval based on test-driven search [9]. While each individually makes an effective contribution to the specific problem for which it was originally developed, we believe that when combined there is a strong synergy between them which can significantly reduce the tension between trustworthiness and reusability.

In the next two sections, we first describe these two technologies independently. While section II talks about built-in testing and test sheets in more detail, section III introduces the approach for automated interface adaptation as developed for our implementation of test-driven reuse. Section IV explains the synergy between the two and presents our vision of how self-adapting components might mitigate the challenge of component integration in the future. We conclude our paper in section V with a summary of our contribution.

II. BUILT-IN TESTING

The idea of building tests into components to increase the trustworthiness of component-based systems was first proposed by Wang et al. [2] back in 1999. The original motivation was to replicate the self-testing capabilities often provided by hardware components within software components. However, during the course of several successive projects this concept has undergone several important changes.

First, during the Component+ project [12], the notion of built-in tests being mainly for self-testing evolved into the notion that they are most suited to contract testing. This was motivated by the recognition that the failure of a previously successful self-test rather reveals a problem in a component's environment than in the component itself. Thus, it was considered more efficient to directly test a component's environment (i.e. the components with which it interacts) in the form of so-called contract tests.

Second, during the MORABIT project [5], the idea that built-in tests are code modules, "hardwired" into the run-time code, evolved into the notion that they are actually test specifications which are executed by the run-time environment (i.e. the component container platform) rather than as part of a component's application logic. The adjective "built-in" therefore was reinterpreted to mean that the tests were packaged with the component (i.e. built into the distribution packages such as a jar file) rather than physically embedded in its source code.

Third, during the ECOMODIS project [3][2], the idea that built-in tests are behavioural specifications rather than implementation enhancements was consolidated and a new user-friendly approach for defining, applying and visualizing the effects of such tests was defined. So called "test sheets" extend the idea of specifying tests in a tabular fashion (initially popularized by the FIT approach [6]) with the expressiveness and flexibility of the well known spread sheet metaphor [3].

A. Run-Time Testing

During this evolution process, a fundamental characteristic of built-in tests that has remained the same is the notion that they are executed at run-time when components have been deployed in their final production environment. The tests may be executed in a special testing phase at the beginning of a system's deployment, but in general it is possible to execute built-in tests at any appropriate time while a system is running, for example, when a change is made to the population of components in the system (dynamic reconfiguration).

To illustrate this idea, let us assume that a component MU (for Matrix User) needs to be connected to a Matrix component which is required to pass the following test –

```
public void testMatrixMultiplication() {
  Matrix mtx1 = new Matrix(2, 3);
  Matrix mtx2 = new Matrix(3, 2);
  mtx1.set(0, 0, 1.0);
  mtx1.set(0, 1, 2.0);
  mtx1.set(1, 0, 2.0);
  mtx1.set(1, 1, 3.0);
  mtx1.set(2, 0, 1.0);
  mtx1.set(2, 1, 4.0);
  mtx2.set(0, 0, 1.0);
  mtx2.set(0, 1, 2.0);
  mtx2.set(0, 2, 3.0);
  mtx2.set(1, 0, 3.0);
  mtx2.set(1, 1, 2.0);
  mtx2.set(1, 2, 1.0);
  mtx1 = mtx1.mul(mtx2);
  assertEquals(mtx1.get(0, 0), 7.0);
  assertEquals(mtx1.get(1, 1), 10.0);
  assertEquals(mtx1.get(2, 1), 10.0);
  assertEquals(mtx1.get(2, 0), 13.0);
}
```

Listing 1. Matrix multiplication test example (JUnit).

This test is written in Java using the assertion features provided by JUnit. However, the form of the test and language used to describe it are not important. What is important is that this test partially defines the semantics that the required component must offer. In the terminology of the test-driven development approach popularized by agile development (as in Extreme Programming [8]), this defines when the Matrix component is "fit for purpose". Although it does not completely define the required semantics in the sense that the test would cover all possible behaviours of the component (over all possible combinations of input values) the test does characterize the essential behaviour that is expected. In other words, it characterizes the contract that exists between MU and a reusable Matrix component. Tests of this form still represent the only practical way of automatically establishing fitness for purpose at run-time.

[2] Which is still ongoing at the time of writing.

In the MORABIT approach, when an instance of MU is connected to an instance of a Matrix component that is supposed to provide the specified functionality, the run-time environment will take this test and apply it to the Matrix on behalf of the MU component. If the Matrix component passes the test, the likelihood that the system will behave as intended increases. If it fails, however, there is obviously a serious problem and the system needs to be reconfigured.

B. *Test Sheets*

As mentioned above, test sheets provide a more concise and easy-to-read description of tests which are freed from a particular programming language's syntactical idiosyncrasies. They can therefore be understood by non-IT personnel such as managers and domain experts who are not familiar with programming languages. Figure 2 shows the test sheet form of the matrix multiplication test case illustrated in Listing 1.

	A	B	C	D	E	F
1	Matrix	create	2	3		
2	Matrix	create	3	2		
3	F1	set	0	0	1.0	
4	F1	set	0	1	2.0	
5	F1	set	1	1	3.0	
6	F1	set	2	0	1.0	
7	F1	set	2	1	4.0	
8	F2	set	0	0	1.0	
9	F2	set	0	2	3.0	
10	F2	set	1	0	3.0	
11	F2	set	1	1	2.0	
12	F2	set	1	2	1.0	
13	F1	mul	F2			
14	F13	get	0	0		7.0
15	F13	get	1	1		10.0
16	F13	get	2	1		10.0
17	F13	get	2	0		13.0

Fig. 2. Matrix multiplication test sheet.

Each row of the test sheet represents a single method invocation, like most of the lines in the JUnit version of the test. The first column (A) identifies the called object, the second column (B) identifies the called method, and the other columns to the left of the so called invocation line – the double-line dividing column E from F – represent the input parameters. These are "filled up" from left to right according to the number of parameters required by the method. The column to the right of the invocation line (F) represents the results returned by the invocation. In a simple test sheet such as that in Figure 2 the rows are executed sequentially from top to bottom just like a sequence of statements in a Java method. There can be more complex forms of test sheets, however, in which the execution order is controlled by a special "behavioural part" at the bottom of the table (see [3] for further details).

Figure 2 is actually an input test sheet, which defines the test data and expected return values. Thus, the values that appear in column (F) represent the results that are expected from the invocation of the operation in the specified sequence. When the test sheet is actually applied to a component, a new version of the sheet is created – a so called output test sheet – that illustrates whether the actual returned value matched the expected value. If it did, the corresponding cell is coloured green. If it did not, the corresponding cell is coloured red and the actual returned value is shown along side the expected value.

III. Dynamic Interface Adaptation

The motivation for a dynamic interface adaptation engine came from the desire to automatically apply tests to candidate components retrieved by a search engine in order to filter out those that do not have the desired behavior [4]. This, in turn, was motivated by the growing practice of test-driven development [7] in which tests for components are written before their implementations. This makes it possible and worthwhile to search for components which pass the specified test (and thus by definition are fit for the specified purpose) before effort is invested in developing an implementation from scratch. In effect, therefore, the test specification serves as the query definition for search engines which are able to perform semantic rather than pure syntactic matching during the search process [13]. A schematic representation of this approach is shown in the following figure.

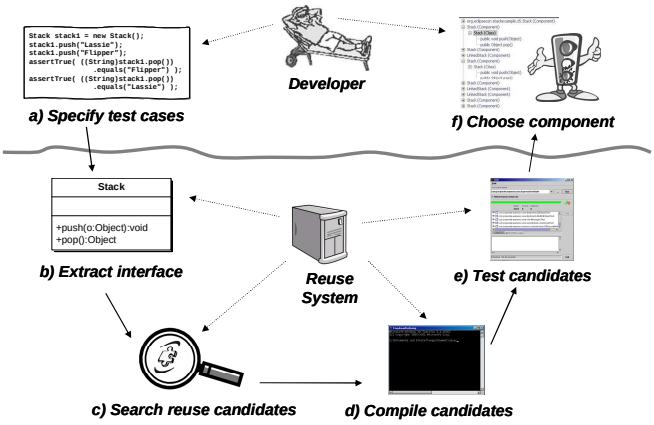


Fig. 3. Required steps for the test-driven reuse of a component.

A key challenge that had to be overcome to implement the above test-driven search algorithm was to automatically adapt candidate components to the interface expected by the test case in order to be able to test them in step (e). Limiting the initial search for candidates to be tested to those that exactly match the desired interface is not desirable because the number of matching components would be far to low and many semantically suitable classes that happen to have the wrong names for their methods would not be considered. In principle, any component with the right set of method signatures (i.e. set of input and output parameter types, cf. [14]) is a potentially matching candidate, but using this criterion to determine the set of candidates to be tested generally results in too many candidates. In effect, it represents the other end of the spectrum. In practice, the optimal balance is a middle-way

solution that combines an interface search based on signature and loose identifier matching with subsequent testing.

Even in this case, however, there are many ways in which a given candidate could be called by the test case. For example, if an operation has several parameters of the same type these could be passed in various orders. To realize the described test-driven search technology, therefore, a technique was needed to systematically explore all possible mappings between the interface expected in the test and the set of operations offered by a component, as characterized by their signatures. A so-called permutation engine was designed to solve this problem automatically [9].

A. *Static Interface Adaptation*

In order to explain how the adaptation engine works, in this section we first briefly review the classic strategy for adapting the interface of one component to the needs of another. The most well known version of this approach is the Adapter Pattern defined in the Gang of Four pattern catalog [11]. The following figure shows the structure of the so-called object adapter, the variant of this pattern that is suitable for object-oriented languages such as Java that do not support multiple inheritance.

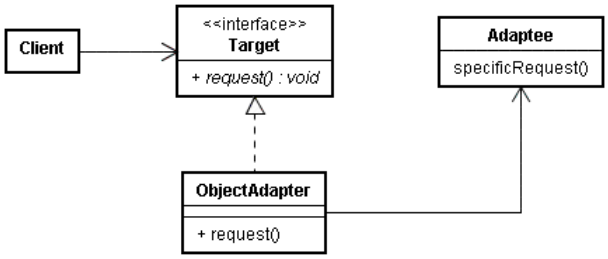


Fig. 4. Object adapter structure.

The underlying idea depicted in figure 4 is simple. The Client on the left relies on the specified Target interface, but this unfortunately, is not provided by the Adaptee component that is supposed to be reused. Thus, an ObjectAdapter needs to be created in order to "translate" calls to the Target interface to those actually supported by the Adaptee.

Although this pattern looks simple at a first glance there are some situations where it is not applicable – One common example is when the class to be adapted (the Adaptee) contains methods with parameters or return values of the class's type. A clear example of such a situation is illustrated in figure 5 that shows how method calls are to be mapped onto an Adaptee in case of a Matrix component. Take for instance the mult method of the Matrix candidate component on the right hand side. Since it is not aware that it is going to be adapted it expects a parameter and returns an instance of the Matrix class type. In other words, it naturally expects and delivers objects of its own type rather than of the type of the adapter. Thus, the simple forwarding of parameters and return values that is normally carried out by the adapter is no longer sufficient in this case.

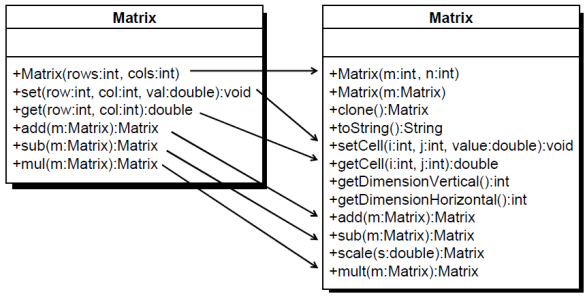


Fig. 5. Matrix adaptation example.

The key idea to overcome this limitation is to manage the relationship between adapter and adaptee objects inside the adapter in order to guarantee a traceable 1:1 relationship between the two. In other words, for each instantiated adapter object there needs to be an adaptee object and vice versa. Furthermore, the adapter needs to keep track of all these relations in a lookup table. This enables the adapter to replace the appropriate adapter object with an adaptee object and vice versa whenever necessary. However, as the solution of this issue is not the focus of this paper we refer the interested reader to [10] for further details on exactly how the classic object adapter is extended to overcome this limitation.

B. *Adaptation Engine*

Figure 5 also demonstrates how our adaptation engine, in the presence of ambiguous operation signatures, maps the operations specified by an extended version of the test cases in listing 1 onto an adaptee [9]. As an example, consider the getCell method of the right Matrix component for instance. Given the signature `int x int -> double` by itself, it is not clear in what order the method expects the two integer parameters. Furthermore, the signature `Matrix -> Matrix` appears three times within its add, sub and mult methods which makes it difficult to decide to which one the calls to the adapter object on the left should be forwarded. Although a human developer might be able to solve these issues by reading and understanding the documentation of the component, he or she will typically also test it afterwards to establish a specific level of trustworthiness. Nevertheless, if there is no adequate documentation or perhaps less expressive operation names as it is often the case today with web services, even a skilled human engineer might be forced to use trial and error to establish which operation relations and parameter orders are expected. When we implemented our automated adaptation engine we initially experimented with heuristics to mimic the human understanding of this task, but realized quickly that only the systematic evaluation of all possible permutations through testing ultimately guarantees the successful discovery of the correct mapping (if one exists). As mentioned before, this engine was able to automatically resolve the solution shown in figure 5 based on an extended version of the test case in Listing 1. Further results can be found in [9].

IV. Self-Adapting Components

As mentioned previously, the technologies described in the previous two sections were developed independently for separate purposes - built-in testing was realized to enhance the trustworthiness of components by arranging for the "fitness" of their server components to be tested at run-time and the automated adaptation engine was implemented to enhance the reusability of components by dynamically mapping invocations to matching operations. Each therefore makes an individual contribution to simplify the development of component-based systems. However, we believe that by using the two together it is possible to overcome the traditional tension between trustworthiness and reusability, and to develop systems which are not constrained to the area of low trustworthiness and/or low reusability depicted in the bottom left corner of figure 1.

The synergy between the two technologies arises in the context where built-in tests can be used to verify that the components to which a given component has been connected in a particular component-based system are able to meet its expectations. In such a situation, which is illustrated in figure 6 between a MatrixUser component and a Matrix component, a test is used to establish the latter's fitness for purpose from the perspective of the former.

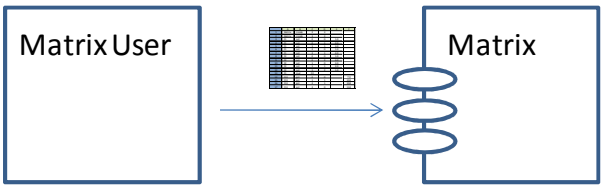


Fig. 6. Contract testing scenario.

In the standard view of built-in testing as developed in the MORABIT [5] and ECOMODIS [3] projects, it is implicitly assumed that there is no need for interface adaptation between the communicating parties. In other words, the MatrixUser in this scenario is assumed to "know" the names and signatures of the operations offered by Matrix and thus to issue only syntactically correct operation invocations. However, as mentioned above, this is unrealistic in practice since independently written components rarely agree exactly on the operation names used in their mutual interface, and if this assumption is enforced in practice it drastically reduces the flexibility of the whole approach and thus the reusability of components.

In this context, an automated adaptation engine can be used to alleviate this problem, because as long as a description of the semantics of the contract is available (as it is in the case of built-in testing) the engine is able to dynamically adapt the actual interface of a component to match that expected by the user (if it is at all feasible). In other words, the relationship between a client and server component in a built-in testing environment, where a test is used to capture the semantics of their contract, provides exactly the conditions needed for an automated adaptation engine to work. Thus, in the context of built-in testing, such an approach would always be able to provide the illusion that a server component exactly matches the expected interface of the client, even if in actual fact it does not. To put it another way, once a component has ensured it can deliver the required semantics (as determined and verified by built-in tests), a self-adapting component equipped with this technology would be able to figure out the correct wiring of input parameters to their internal representations based on a set of test cases delivered by its clients.

The idea that naturally follows from this observation is that by including self-adapting behaviour in all reusable components, and adopting a component deployment approach in which built-in tests are used to verify semantic compatibility, the problem of interface adaptation can be solved automatically. This is the justification for our claim that when used together the two technologies can enhance trustworthiness and reusability at the same time. A schematic picture of such a self-adapting component (a component with the ability to automatically adapt its interface using a built-in adaptation engine) is shown in figure 7.

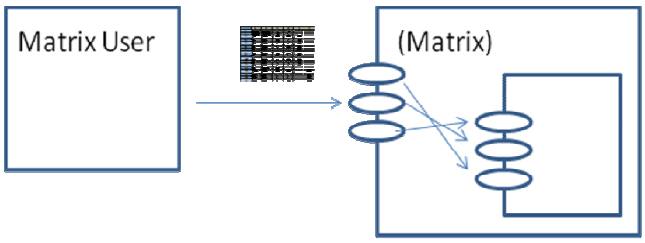


Fig. 7. Schematic representation of a self-adapting component.

A further natural extension of this approach is to use test-driven search to find suitable components in the first place. In fact, there is little conceptual difference between searching for components in the conventional way and then using built-in testing and automated adaptation to verify their fitness for purpose or to use a test-driven search up front and only considering semantically suitable components for reuse. This raises the issue of the value of making components self-adapting when a test-driven component search engine is already able to readily deliver fixed adapters for components that match the specified test by saving successful operation and parameter mappings. However, fixed adapters of this kind are obviously only of use as long as the communication pattern between components stays fixed. In long lived systems where component populations and configurations are changed over time (e.g. SOAs) and in systems which have been deliberately built to be "adaptive" at run-time, such concrete adaptation is too rigid. Building interface adaption facilities into the components or services of such systems clearly enhances the ease with which they can be reconfigured in a trustworthy way.

Since our proposed approach is able to perform the adaptation automatically, it can be brought into play whenever the compatibility of interacting components is cast into doubt, including dynamic changes to the run-time component architecture (i.e. dynamic reconfigurations). In principle, as long as

it is clear that a component is able to semantically fulfil its contract, all that is needed to re-establish syntactical compatibility is to rerun the adapter to re-configure the correct internal wiring.

V. Conclusion

In this position paper we have made the case for combining what at first sight appear to be two unrelated technologies – semantic contract validation through built-in testing and dynamic interface adaptation – in order to lower the tension that usually exists between two highly desirable characteristics of components and component based systems – trustworthiness and reusability. The ultimate effect of this synergy is to support a composition paradigm in which interface incompatibilities are automatically resolved, and component compatibility is determined solely by behaviour. The only prerequisite for such a paradigm is the use of tests to describe the behaviour required by a component. Since these tests will therefore play an increasingly critical role in the overall composition process, it is important that they are easily understandable and writeable by human engineers. We therefore believe that platform independent test description techniques such as test sheets will play an increasingly important role in the future of component-based development.

One objection that is often brought up when test cases are used to specify components is that they only cover samples of a component's input space and thus are not able to guarantee correctness. This is indeed an issue, but even in defect testing it is impossible to cover the full input space of components and thus to be sure that they behave exactly as expected. In all practical software engineering projects, therefore, developers have to decide on a set of test cases (i.e. a sample) that they feel is "good enough" for the purpose in hand. For example, in agile development methods based on test-driven development, tests are used as the ultimate judge of a component's "fitness for purpose", even though they cover just a fraction of all possible inputs. The key is to use a sufficiently large and well defined sample in order to gain the desired level of confidence that the component has the behaviour desired. Exactly the same criteria used to determine the adequateness of tests in test-driven development can be used to judge the adequateness of the built-in tests in our approach to component-based development.

Another interesting question related to this approach is its potential scalability - how well does the approach work when the components become more complex than in our matrix example and perhaps depend on other components as well? This question is not limited to our approach alone, but is a general problem for all software systems composed hierarchically from components and objects. Our ultimate vision, of course, is that all component contracts will be governed and verified using the built-in testing and self-adaptation approach described in this paper. Thus, for large systems composed of many components, the obvious approach is to start the contract verification process at the leafs of the component or object tree - in other words, with those units that do not require any other units to function – and to then work up towards the root. This strategy is, for example, nicely explained in [15] in the context of object-oriented development and can easily be used for the composition of self-adapting components as well.

References

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B., Paech, J. Wust, J. Zettel, *Component-based Product Line Engineering with UML*. Addison Wesley, 2002.

[2] Y. Wang, G. King, H. Wickburg, "A method for built-in tests in component-based software maintenance", in *IEEE International Conference on Software Maintenance and Reengineering*, IEEE Computer Science Press, 1999.

[3] C. Atkinson, D. Brenner, G. Falcone, M. Juhasz, "Specifying High-Assurance Services", in *IEEE Computer*, vol. 41, no. 8, 2008.

[4] Hummel, O. and C. Atkinson: "Extreme Harvesting: Test-Driven Discovery and Reuse of Software Components", in *International Conference on Information Reuse and Integration*, 2004.

[5] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, D. Suliman, B. Paech, "Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing", in *Information Systems Frontiers*, vol. 9, no. 2-3, Springer 2007.

[6] R. Mugridge, W. Cunningham, *FIT for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005.

[7] K. Beck, *Test-Driven Development by Example*. Addison-Wesley, 2003.

[8] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[9] O. Hummel, "*Semantic Component Retrieval in Software E*ngineering", Ph.D. dissertation, Faculty of Mathematics and Computer Science, University of Mannheim, 2008.

[10] O. Hummel, C. Atkinson, "The Managed Adapter: A Pattern for Dynamic Component Adaptation", in *International Conference on Software Reuse*, 2009, submitted for publication.

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

[12] C. Atkinson, H. Gross, "Built-In Contract Testing in Model-driven, Component-Based Development", in *First International Working Conference on Component Deployment*, 2002.

[13] O. Hummel, C. Atkinson, W. Janjic, "Code Conjurer: Pulling Reusable Software out of Thin Air", in *IEEE Software*, vol. 25 no. 5, 2008.

[14] A.M. Zaremski, J.M. Wing, "Signature Matching: A Tool for Using Software Libraries", in ACM Transactions on Software Engineering and Methodology, vol. 4, no. 2, 1995.

[15] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterarive Development (3rd ed.). Prentice Hall, 2004.

Certification of Software Component Performance Specifications

Henning Groenda
FZI Forschungszentrum Informatik,
Software Engineering
10-14 Haid-und-Neu-Straße,
76131 Karlsruhe, Germany
Email: groenda@fzi.de

***Abstract*—In software engineering, performance specifications of components support the successful evolution of complex software systems. Having trustworthy specifications is important to reliably detect unwanted effects of modifications on the performance using prediction techniques before they are experienced in live systems. This is especially important if there is no test system available and a system can't be taken down or replaced in its entirety. Existing approaches neglect stating the quality of specifications at all and hence the quality of the prediction is lowered if the assumption that all used specifications are suitable does not hold. In this paper, we propose a test-based approach to validate performance specifications against deployed component implementations. The validation is used to certify specifications which in turn allow assessing the suitability of specifications for predicting the performance of a software system. A small example shows that the certification approach is applicable and creates trustworthy performance specifications.**

I. Introduction

Performance prediction plays an important role in the development and evolution of complex component-based software systems. For example, their use in the development phase enables software engineers to predict the performance of different design alternatives and hence select the best alternative. In the deployment phase, these specifications guide the selection and sizing of an appropriate execution environment and on the deployment of components within this environment. In the maintenance or evolution phase, performance predictions allow to examine the effect of modifications on the performance and reduce the probability of discovering unwanted behavior in live systems. This detection is especially important if there is no test system and test data available and a system can't be taken down or replaced in its entirety. In general, prediction techniques for the performance of component-based system use specifications for the components' performance-relevant behavior as well as information on their assembly for their predictions.

The reliance on such performance predictions requires a predictable assembly of components. A predictable assembly of components in turn requires trust in both, the prediction method as well as the component performance specifications. Prediction method should be based on a sound and falsifiable scientific theory. Component specifications should state objective, test- and verifiable information on the performance-relevant behavior. Testing by independent certification authorities according to a procedure checked by experts can ensure the trustworthiness of such specifications. Due to the necessary effort the testing of complex software systems needs to be limited in most cases to certain parameter ranges. For example testing a performance specification regardless of deployment environment and usage profile requires in general prohibitive effort. Explicit statements about such limits enable statements about quality of the prediction and aid in identify potential risks.

Existing performance prediction approaches rely on the capability of software engineers to select suitable performance specifications of a component. Research focused on validating prediction approaches under the assumption that the specifications were suitable for the situation at hand. Validating performance specifications is seen as manual activity during the generation of the specifications. Reasoning about the suitability or quality of an existing performance specification is however complicated if there are no explicit validity statements connected to it. Especially if specifications should be reused in different context, for example because of late composition, missing validity statements increase overall efforts as specifications have to be recreated and revalidated.

In this paper, we propose an approach to certify performance specifications and explicitly state the quality and limitation of the specifications. The approach is based on a test-based validation of the specifications against deployed component implementations. Statistical reasoning is used to assure trust in the performed validation. This enables the verifiability by third parties and eases testing the validity. Additionally, it aids in the protection of interests and trade secrets in marketplaces as it is sufficient to publish the certified specifications.

The applicability of the approach to create trustworthy specifications is demonstrated in a small example. The necessary validation effort as well as the certifiable quality are shown and discussed.

The paper is structured as follows. Section II contains a description of the information necessary to validate performance specifications. The envisioned certification process is describes in Section III. Section IV contains a description of existing influencing factors on software performance and which specification is chosen in the presented approach. Section V shows a small certification example. Section VI points out and

discusses related work. Section VII concludes the paper.

II. Validating Performance Specifications

Validation of specifications is often made based on test suites. For example, the functional validation if an application server fulfills the Java platform enterprise edition requirements. Each test in such a test suite provides a simple pass or fail outcome by comparing expected and experienced results.

However, if performance is considered the comparison of expected and experienced results is much more difficult. Performance is for example influenced by input parameters, internal state, the performance of other required services, resource contention, and the deployment platform. Depending on the use measurement method and granularity, performance measurements often influence the performance itself. Additionally, experience shows that measured execution times are scattered in productive environments. Reasons are manifold, e.g. physical effects like heat or age, or caching of data, concurrent processing of background services, and the precision of measurements can cause scatter. Many performance specifications only refer to average or worst case considerations. These are for example used in Service Level Agreements (SLA) to ensure adequate performance.

Validating performance specifications stating the exact execution time requires determining for which environments which variations between expected and experienced results are acceptable. For example, a specification can state that for a given environment an undisturbed execution takes between 4.9 and 5.1 ms, or equally that the undisturbed execution takes 5 ms with an acceptable deviation of $\pm 2\%$. The definition of acceptable variations between expected and experienced result depends on two independent factors besides the environment. One factor is the range of input parameters for which the specification is valid and the second factor is the precision of the specified resource demands. All factors are explained further in the following paragraphs.

A. Hardware Environment

The hardware environment describes the hardware and its configuration for which the specification is valid. If a specification is validated for an environment it may also be valid for other environments. This is due to the fact that prediction approaches can allow transferring specifications from one environment to another. The environments for which transfers are supported can be considered as an equivalence class and validation can be limited to one instance of this class. An example in which transfers work quite well is if only the speed of a processor changes between two environments. However, automated and trustworthy reasoning on this would require machine-readable specifications and validating the specification transfer capabilities of the prediction approach. It is assumed that software engineers using the prediction approaches know the limits of the transferring capabilities and are aware of their implications. The validation of transfer capabilities is not part of the presented approach. An example for a hardware specification is a Pentium IV Northwood with 2.6 Ghz, Intel Chipset, SATA hard drive, and 2 GB of RAM.

B. Software Environment

The software environment describes the software and it's configuration for which the specification is valid. The same capabilities and limitation apply as for hardware environments. An example for a specification is a Sun JRE 1.5 on Windows XP SP3 with the JRE configured in server mode.

C. Input Range

The input range specifies the parameter ranges of parameterized specifications for which the specification is valid. For example, if a specification has the parameter *file size* and it is validated for file sizes between 3 and 50 MB this should be stated as input range.

D. Resource Demand Precision

The resource demand precision states how exact resource demands of the specification are within the input range in relation to the implementation executed in the specified environment. The specification is considered valid as long as specified and measured resource demands are below the deviation threshold defined as resource demand precision. A certain deviation will exist in most cases. This is either due to the measurement method or due to the fact that performance specifications are abstractions of the real behavior. Depending on the kind of the specified demand (expected constant value, gauss distribution, arbitrary distribution) different validation estimators can be used. A *Validation Estimator* can for example be an interval, the mean, maximal deviance, variance, or the Mann-Whitney-Wilcoxon test. As statistical testing is used for validation, the validity statement can only be made with a certain probability. For example the certainty that the distribution of traces from the specification and the implementation are from the same distribution can be 95%. An example of a resource demand precision for an expected resource demand of 100 ms and a maximal deviance of $\pm 10\%$ will still accept a measured value 105 ms.

III. Certification of Performance Specifications

Our certification approach implements a test-based validation of performance specifications against deployed component implementation. The certification process is sketched in figure 1 and explained in the following paragraphs.

At the beginning, the component creator issues a validation request to the certification authority. He has to provide the component specification, the respective implementation, and the validity statements which should be used for validation. Validity statements contain the information discussed in chapter II.

An evaluator within the certification authority then assesses the validity of the specification. Therefore, execution of the following steps is necessary.

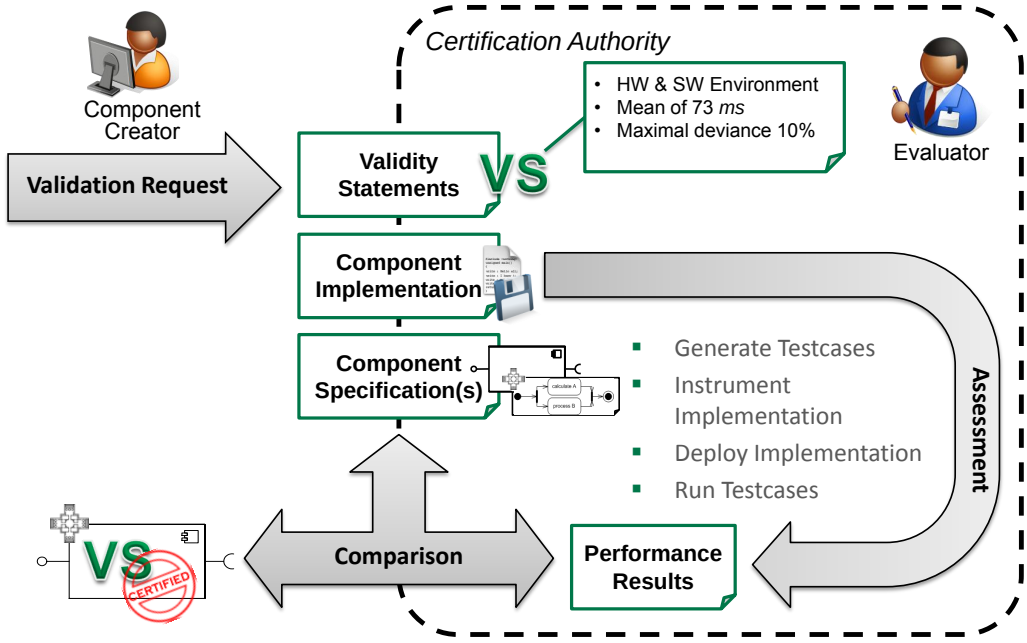


Figure 1. Overview about the Component Certification Process

A. Generate Testcases

In this step, test cases are generated automatically which are used to evaluate the specification. Automatic generation is chosen to ensure reproducibility of validation results and reduce the necessary human effort. If random numbers are used, a pseudo-random number generator has to be used and initialized. The initialization number has to be stored along the with the validation information.

Both directions, specification against executed implementation and vice versa, are checked by the test cases. The forward direction checks if the statements in the specification are correct, for example that a resource demand depends linearly on an input parameter. The backward direction checks that the implementation does not contain more dependencies than the ones specified, for example unspecified calls to external services, another sequence of calls, or unspecified dependencies to return values of calls. White-box code-analysis techniques are used to discover these dependencies. Additionally, input parameters which should not influence performance according to the specification should be varied randomly for the test cases to raise the chance of detecting otherwise undiscovered effects. However, without exhaustive checking of all possibilities the chance remains that there are some values which would lead to an invalid specification. This is mitigated by publishing the sample size for the measured demands so Software Engineers can judge for themselves if the risk is acceptable.

If there are required services, mock-ups for the required services are created to allow checking return-value dependencies. As with input parameters, ranges for the return values should be specified for which the validation is executed.

B. Instrument Implementation

The implementation itself must be instrumented in order to log the resource demands within the component and correlate them to the resource demand of the specification. Depending on the measurement method either measurement facilities are directly inserted into the code, platform functions of an adapted application or virtual machine container are applied, or operating system functions accessing the scheduled demands are used.

C. Deploy Implementation

The instrumented implementation and mock-ups must then be deployed in the target hardware and software environment. The environment must match the environment specified in the validity statements to allow a meaningful result of the certification. If some properties of an environment are not important for a specification and are hence not stated in the hardware environment, the evaluator can choose an appropriate environment. An example is if a component does not issue requests for a hard disk it is not relevant which hard disk exists in the execution environment.

D. Run Testcases

The testcases derived in the first step are run on the deployed implementation and measurements of the necessary resource demands are gathered. A test case is run until the requested sample size is reached, which can be either specified directly or being calculated by a confidence level. The overhead for storing the measurements themselves should be as small as possible to prevent unwanted side effects on the performance of the implementation. The measurements from all runs of the test cases are denoted as performance results.

The evaluator uses the performance results to reason about the validity of the supplied specification. He uses statistical analysis to check if the measured values are considered valid with respect to the validity statements. If this is the case then a certificate is issued referencing the tested implementation, validity statements and component specification. If the specification is not valid then the testcase(s) leading to the rejection of the validity are given to the component creator.

The presented approach currently has the following limitations and assumptions:

- The approach currently considers only the validation of resource demands for processors. Hard disk or network requests are not validated.
- It focuses on the performance needs of business information systems and does not consider guarantees, for example by a schedulability analysis, which are important in many embedded systems. The methodology is inteded to compare experienced execution times and not to reason about best or worst case execution times.
- Validation is currently limited to basic components. The specifics of composite components specifications, e.g. how wiring the components is implemented, are not considered at the moment. However, if composite components are specified as basic components the approach is applicable.
- Resource demand specifications are currently only validated for fixed resource demands in the specification and a maximal deviance in percent.
- Checks that the implementation does not contain more dependencies than specified are currently not implemented.
- Data dependencies of components are currently not validated yet. However, the support for parameters of the used performance specifications allows taking these into account later on.
- Plain java objects are considered. Code weaving which is for example used in Java Enterprise Edition application servers is not considered yet.

IV. Component Performance Specifications

In this chapter, links to existing prediction approaches and their component performance specifications are given. We show the selection criteria for the specification which promises the highest pay off for certification and introduce the selected specification.

Most of the existing performance prediction and assessment approaches provide own performance specifications for components. These specifications describe the performance-relevant behavior of individual components and are used by the prediction approach to reason about the behavior of an assembled system. A general survey on model-based performance prediction approaches was created by Balsamo et al. [2]. They point out and compare which information is required by each of the approaches as well as on which software life cycle phase they focus, the degree of automation, and tool support. Another view is provided by Becker et al. in [3]. They focus their survey on performance prediction of component-based systems from an engineering perspective. They review existing approaches and tools and indentify their respective strengths and weaknesses in supporting different aspects in the design and development of component-based systems.

In general, performance specifications depend on the methodology used in the approach to predict the performance as well as which influencing factors on software components are taken into account. These influencing factors are stated for example by Koziolek in [4, p.42]:

- Implemented Algorithms
- Service Parameters and Internal State
- Performance of Required Services
- Resource Contention
- Deployment Platform

We reviewed the different existing prediction approaches and specifications in order to identify the one with the best support of accounting for the influencing factors and which is suitable for complex component-based systems. These requirements were selected because a better consideration of these factors fosters reusing a specification in different context. A higher degree of reuse promises higher payoffs for certification effort. Finally, the performance specifications of the Palladio Component Model (PCM) [1] were selected. PCM specifications allow taking implemented algorithms, service parameters, dependencies to required services, resource contention and dependencies to the deployment platform into account. However, the prediction of resource contention effects is focused on the resource processor. Other resources, e.g. memory or hard disk access, are only considered on a basic level.

In PCM, the behavior of services provided by a component is described by so-called Resource Demanding Service EFFect (RDSEFF) specifications. They describe the control and data flow of a component. The description is as abstract as possible while still allowing accounting for the influencing factors listed in the last paragraph. The elements of a RDSEFF and their relation to the influencing factors are explained in the following paragraph. More detailed descriptions of RDSEFFs are available in [5], [6], and [1].

The behavior modeled in a RDSEFF consists of required service calls, branches, loops, resource demands, resource acquisitions, resource releases, and forked behaviors. Of course, all of these reflect the implemented algorithm or, if still in the design stage, the estimation of the performance of an implemented algorithm. All parameters of the elements, e.g. number of loops or resource demand, can be specified as abitrary distribution functions and/or contain service parameters which influence the demand.

a) Required service calls: allow accounting for the performance of required service by making the point of a service call explicit and allow weaving in specifications for the behavior of the required service. These calls can be parameterized to take service parameters into account.

b) Branches and loops: allow accounting for service parameters influencing the performance of the component

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

  requiredService1();

  // internal computation
  innerMethod();

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

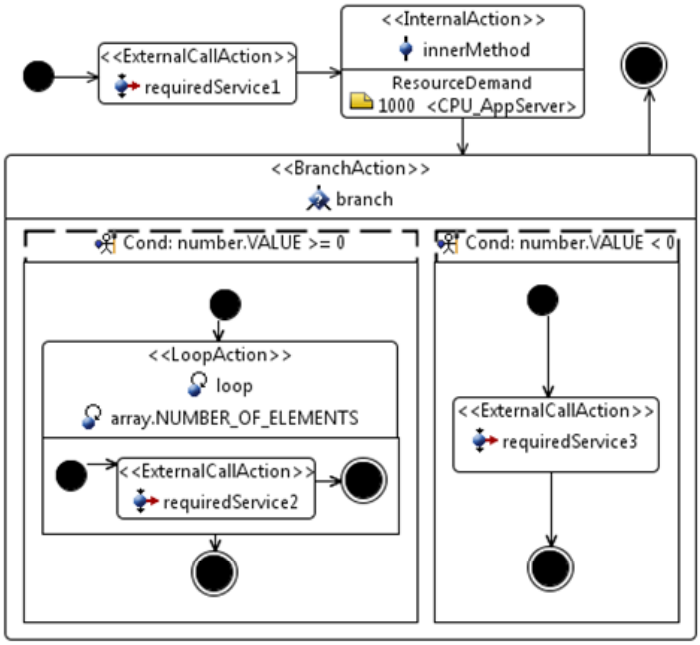


Figure 2. Example: Source Code and corresponding RDSEFF [1]

itself or the effect of control flow on the order of calls to required services.

c) Resource demands: allow to account for the effect of resource contention due to the demands issued by components. The demand in abstract units of a component on resources is specified. If the deployment of the component is known these units can be converted and the time required completing the demand computed.

d) Resource acquisition and release: allows accounting for the internal state of components and enables synchronization.

e) Forked behavior: allows accounting for resource contention effects even within a component by enabling to express concurrent execution of behavior within a component.

Figure 2 provides a simple example of the structure and information contained in a RDSEFF. In the example, the abstract performance-relevant behavior of the service *execute* of a component is described. Algorithmic complexity and substantial source code is hidden in the method calls to allow easier presentation. First, a call to a required service is issued. After its completion a component internal calculation is made, encapsulated in the *innerMethod*. The execution of this calculation requires 1000 units on the processor. Afterwards, the control flow splits depending on the number of elements in the service parameter *array*. Either the required service *requiredService2* is executed as many times as *array* has elements, or the required service *requiredService3* is executed.

V. Example

In this chapter, a simple example demonstrates the validation of a performance specification.

The validated component is labeled *ComponentUnderTest*. It requires the interface *ProcessingRequest* which provides the parameterless service *process*. It requires a component implementing the interface *IHelperService* which provides the parameterless service *calculate*. The RDSEFF of *ProcessingRequest* for *process* first request 500 units of CPU demand for an internal calculation (*innerMethod*). Afterwards the required service is executed and another internal calculation requests 250 units of CPU demand (*dataProcessing*). This is depicted in figure 3.

Two implementations are generated using the performance prototype approach of Becker et. al. [7]. Both implementations are based on the specification, but the second one should request 270 units of CPU demand at *dataProcessing*.

The validity statements are as follows. The hardware environment consists of a Intel Core 2 processor T5600, Intel chipset, and 2 GB RAM. One CPU unit in the specification equals 0.0001 ms on that machine. The software environment consists of Windows XP Professional SP3, a SUN Java VM 1.6.0_13 with HotSpot Client VM (build 11.3-b02, mixed mode, sharing). The example does not use input parameters, so none are listed. The resource demand precision for all resource demands uses intervals as validation estimator. 95% of the experienced resource demands of the internal action *innerMethod* should lie in the interval 50ms $\pm$6ms, and within 25ms $\pm$3ms for the internal action *dataProcessing*.

In the generate testcases phase, a sample size of 10000 is selected to be sure that a validation returns a trustworthy result. As there are no parameters in the specification there is just one test requesting the service *process*. In the example, the validation is limited to the forward direction. A mock-up for

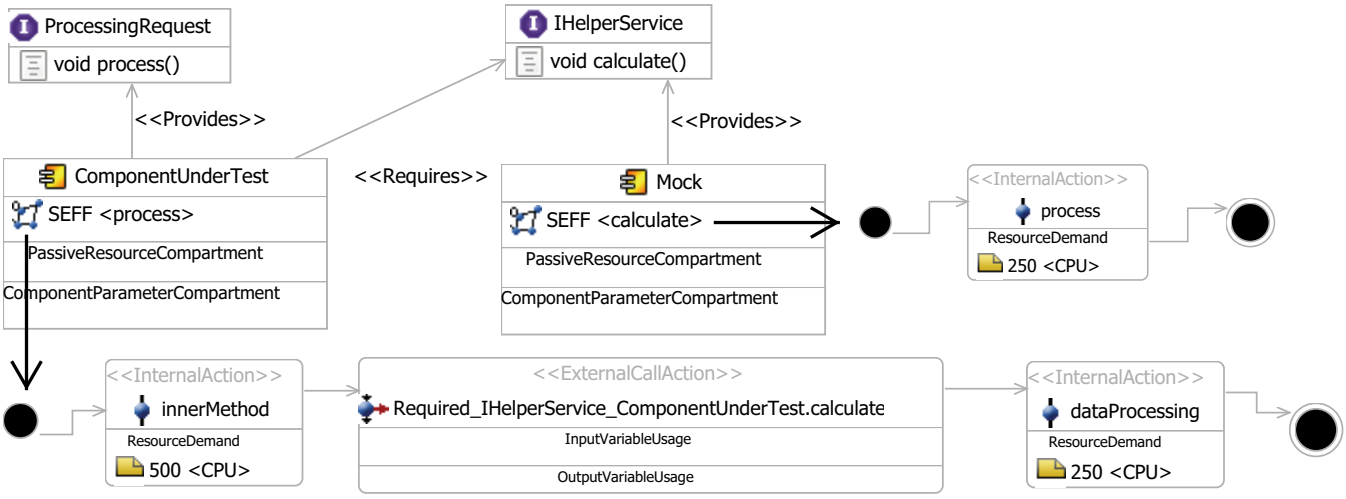


Figure 3. Example components and resource demands

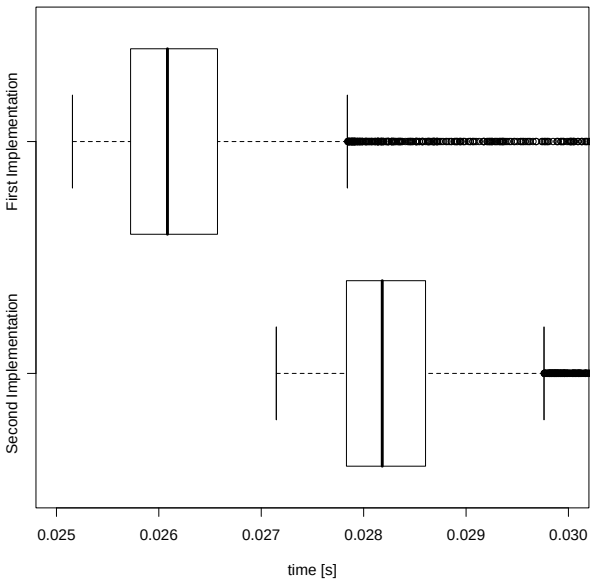


Figure 4. Boxplots for experienced resource demands

the required service is created manually. To demonstrate that any internal processing, for example to calculate any returned values, does not negatively influence validation an internal demand of 250 CPU units is issued (see also figure 3). In the instrument implementation phase, the code is instrumented manually. Passed time is measured using the wall-clock time returned by the Java operation `System.nanoTime()`. The implementation is then deployed manually in the validation environment. In the run testcases phase the test are run automatically until the sample size is reached.

The results for *dataProcessing* are depicted in figure 4. For implementation 1 and *dataProcessing*, 96.26% lie within the interval. For implementation 2 and *dataProcessing*, 37.63% which would lead to the decision that the specification is invalid for this implementation. For *innerMethod*, the values are 97.47% and 97.36% respectively. Overall, implementation 1 is regarded as valid.

The example is limited to the call of required services and processing of internal actions. The other constructs are not supported by this early lab prototype because they depend on validating parameterized specifications and considered as future work. The value of 95% of all measurements that should lie within an interval was chosen as the measurement method used wall-clock time and the used validation system had many low-load background jobs running which influenced the measurements.

VI. Related Work

Related work can be split into three different areas: Certifications, Performance Testing, and Performance Specification Validation. Each area is presented in a separate subchapter.

A. Certification

The research on component certification started in the early 90s and is still ongoing as shown in Alvaro et al.'s survey [8]. The survey shows the history of component certification and that certification approaches developed in the 90s focus on statements about the reliability of software components using test cases or mathematically analyzable models. Starting around 2000, the focus of the approaches shifted towards the certification of extra-functional aspects in general and the prediction of systems built out of certified components, e.g. [9], [10].

Wallnau also did some basic work on classifying certification approaches, for example the 10 useful distinctions for certification approaches in [11]. According to this classification the approach presented in this paper aims at reducing the gap between the knowledge about what a component does to what it actually does. It supports a *descriptive* certification of *objective* measures. The software *products* will be examined *empirically* with a given *context* in a *procedural* manner.

Meyer introduced in [12] a component quality model which distinguishes certification approaches between a "low road"

and a “high road". The *low road* summarizes the validation of existing component behavior and the *high road* means verification of component behavior with fully proven correctness properties. The approach described in this paper is based on statistical and empirical testing and hence belongs to the former category.

Hissam, Moreno, Wallnau et al. introduced the concept of predictable assemblies in [10] and later extended it to the *Predictable Assembly from Certifiable Components (PACC)* approach, described in [13]. PACC allows the prediction of the runtime behavior of a software system from the properties of its components and their patterns of interactions. PACC's performance reasoning framework currently focuses on fixed-priority preemptive scheduling, making it suitable to analyze hard real-time systems [14]. In contrast to the approach proposed in this paper, the authors concentrate on the support for hard real-time systems instead of business information systems. Additionally, they focus on the small areas and conditions in which some correctness properties can be proven.

The Cleanroom Software Engineering approach of Mills et al. [15] and the corresponding process of [16] target reliability of software systems. The approach and process aim to make development more manageable and predictable by using statistical quality control. The philosophy behind cleanroom software engineering is to avoid dependences on costly defect-removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system. Cleanroom software engineering yields software that is correct by mathematically sound design and software that is certified by statistically-valid testing. In contrast to the approach presented in this paper, the Cleanroom Software Engineering approach belongs to the high road approaches. However, statistical quality control is also used in the approach presented in this paper.

Alvaro et al. show in [17] the need of component certification within component-based software development for business information systems and discuss similarities and interdependencies between component selection and certification. The authors also provide a framework for component selection [18] as well as an selection process [19]. The approach proposed in this paper focuses solely on the extra-functional aspect performance and could later on be integrated in the more general framework of Alvaro et al..

Bøegh describes in [20] a formalized approach to state component properties and ensure trust in them by third-party certification whilst considering a multi-certification-standards scenario. The approach presented in this paper can be used as measure to ensure trust in quality claims for the performance of components of business information software.

In contrast to Bøegh's third-party approach, Morris proposes in [21] an approach for self-certification. It is designed to ease certification of functional aspects for open-source or free software. He developed a generic model to express test data. Published instances of these model and the software itself allow to verify quality claims by interested parties. The approach presented in this paper differs as it focuses on the extra-functional property performance instead of functionality and there is no need to publicize the software itself, nor does it depend on a preselected set of test cases.

B. Performance Testing

The performance testing market has been growing steadily [22] and hence there is a number of commercial and non-commercial performance testing tools available. The tools are presented in the following paragraphs. In contrast to the approach presented in this paper, the test cases run by the shown tools must be specified manually. In the approach of this paper, the information in the specification and implementation is used to deduce testcases for validation.

1) HP LoadRunner software: This commercial software is part of the Performance Center from HP and generates load, measures the performance, and helps to identify problems within a system. A more detailed overview is provided at [23]. It is designed to stress test an application from end-to-end and point out scalability issues. It provides support for diagnostic probes at code-level, non-intrusive real-time monitoring at system-level, and the inspection of SQL statements.

The approach presented in this paper uses probes on the code-level to measure the runtime of component internal processing sections as exact as possible. Hence, specialized probes have to be used for measurements.

2) LISA: The commercial LISA suite from iTKO is available at [24]. It consists of three tools: LISA Test [25] for designing and executing tests at UI-level and below, LISA Validate [26] for functional and performance monitoring, and LISA Virtualize [27] for behavior simulation of dependent services. The suite can run stand-alone as well as integrated with JUnit. The advantages of such a combined approach to end-to-end functional, load, and production testing are pointed out in [28]. LISA Virtualize provides the concept of a virtualized service as described in [29]:

> Service virtualization involves the imaging of software service behavior and the modeling of a virtual service to stand in for the actual service during development and testing. With a virtual service, you image the behavior of a particular service, you construct the virtual service from that behavior, and then you deploy it to a virtual service environment. [29]

The approach presented in this paper is related to LISA Test and LISA Validate. It is related to the former as it needs to automatically generate and execute tests on implementations. The later one is interesting as the behavior of required components or services has to be simulated in order to check the specifications.

3) JMeter: Apache JMeter is part of the Apache Jakarta Project and available at [30]. It is a java-based tool designed to load test functional behavior and measure performance for various server types. Supported server types are Web, SOAP,

JDBC, LDAP, JMS, and Email. It supports caching and offline analysis/replaying of test results.

Its testing strengths lie in heavy concurrent load conditions. The approach presented in this paper focuses on specification validation in single-user cases as it can rely on PCM's validated prediction approach to scale correctly in high concurrency situations.

4) OpenSTA: The Open System Testing Architecture (OpenSTA) is available at [31]. The current toolset has the capability of performing scripted HTTP and HTTPS load tests with performance measurements from Win32 platforms. Testing is performed using the record and replay metaphor common in most other similar and commercially available toolsets. Data collections include scripted timers, SNMP data, Windows Performance Monitor stats, and HTTP results & timings.

The data collection and capturing possibilities are of interest for the approach presented in this paper. However, capture and replay methods are of minor interest, as the specification validation should not depend on this kind of functional test data.

5) PushToTest: PushToTest is a commercial open source alternative for testing and monitoring and available at [32]. It provides capabilities for functional testing, load testing, and monitoring.

The approach presented in this paper could use integrated open-source tools, like Glassbox for monitoring implementations.

C. Performance Specification Validation

Pavlopoulou and Young examined residual test coverage in [33]. The program statements not covered by previous testing approaches are instrumented to see if they are actually used or if the assumption that they are seldom used in practice holds.

The approach presented in this paper can use statement coverage to determine areas which might influence performance but have not been measured before.

The effort of testing and reusing components was addressed by Weyuker in [34]. She states that high reliability and availability requirements lead to enormous costs. Additionally, components have to be tested in isolation and after integration so savings of components-of-the-shelf are not sure. Reusing components requires retesting for stability, reliability, stress, and performance testing.

The approach presented in this paper will allow testing components for specified ranges of parameters and environments. As long as the components are used within these boundaries predicting the expected behavior wrt performance should require only very low effort.

Extra-functional behavior and component testing is also considered in [35] by Hamlet. The article is about a compositional testing theory based on subdomain or partition testing. Component test points and their (input and output) propagation are considered to identify the best test criteria or cases. Focus is put on functional behavior but extra-functional behavior is covered as well. Models are used to abstract the data flow and allow deriving test cases. Resulting from the type of modeling data flow, the approach has difficulties for example in finding fix points for loops / iterations. Additionally, the article points out that theoretical comparisons between random and subdomain testing have not shown a conclusive advantage either way in detecting failures.

In contrast to the approach presented in this paper, Hamlet focuses on functional testing which is also required for the kind of extra-functional testing described in his article.

VII. Conclusion

In this paper, we showed links to existing performance prediction approaches and their performance specifications. We gave the reason why the RDSEFF specification was chosen for this approach and introduced the specification itself. Additionally, we identified and explained which information is necessary to validate performance specifications. We presented our approach to certify specifications against implementations and explained the process of assessment and certification as well as listed current limitations and assumptions. A small lab example demonstrated the applicability of the process.

The presented approach aids companies in offering components in marketplaces. The publication of certified performance specifications in a marketplace is sufficient for potential customers to reliably evaluate and select components. However, the interests and intellectual properties of the offering companies are still protected as the specifications only contain a highly abstract view on the component's behavior and keep the disclosure of details on the used algorithms and techniques to a minimum. Having certified performance specifications additionally supports software engineers in late composition of components. The software engineers gain the knowledge if the performance specifications fulfill their requirements for the intended composition which can in turn ease the evaluation of components and reduce the necessary effort. The certification of specifications also supports a predictable assembly of components. The information contained in these specifications allows increasing confidence in the results produced by validated performance prediction approaches or identifying potential risks. Last but not least, certification by independent authorities provides a mean for quality assurance. If the development of components is given to a contractor the stipulation of certification enables the contracting body to trust the performance of a developed component beyond a few test cases while keeping its own quality assurance effort low. The contracting body can focus on its own expertise and does not need to employ performance engineers just for quality assurance.

As a next step the design of a model describing the validity of performance specifications is planned. It is also planned to extend the approach to allow the comparison and validation of parameterized distribution functions including predictions on the necessary effort in terms of test cases and runs for the requested validation. Furthermore, support for validating dependencies to input parameters of a component's service is planned as well. In the medium term validating the specified

against the implemented required interface and the specified and implemented external call sequence is planned. In the long term the validation should also include return value dependencies from external calls.

ACKNOWLEDGEMENTS

Support for this work has been provided by the German Federal Ministry of Education and Research (BMBF), grant No. 01BS0822. The author is thankful for this support.

REFERENCES

[1] 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

[2] 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.

[3] S. Becker, L. Grunske, R. Mirandola, and S. Overhage, "Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective," in *Architecting Systems with Trustworthy Components*, ser. LNCS, R. H. Reussner, J. Stafford, and C. Szyperski, Eds. Springer, 2006, vol. 3938, pp. 169–192.

[4] 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

[5] 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, 2007, october 2007. [Online]. Available: http://sdqweb.ipd.uka.de/publications/pdfs/reussner2007a.pdf

[6] K. Krogmann, "Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions," in *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*, ser. Interne Berichte, R. H. Reussner, C. Szyperski, and W. Weck, Eds., vol. 2007-13. Karlsruhe, Germany: Universität Karlsruhe (TH), July 31 2007, pp. 23–29. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007172

[7] S. Becker, T. Dencker, and J. Happe, "Model-Driven Generation of Performance Prototypes," in *Performance Evaluation: Metrics, Models and Benchmarks (SIPEW 2008)*, vol. 5119, 2008, pp. 79–98. [Online]. Available: http://www.springerlink.com/content/62t1277642tt8676/fulltext.pdf

[8] A. Alvaro, E. de Almeida, and S. de Lemos Meira, "Software component certification: a survey," *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 106–113, September 2005.

[9] J. Stafford and K. Wallnau, "Is third party certification necessary?" in *4th ICSE Workshop on Component-Based Software Engineering*, 2001.

[10] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau, "Enabling predictable assembly," *J. Syst. Softw.*, vol. 65, no. 3, pp. 185–198, 2003.

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

[12] B. Meyer, "The grand challenge of trusted components," *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 660–667, May 2003.

[13] K. C. Wallnau, "Volume iii: A technology for predictable assembly from certifiable components (pacc)," SEI, CMU, Tech. Rep. CMU/SEI-2003-TR-009, 2003.

[14] G. A. Moreno and P. Merson, "Model-Driven Performance Analysis," in *proceedings of the Fourth International Conference on the Quality of Software Architectures (QoSA 2008).*, ser. LNCS, S. Becker, F. Plasil, and R. Reussner, Eds., vol. 5281. Springer, September 2008, pp. 135–152.

[15] H. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, September 1987.

[16] R. Linger, "Cleanroom process model," *IEEE Software*, vol. 11, no. 2, pp. 50–58, March 1994.

[17] A. Alvaro, R. Land, and I. Crnkovic, "Software component evaluation: A theoretical study on component selection and certification," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-217/2007-1-SE, November 2007. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1371

[18] A. Alvaro, E. S. de Almeida, and S. L. Meira, "Towards a software component certification framework," *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pp. 298–303, October 2007.

[19] ——, "Component quality assurance: Towards a software component certification process," *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on*, pp. 134–139, August 2007.

[20] J. Boegh, "Certifying software component attributes," *Software, IEEE*, vol. 23, no. 3, pp. 74–81, June 2006.

[21] J. Morris, G. Lee, K. Parker, G. Bundell, and C. P. Lam, "Software component certification," *Computer*, vol. 34, no. 9, pp. 30–36, September 2001.

[22] G. Hamilton, "Application complexity spurs growth in performance validation market," Yankee Group Research, Inc., Tech. Rep. 4AA1-7656ENW, December 2007.

[23] L. P., "HP LoadRunner Software Data sheet," Hewlett-Packard Development Company, Tech. Rep. 4AA1-2118ENW, November 2008.

[24] "itko lisa website," http://www.itko.com/lisa.

[25] J. English, "itko lisa™ test ensuring both end-user and system-wide quality for enterprise applications," Interactive TKO, Inc. (iTKO), Tech. Rep., November 2008. [Online]. Available: http://www.itko.com/site/resources/iTKO_LISAtest_PS_Nov08.pdf

[26] ——, "itko lisa™ validate the "always on" build time, runtime, and change time quality governance platform," Interactive TKO, Inc. (iTKO), Tech. Rep., November 2008. [Online]. Available: http://www.itko.com/site/resources/iTKO_LISAvalidate_PS_Nov08.pdf

[27] ——, "itko lisa™ virtualize eliminate the costs and limitations of constrained services for quality and agility," Interactive TKO, Inc. (iTKO), Tech. Rep., November 2008. [Online]. Available: http://www.itko.com/site/resources/iTKO_LISAvirtualize_PS_Nov08.pdf

[28] J. Michelsen, "Merging open source and testing strategies. junit and lisa: The perfect match," Interactive TKO, Inc. (iTKO), Tech. Rep., December 2007. [Online]. Available: http://www.itko.com/site/resources/open_source.jsp

[29] ——, "Service virtualization in enterprise application development," Interactive TKO, Inc. (iTKO), Tech. Rep., January 2009. [Online]. Available: http://www.itko.com/site/resources/iTKO_WP_VirtualServicesGuide_Jan062009.pdf

[30] Apache, "Jmeter website," http://jakarta.apache.org/jmeter/.

[31] "Opensta website," http://www.opensta.org/.

[32] "Pushtotest website," http://www.pushtotest.com/.

[33] C. Pavlopoulou and M. Young, "Residual test coverage monitoring," *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pp. 277–284, 1999.

[34] E. J. Weyuker, "Testing component-based software: A cautionary tale," *Software, IEEE*, vol. 15, no. 5, pp. 54–59, Sep/Oct 1998.

[35] D. Hamlet, "Software component composition: a subdomain-based testing foundation," *Software Testing, Verification and Reliability*, vol. 17, no. 4, pp. 243–269, 2007.

Using UML for Domain-Specific Component Models

Ana Petričić, Luka Lednicki
Department of Control and Computer Engineering in Automation
Faculty of Electrical Engineering and Computing
Zagreb, Croatia
{ana.petricic, luka lednicki}@fer.hr

Ivica Crnković
Mälardalen Research and Technology Centre
Mälardalen University
Västerås, Sweden
{ivica.crnkovic}@mdh.se

Abstract – **Over the recent years there is a tendency for using domain-specific languages which enable expressing design solutions in the idiom and level of abstraction appropriate for a specific problem domain. While this approach enables an efficient and accurate design, it suffers from problems of standardization, portability and transformation between the models. This paper addresses a challenge of tailoring UML, a widely used modelling language, for domain-specific modelling. We discuss a possible solution for achieving interoperability between UML and the domain-specific language SaveComp Component Model (SaveCCM) intended for real-time embedded systems, by means of implementing a transformation between UML and SaveCCM models. The challenge of the transformation is to keep all necessary information including the domain specific semantics. The paper presents the strategy for the transformation, its implementation and an analysis. We also address the second challenge, a usability of the domain-specific language (i.e. SaveCCM) in comparison with usability of extended UML and by an experiment analyse its usability in comparison with SaveCCM.**

Keywords–**Software component models, model transformation, UML, UML profile, domain-specific languages, modelling tools**

I. INTRODUCTION

A number of Domain Specific Languages (DSL) exists nowadays which provide more expressiveness at the design time and efficiency in analysis and testing. One of such DSLs is the SaveComp Component Model (SaveCCM) [1][2], intended for building embedded control applications in vehicular systems. SaveCCM is a research component model in which design flexibility is limited to facilitate analysis of real-time characteristics and dependability. As a domain specific language, SaveCCM is productive for designing safety-critical systems responsible for controlling the vehicle dynamics.

A disadvantage of DSL is paradoxically, its specificity – it may require additional efforts to be used, it can cause obstacles in communication of design decisions between different stakeholders, and it requires development of custom design tools. Contrary to most DSLs, the Unified Modeling Language (UML) [3] as a de facto standard in industry has a wide spread base of trained users and a number of modelling tools. Therefore, providing a way for domain-specific modelling using UML could prove very beneficial. By combining UML and SaveCCM, we could take advantages of both languages in different aspects they provide, and in different stages of system development process. Using of UML for domain specific modelling can reduce time and cost of building specific modelling tools, and can bring the feature of portability and standardization to a system model. However there can be a challenge to a) to express a DSL by UML and b) implement a transformation between them.

In this paper we set up two questions. The first one is the feasibility of combining general-purpose and domain specific languages in terms of full and unique transformation of models in both directions. The second question is the usability of our approach, compared to using only standard, domain specific modelling in order to perceive if there is any need for building specialized tools instead of using general purpose ones with appropriate extensions. These two questions we apply on SaveCCM and UML. The specificity of the case is the domain, namely real-time and embedded systems, which requires quite different modelling, due to specific interaction styles and specific concerns, such as real-time properties and resource constraints. To obtain the answer to the first question we provide a simple solution for achieving interoperability between SaveCCM and UML through a formal way of representing SaveCCM models using UML 2.0 component diagram and defining a transformation between the two model formats. Usability is discussed over the results of an empirical evaluation that we carried out to test the efficiency and user-friendliness in using extended UML compared to using only a domain-specific language and specialized tools.

The rest of the paper is organized as follows: After describing our motivation in section 1, we bring a brief description of SaveCCM in section 2. In sections 3 and 4 we present our UML profile and the design of transformation between UML and SaveCCM. Section 5 discusses applicability of our approach and presents results of an empirical evaluation that we have conducted. Finally,

section 6 presents related work and our concluding marks are given in section 7.

II. THE SAVECCM OVERVIEW

SaveCCM is a component model intended for designing safety-critical resource constrained systems responsible for controlling the vehicle dynamics. SaveCCM technology provides a support for designing systems and analysis of their timing properties built in an integrated development environment named SaveIDE.

The main architectural elements in SaveCCM are:

- *Components*, which are the basic units of encapsulated behaviour with a functionality that is usually implemented by a single function written in C programming language. Besides the C function, each component is defined by associated ports and optionally quality attributes.
- *Switches*, which provide facilities to dynamically change the component interconnection structure (at configuration or run-time); thus allowing a conditional transfer of data or triggering between components.
- *Assemblies*, which provide means to form aggregate components from sets of interconnected components and switches.

SaveCCM also provides a hierarchical component composition mechanism in a form of a special type of a component – *composite component*, where the functionality of a component is specified by an internal composition instead of using a C function.

An important characteristic of SaveCCM is the distinction between data transfer and control flow, which is achieved by distinguishing two kinds of ports; *data ports,* where data of a given type can be written and read, and *trigger ports* that control the activation of components. The separation of data and control flow allows a model to support both periodic and event-driven activities. In addition to ports, the interface of a component can contain quality attributes, having each attribute associated with a value or a model-based specification and possibly a confidence measure. These attributes can hold the information about the worst case execution time, reliability estimates, safety models, etc.

An example of a simple temperature regulation system modelled using SaveCCM (in SaveIDE tool) is shown on Figure 1. It consists of two SaveCCM components, one assembly and one composite component. On the figure are also visible various types of SaveCCM ports: input and output trigger ports (triangle shape), input and output data ports (square shape), and input and output combined ports (combined triangle and square shape).

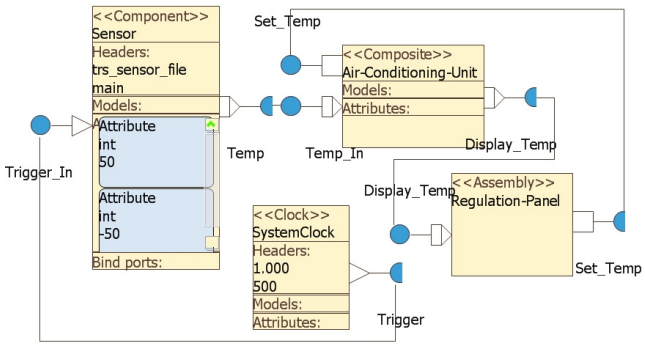


Figure 1. SaveCCM model of Temperature regulation system

More information on SaveCCM with a detail description of model elements and their attributes, as well as an overview of the SaveCCM execution model can be found in SaveCCM reference manual [2] and [1][9].

Apart from unique modelling elements, such as switches or assemblies that provide specific behaviour, as well as clock and delay components, SaveCCM introduces several valuable concepts that can not be found in UML.

- The distinction between data transfer and control flow.
- Concept of component interface. In SaveCCM the functional interface of every modelling element is defined by a set of ports associated to the element and optionally, quality attributes.
- Model analysis and verification. SaveCCM uses quality attributes for defining non-functional properties of components and systems which allow analysis of various properties and system verification.

Execution semantics of active model elements, defined to provide run-time model analysability. The execution model is rather restrictive, its basis is the pipes and filter control-flow paradigm in which component execution is defined by a sequence of activities: start by trigger, read, execute, and write.

III. THE SAVEUML PROFILE – A UML SPECIFICATION OF SAVECCM

A common way of specialising UML to align it with important design issues in different domains is to define a UML profile suitable for the domain. We will use the UML profiling mechanisms to tailor UML for SaveCCM domain in a controlled way. By defining a profile we generate an extension to UML consisting of elements with different semantics. However we also must limit the use of standard UML elements to a subset that fits our target domain. Our UML profile, named *SaveUML profile*, defines a one-to-one mapping to elements defined by SaveCCM. We have identified the UML 2.0 subset that addresses the concepts used in component-based development as well as the ones existing in SaveCCM. It includes the UML 2.0 *Components* and *Composite Structures* packages. We call this subset a *UML 2.0 component model.*

The UML profile we developed, is to provide an equivalent language to SaveCCM language. It aims at modelling systems in UML but using SaveCCM semantics, and supporting unambiguous transformation between the UML and SaveCCM models.

Considering that UML profiles are a standard UML extension mechanism and are therefore a part of UML, they are as widely recognized as UML itself and should be supported by all standard modelling CASE (Computer-Aided Software Engineering) tools. This possibility of customizing UML for specific domain purposes while remaining within boundaries of the UML standard and keeping the possibility of using UML CASE tools, presents a reasonable motivation for customizing and using UML instead of a specific modelling language.

The process of defining the SaveUML profile consisted of three phases:

1. *Identification of SaveCCM and UML component model elements*. We have made a detail analysis of UML 2.0 component model (a subset of UML concerning UML components), which allowed us to survey the similarities between UML and SaveCCM and identify compatible elements. In addition we defined mapping rules for all SaveCCM elements that need to be translated to UML elements and corresponding UML elements that can be used for mapping.
2. *Identification of SaveCCM language constraints*. Designing SaveCCM elements with UML 2.0 elements brought up various problems resulting from a strict syntax of SaveCCM and the universality of UML. Therefore, we had to create a set of constraints to refine the UML 2.0 component model semantics to be suitable for designing SaveCCM models.
3. *Translation of previously identified elements,* during which a suitable UML element is found for every SaveCCM language elements. Chosen UML elements were then further customized through the use of necessary stereotypes, properties and constraints.

The diagram of the SaveUML profile is depicted in Figure 2. The SaveUML profile specifies a set of stereotypes which extend elements of the UML 2.0, namely UML `Component`, `Port`, `Property`, `Artifact`, `Usage` and `Dependency`. Each element from SaveCCM domain has its corresponding element in the SaveUML profile. For introducing the properties of SaveCCM elements (e.g. jitter and period attributes of SaveCCM clock component etc.) we used the tagged value mechanism. The SaveCCM semantics is imposed upon the UML model using Object

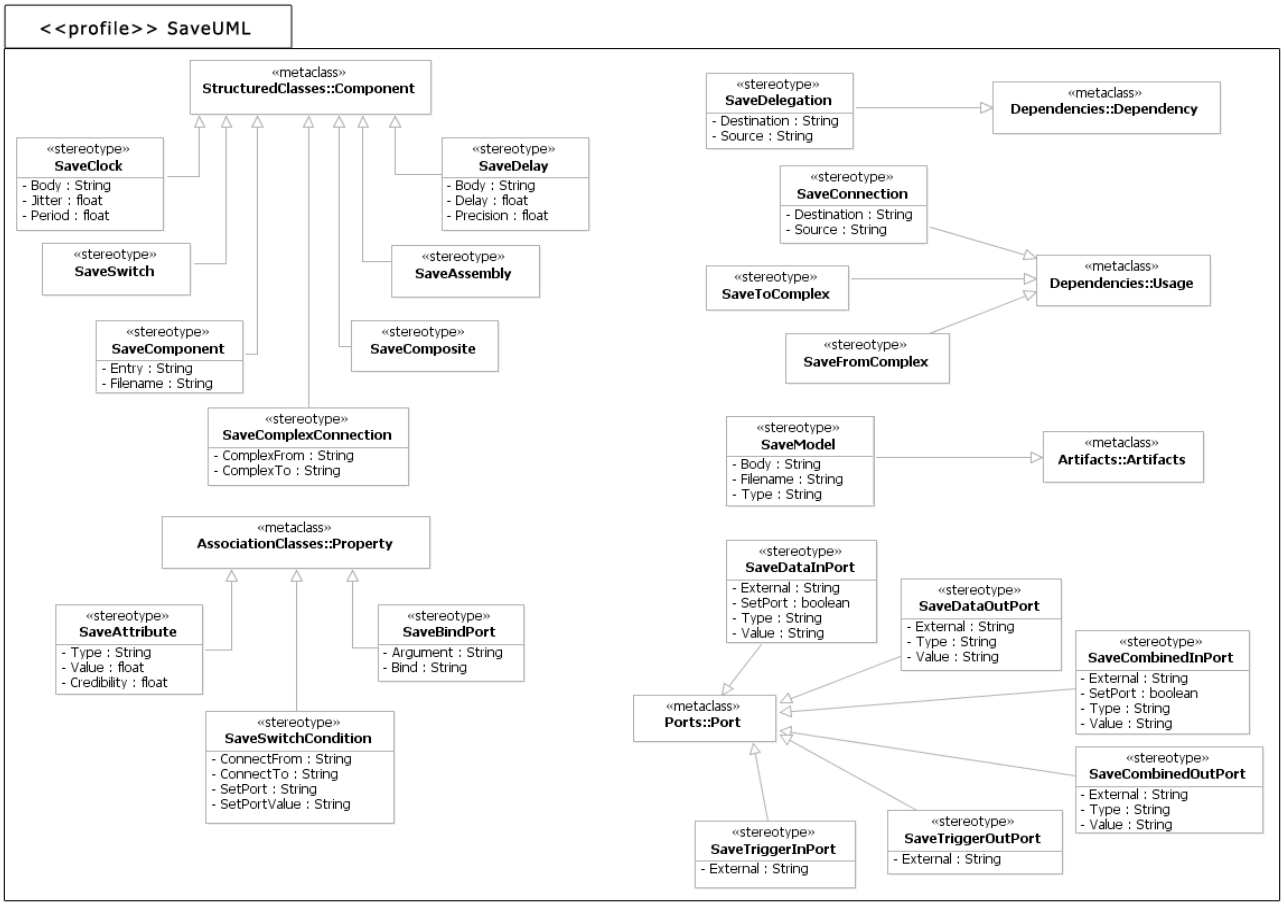


Figure 2. Diagram of the SaveUML profile

Constraint Language (OCL).
During the process of creating the SaveUML profile, we have made several design decisions considering representing of SaveCCM architectural elements within the profile, the method of defining substructure of components and different concepts of interfaces in SaveCCM and UML. These design decisions are presented below.

Components
Since SaveCCM introduces three main architectural elements (component, assembly and switch) and three subtypes of SaveCCM component (clock, delay and composite component) we had to define six new UML elements by using stereotypes that will extend the UML `Component` element. Similar concept was applied for defining different port types that exist in SaveCCM language.

*Subcomponent*s
SaveCCM offers two elements that may have an internal structure: assembly and composite component. In UML 2.0, we can specify internal sub-element of a component either as its property (by using the `Property` metaclass) or as a packaged element (using the `PackageableElement` metaclass). We chose the latter approach – using `PackageableElement`. Unlike the first method, usage of PackageableElement enables defining a hierarchical composition of components and its nested subcomponents to an arbitrary depth. Definition of an owning component also includes the definition of its subcomponents, leaving no need for referencing outside elements. Such a definition of subcomponents is also referred as an *embedded definition of components*.

Interfaces
In SaveCCM the functional interface of every modelling element is defined by a set of ports associated with the element. Because of semantic differences of interface in SaveCCM and UML, we decided not to use UML interfaces in SaveUML profile. It is supposed that when modelling a user model in UML using the SaveUML profile, the interface of a component will be determined implicitly by its ports, as it is done in SaveCCM.

A *Using OCL for user model validation*

In order enforce the SaveCCM semantics to the SaveUML profile we defined a number of constraints within the profile using the Object Constraint Language (OCL) [3]. We used OCL constraints to enforce the SaveCCM semantics and restrict the usage of UML concepts that do not have equivalent elements within SaveCCM.
We divided the implemented constraints into two main groups – *Restrictions on UML* and *SaveCCM semantics*. Each group has several sub-groups which are described in Table 1. In total we implemented 117 constraints.

We found that identifying and specifying OCL constraints is the major part in development of a UML profile and can be a challenging task for non-experienced UML user, as UML is a complex language with many elements and various diagram types.
An important part in using OCL is the tool support. We have chosen to use the Rational Software Modeler (RSM) [9] tool for implementing SaveUML technology. RSM is built on the extensible Eclipse framework and it fully supports the definition of UML profiles, which are consequently stored in XML files.

TABLE 1. CONSTRAINTS IMPLEMENTED IN SAVEUML PROFILE

Constraint group	Count
Restrictions on UML	**56**
Forbidden connections	***17***
Restrictions on UML 2.0 considering using various types of connectors. Also, connectors should not connect elements directly etc.	
Using interfaces	***12***
Using UML interfaces is not allowed within the SaveUML profile, these constraints are dealing with this issue.	
Substructure definition	***6***
Internal structure of an element may only be defined using packaged elements. Further, the only allowed packaged element is a Component.	
Number of stereotypes	***21***
Even though UML has the option to apply multiple stereotypes to one element, in SaveUML profile, one element can have only one stereotype applied.	
SaveCCM semantics	**61**
Owning attributes	***6***
These constraints are defining attributes that main SaveCCM elements may own.	
Owning ports	***13***
Since SaveCCM offers several kinds of ports, each port must have appropriate stereotype applied in order to determine its type. Further, some SaveCCM elements have restrictions on number of ports that they own.	
Bind port	***3***
These constraints introduce semantic rules considering special type of port – *bind port.*	
External ports	***6***
These constraints introduce semantic rules considering special type of port – *external port.*	
Switch semantics	***5***
Switch component is specific SaveCCM element. These constraints introduce its semantics. They deal with concept of *set port*, *switch condition* and *switch connection*.	
Connections between SaveCCM elements	***23***
Since SaveCCM offers two kinds of connections, each connector must have an appropriate stereotype applied. Also, depending on the connection type, cyclic connections are forbidden or allowed. Finally, these constraints ensure conformance of the connected ports (their types and directions).	

IV. SAVEUML TRANSFORMATIONS

The transformation approach is based on using the eXtensible Stylesheet Language for Transformations (XSLT) [11]. Recommended by the World Wide Web

Consortium (W3C), XSLT is a flexible language for transforming XML documents into various formats including HTML, XML, text, PDF etc. The input to XSLT transformations are XML Metadata Interchange (XMI) representations of models, which are based on XML syntax. XMI eases the problem of tool interoperability by providing a flexible and easily parsed information interchange format. In principle, a tool needs only to be able to save and load the data in XMI format.

The conceptual design of SaveUML transformations is depicted graphically in Figure 3.

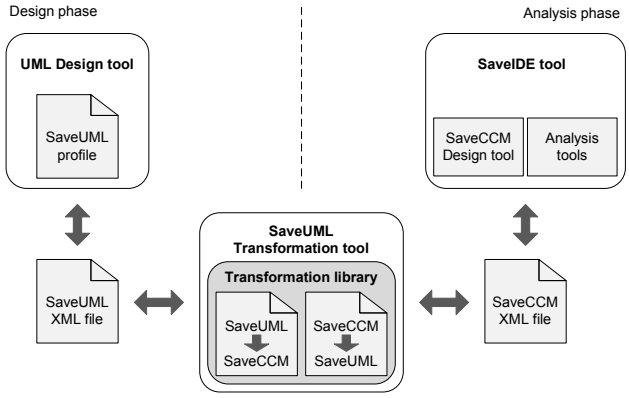


Figure 3. Conceptual design of the SaveUML transformations

The UML CASE tool is used for creating a UML model. Application of the SaveUML profile is necessary in order to create a UML model which can be transferred into a SaveCCM model. After designing the model, it is exported into an XMI file which is then used as the input to the transformation. The SaveCCM design tool is Save-IDE. For representing model information, SaveIDE uses several files which are compatible with XML and are used by the transformation tool to perform the SaveCCM to UML transformation. The tool uses the transformation library to perform translations. It contains XSLT style sheets for transforming from SaveUML into SaveCCM and vice versa. Input files based on XML are parsed through the XSL transformation style sheets and then XML-based output files, compatible with the desired tool, are generated.

A *Characteristics of SaveUML transformations*

Inspired by Visser's classification for program transformation [12] we classified SaveUML transformations as *language translation*, more precisely *migration*, as we transform between models specified in different languages at the same level of abstraction.

After the transformation, the source model and the target model do not stay untouched but coexist and may evolve independently due to the development process. Therefore, we implemented transformations in both directions, from UML to SaveCCM model and reverse, having in mind this request. Reverse transformation, i.e. transforming the model from one language to another and back to the starting language, should produce a model equivalent to the initial one. SaveUML profile already provides a one-to-one mapping from UML to SaveCCM. In addition, models are transformed at the same level of abstraction which makes these transformations injective. The transformation process itself comes to transforming from one XML representation of a model to another XML file. Therefore, the request for a unique transformation is fulfilled.

We implemented a prototype of the SaveUML transformation tool as a Java application. Transformation tool we developed can be used either as an Eclipse (i.e. RSM) plug-in or as a standalone application to perform transformation in both directions, UML to SaveCCM and SaveCCM to UML models.

V. AN EMPIRICAL COMPARISON OF SAVECCM AND SAVEUML TECHNOLOGIES

In previous sections we have described an approach for connecting two different modelling languages. A question arises on its usability in practical cases. Therefore we performed an experiment to verify the approach by comparing the modelling capabilities of both SaveIDE tool and RSM tool combined with SaveUML profile.

In this section we provide an overview of the experiment and its results, more detail can be found in [13].

A *Discussion and experiment objectives*

Even though there are advantages of combining UML and SaveCCM in different development stages, how many benefits comes from these advantages and can they overwhelm the existing disadvantages and problems? In cases when usability of an UML profile is satisfying, and the expressiveness of UML extensibility mechanisms is sufficient for particular domain, then the need for building specialized tools is questionable. In these cases, designing an UML profile and using some of existing UML tools could replace a custom built tool.

There are several significant advantages that support this approach:

- Using of already existing UML tools, which reduces time and cost spent on developing a domain-specific modelling environment.
- Any knowledge of and experience with standard UML is directly applicable.
- The UML profile is compatible with standard UML, thus any tool that supports UML can be used for manipulating models based on a UML profile. This brings the portability to models designed using SaveUML profile among many CASE tools.

However, the approach has some drawbacks:

- A UML profile as an upgrade to basic UML can lead to an overly complicated model within an already complex

UML specification and a modeller might get confused with extraneous UML semantics or modelling elements.

- Using standard UML notation, in which an existing shape corresponding a DSL (such as SaveCCM) element is used, could compromise the readability and clarity of the diagrams.
- A DSL is usually not used independently, but in combination with other tools, models and DSLs. In such a case the problem is not solved by expressing one DSL by UML extensions, but about a set of DSLs that should be mapped.

Our aim was to empirically evaluate our approach, with respect to development efficiency and ease of use. We wanted to compare the two modelling tools (SaveIDE and RSM along with SaveUML profile), to get feedback from the users and to ascertain advantages or disadvantages in using these tools.
With this experiment we tried to answer the following questions:

- Is using of SaveUML profile efficient with regard to time and efforts, in comparison to using SaveIDE?
- Do extraneous UML elements and semantics confuse developers and lead to an invalid or incomplete SaveCCM model?
- Which of technologies is more user-friendly and provide better user experience?

The given questions have more explorative character, so the results are shown mostly as descriptive statistics.

B Conduction of the experiment

In this experiment 18 software engineering master students were given the task to design a model of a real-time system using either RSM (with SaveUML profile) or SaveIDE tool. As one of criteria for measuring development efficiency, we were monitoring efforts spent and quality of designed model in terms of its validity and detailness. After the modelling was finished we analysed models delivered by students, and students were given a questionnaire regarding their experience of working with the given modelling tool. Almost all of the students had some experience with UML. We conducted the study in a form of minutely described assignments for students with strictly defined deliverable deadlines. Our experiment was not conducted under controlled conditions in laboratory, but it is executed in the field under normal conditions i.e. in a real development situation. Except for the varying factor we wanted to study, which was a modelling technology, we controlled the qualification of the testers and an input for testing i.e. an example of a real-time system.
The actual study consisted of three phases. First we trained students in concerned technologies, then we conducted the experiment, and finally we let students to fill in a questionnaire and we analysed the results.

Training phase

The training phase lasted for three weeks. First two weeks were reserved for studying SaveCCM and UML (precisely UML component diagram). After two weeks students were given an exam which tested their knowledge. Based on the results of this exam we separated students into two groups, one that will use SaveIDE tool, and one that will use RSM tool with SaveUML profile. The disposition was made in a way to have two homogeneous groups with a comparable qualification range. Third week of training phase was intended for getting familiar with the tools by for modelling a simple system example.

Modelling phase

For the experiment we prepared a specification of Autonomous Truck Navigation (ATN) system demonstrated in [5]. This autonomous system is intended to navigate the truck to find and follow a straight black line drawn on the surface area. A model of the system, designed in RSM tool using SaveUML profile is shown in Figure 4.

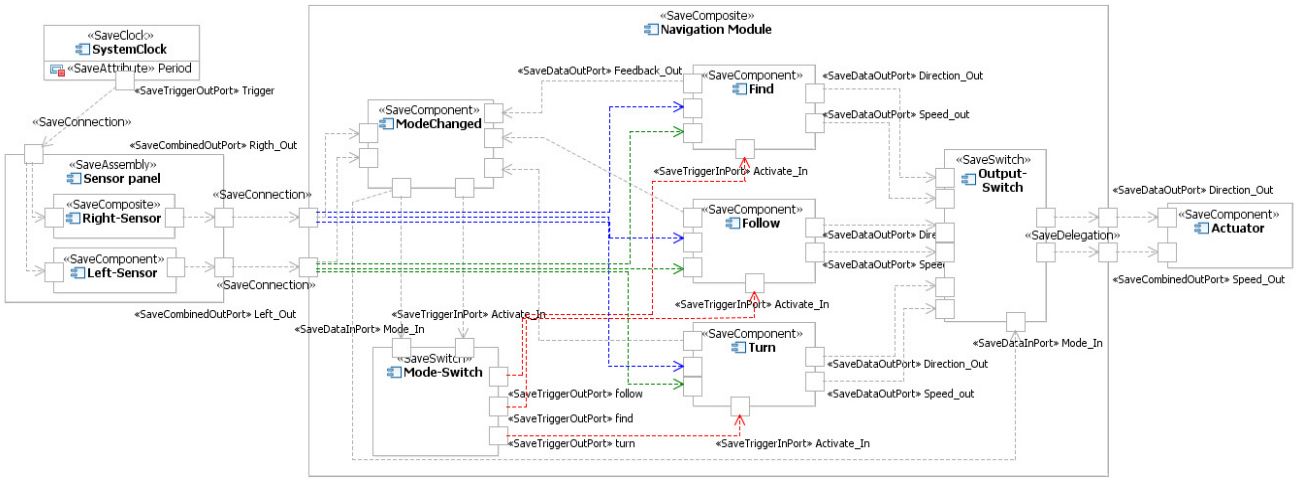


Figure 4. A simplified model of Autonomous truck navigation system

Questionnaire

After the modelling was finished, participants completed a questionnaire regarding their user experience in the given tool. This questionnaire covered a number of subjects such as initial effort participants had to make to learn the technology, complexity of using the tool and clearness of graphical representation of modelling elements. Also there were questions about problems and bugs that encountered during their work, and several questions about different aspects of using SaveUML profile.

C Results

By analysing the results of the experiment, our goal is answer the three questions raised above.

Which of technologies is more user-friendly and provide better user experience?

The initial effort participants had to make to get to know with the tool was slightly different for the two tools. As it can be seen in Figure 5, group using SaveUML had more problems when starting to use the tool. One of the possible reasons was that RSM is a more complex tool and offers a lot of possibilities which confused them. However, the SaveUML group also reported more improvement after having some training with the tool (Figure 6).

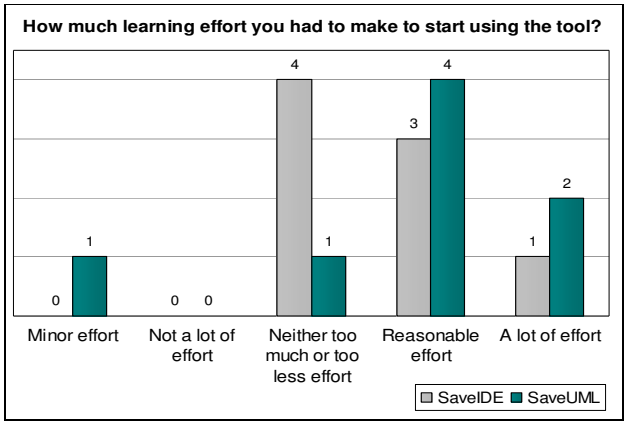


Figure 5. Initial effort for using the tool

Considering the complexity of using the tool, users of SaveUML found project management, defining properties of model elements and adding new elements to the model a bit more intuitive than the SaveIDE users. This is not very surprising for the first two as the RSM tool is more advanced and these actions are common to all UML modelling tools. But one would expect that a custom tool would have better usability when it comes to adding new (custom) elements than the usage of UML profile. From this we can conclude that probably some flaws (unintuitive steps) in the whole process of adding elements can have more negative impacts than the steps needed for applying a UML stereotype.

Workspace organisation and overall complexity of using the tool were graded very similar by both groups. SaveUML users also reported better assistance (automated procedures, offering default values etc.) from the tool, which is not surprising considering that RSM is a professional tool.

As SaveIDE uses all custom graphics and SaveUML only the default UML graphics, it was expected that the SaveIDE group would report much better readability of models than the SaveUML group. Although the SaveIDE group reported a better readability, the difference between the SaveIDE and SaveUML groups is not as large as expected, as it can be seen on Figure 7. The comments given by participants indicate that some flaws in the graphical representation in the custom tool have a big impact on experience of graphical environment.

Advantages of the professional RSM tool are clearly visible to the users. The RSM tool has been graded as much more stable, and work with it much more tolerable. The overall grade of using experience was better for SaveUML users, as it is presented in Table 2. We can explain this by two reasons: SaveIDE tool is a research tool, of a prototype level, while RSM is much mature professional tool. The second reason is the students' familiarity with the UML notation.

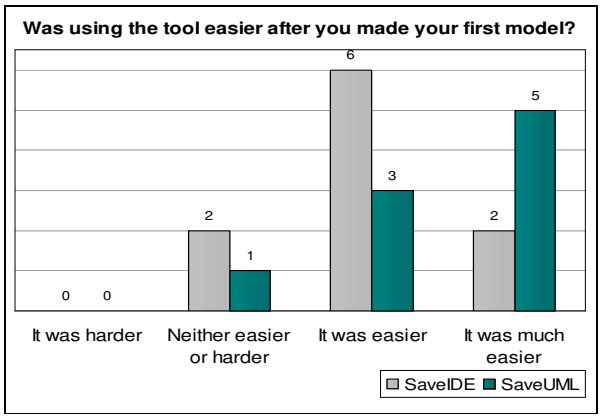


Figure 6. Effort made after adaption period

Do extraneous UML elements and semantics confuse developers and lead to an invalid or incomplete SaveCCM model?

The group using SaveUML found it harder to get accustomed with using the RSM tool to model SaveCCM. From their comments we concluded that all the features that the RSM tool provides initially confused them. In addition, UML profile extends UML, but does not repress the usage of non-extended part of UML. The availability of various UML elements which do not belong to SaveCCM domain was the cause of most mistakes that students from

SaveUML group made in their models.

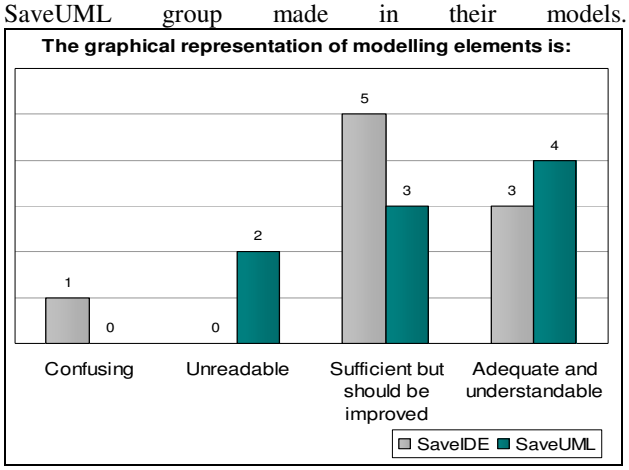


Figure 7. Perspicuity of graphical representation

The common mistakes were:

- Using of wrong relationships for connecting model elements.
- Not setting the properties (e.g. data type of ports) through the tagged value mechanism (setting of properties using tagged values is not as intuitive as setting of standard properties).

However, after the training phase almost all of the participants got acquainted with the tool, and these mistakes were rare.

Is using of SaveUML profile efficient with regard to time consumption, in comparison to using SaveIDE?

The usage of SaveUML profile had a negative impact on time consumption (Table 2). This inefficiency arises from the fact that it takes many steps to accomplish a simple operation. For example to add a SaveCCM component to the model, first a UML component has to be added to the model, then an appropriate stereotype from the UML profile has to be applied and finally component attributes can be set.

As a conclusion, while giving the overall experience of using the tool, even though they reported more working hours, RSM group also rated the experience as better than the SaveIDE group.

TABLE 2. OVERALL EXPERIENCE OF USING SAVEUML AND SAVEIDE

Technology	Working hours			Overall grade
	Avg.	Min.	Max.	
SaveIDE	6.28	3	15	3.78
SaveUML	8.22	4	13	4.11

Overall results

By this experiment we have indicated usefulness of the approach of adoption of a general purpose tool instead of a DSL (or in this concrete example of creating UML profile SaveUML as an alternative to SaveIDE). This experiment was focused on feasibility and usability of the design phase of component-based systems. We have not tested usability in using the specifications documents and means for exchanging information between users. Neither have we evaluated a larger scope of the lifecycle that includes analysis and verification part which requires repetitive transformation between the tools. The approach cannot be generalised in the sense that such approach is always better or feasible, but the experience indicates this possibility with pointing out the possible challenges.

VI. RELATED WORK

Many researchers have tried to accomplish linking of UML with some DSLs. For instance, Polak and Mencl developed a mapping from UML 2.0 to SOFA and Fractal research component models [14]. The approach also uses UML profiles for designing UML models and a tool prototype generates SOFA and Fractal source code from UML model. Contrary to SaveUML profile, the UML profile they created is used only to define new UML metaclasses using stereotypes and tagged values, while constraints (defined by OCL) do not exist.

The work by Malavolta et at. [15], is not limited to particular modelling languages. The automated framework called DUALLy creates interoperability among various ADLs, as well as UML. DUALLy is partitioned to two abstraction levels, separating meta-model definition process and system development. The transformations between languages are not done directly but there is a central *A_0 model* using as a intermediate step of every transformation. A_0 is a UML profile and it represents a semantic core set of architectural elements (e.g. components, connectors, behaviour). It provides the infrastructure upon which to construct semantic relations among different ADL and acts as a bridge among architectural languages. The disadvantage of this approach is that defined mappings are not injective, thus the unique reverse transformation is not ensured.

VII. CONCLUSION

In this paper we presented a simple approach for achieving modelling language interoperability between UML and a domain specific language SaveCCM. The main idea is creating a UML profile to allow developing UML models with domain-specific semantics. Further, a transformation tool for such model to SaveCCM is implemented, which makes it possible to use analysis of timing and other properties. The transformation is achieved using XML representations of models as an input for XSLT style sheets. The proposed approach fosters combining of GPL and DSL at different design stages. Some of the benefits are making a good use of advantages of both languages which improves design productivity, portability of the model as well as already mentioned standardization. We have validated the feasibility of the approach and usability of the UML profile-based tool in comparison to SaveCCM tool, and found that in spite of quite different characteristics of

models, adopting a general purpose tool in this case was feasible since in the experiment similar results were achieved by both tools.

ACKNOWLEDGEMENT

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and the Unity Through Knowledge Fund supported by Croatian Government and the World Bank via the DICES project, and EU FP7 Q-Impress project.

REFERENCES

[1] Akerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., and Tivoli, M. 2007. The SAVE approach to component-based development of vehicular systems. *J. Syst. Softw.* 80, 5 (May. 2007), 655-667. DOI= http://dx.doi.org/10.1016/j.jss.2006.08.016

[2] Åkerholm, M., Carlson, J., Håkansson, J., Hansson, H., Nolin, M., Nolte, T., and Pettersson, P. 2007. The SaveCCM Language Reference Manual. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, January, 2007.

[3] OMG 2007. Unified Modelling Language Superstructure Specification. Version 2.1.1, February, 2007., http://www.omg.org/uml/

[4] Sentilles, S., Hakansson, J., Pettersson, P., and Crnkovic, I. 2008. Save-IDE – An Integrated development environment for building predictable component-based embedded systems. Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), L'Aquila, Italy, September, 2008.

[5] Sentilles, S., Pettersson, A., Nyström, D., Nolte, T., Pettersson, P., and Crnkovic, I., 2009. Save-IDE - A Tool for Design, Analysis and Implementation of Component-Based Embedded Systems, Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE), Vancouver, May, 2009

[6] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer, 1:134–152, 1997

[7] Magee, J., Kramer, J., 1999. Concurrency: State Models & Java Programs, John Wiley & Sons, Inc., New York, NY, USA.

[8] Hansson, H., Akerholm, M., Crnkovic, I., and Torngren, M. 2004. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In Proceedings of the 30th EUROMICRO Conference (August 31 - September 03, 2004). EUROMICRO. IEEE Computer Society, Washington, DC, 627-635. DOI= http://dx.doi.org/10.1109/EUROMICRO.2004.72

[9] IBM Rational Software Modeller web page: http://www.ibm.com/software/awdtools/modeler/swmodeler/, April 2008

[10] Selic, B. 2007. A Systematic Approach to Domain-Specific Language Design Using UML. In Proceedings of the 10th IEEE international Symposium on Object and Component-Oriented Real-Time Distributed Computing (May 07 - 09, 2007). ISORC. IEEE Computer Society, Washington, DC, 2-9. DOI= http://dx.doi.org/10.1109/ISORC.2007.10

[11] W3C 1999. XSL Transformations (XSLT). Version 1.0, W3C Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[12] Visser, E. 2001. A Survey of Strategies in Program Transformation Systems. Electronic Notes in Theoretical Computer Science, eds. Gramlich and Lucas, vol. 57, Elsevier, 2001.

[13] Petričić, A., Lednicki, L., Crnkovic, I., 2009. An empirical comparison of SaveCCM and SaveUML technologies, http://www.mrtc.mdh.se/publications/1621.pdf, MRTC, Mälardalen University. March, 2009.

[14] Polak, M. 2005. UML 2.0 Components, Master Thesis, advisor: Vladimir Mencl, Charles Univ., Prague, September, 2005.

[15] Malavolta, I., Muccini, H. and Pelliccione, P. 2008. DUALLY: a framework for Architectural Languages and Tools Interoperability. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE2008). September 15-19 2008 L'Aquila, Italy. IEEE Press.

[16] Mencl, V., and Polak, M. 2006. UML 2.0 Components and Fractal: An Analysis. 5th Fractal Workshop (part of ECOOP'06), July 3rd, 2006, Nantes, France, Jul. 2006.